Java byte keyword

Java case keyword

Java char keyword

Java class keyword

Java double keyword

Java float keyword

AD

← prev next →

Memory Management in Java

In Java, memory management is the process of allocation and de-allocation of objects, called Memory management. Java does memory management automatically. Java uses an automatic memory management system called a **garbage collector**. Thus, we are not required to implement memory management logic in our application. Java memory management divides into two major parts:

- **JVM Memory Structure**
- **Working of the Garbage Collector**

JVM Memory Structure

JVM creates various run time data areas in a heap. These areas are used during the program execution. The memory areas are destroyed when JVM exits, whereas the data areas are destroyed when the thread exits.

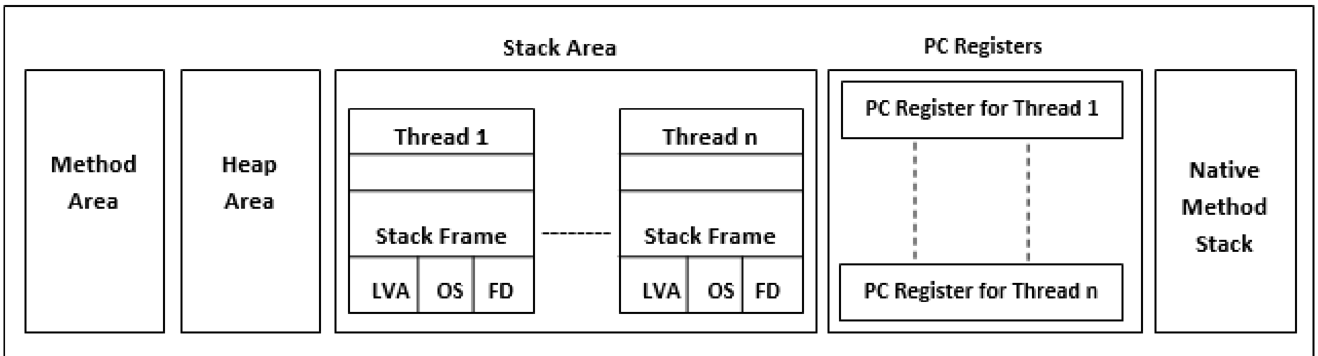


Figure: JVM Memory Structure

Method Area

Method Area is a part of the heap memory which is shared among all the threads. It creates when the JVM starts up. It is used to store class structure, superclass name, interface name, and constructors. The JVM stores the following kinds of information in the method area:

- A Fully qualified name of a type (ex: String)
- The type's modifiers
- Type's direct superclass name
- A structured list of the fully qualified names of super interfaces.

AD

AD
Bengaluru to Kolkata
from Bengaluru
from Rs8,455
Book

Heap Area

Heap stores the actual objects. It creates when the JVM starts up. The user can control the heap if needed. It can be of fixed or dynamic size. When we use a new keyword, the JVM creates an instance for the object in a heap. While the reference of that object stores in the stack. There exists only one heap for each running JVM process. When heap becomes full, the garbage is collected. For example:

```
StringBuilder sb= new StringBuilder();
```

The above statement creates an object of the StringBuilder class. The object allocates to the heap, and the reference sb allocates to stack. Heap is divided into the following parts:

- Young generation
- Survivor space
- Old generation
- Permanent generation

- Code Cache

Young Generation

The **Young Generation** is the section of Java heap memory where new objects are initially allocated. When this area becomes full, a **Minor Garbage Collection** (GC) is triggered. The Young Generation is structured into the following three parts:

- **Eden Memory and**
- **Two Survivor Memory spaces**

Let's discuss the above two in detail.

- **Eden Memory Space**: It is the primary location for allocating newest objects.
- **Survivor Spaces**: Upon filling up, the Eden space triggers a Minor GC, during which objects that are still in use are relocated to one of the survivor spaces. As a result, one survivor space is always maintained empty to accommodate these surviving objects in future collections.
- **Object Promotion**: Objects that persist through multiple garbage collection cycles in the survivor spaces are eventually promoted to the Old Generation. Promotion typically occurs after the objects have reached a certain age, determined by a preset threshold.

Old Generation

The **Old Generation**, also known as the **Tenured Generation**, is a key area in Java's heap memory where long-lived objects are stored. It comes into play when objects survive multiple garbage collection cycles in the Young Generation. Unlike the Young Generation, the Old Generation consists of a single, continuous memory space.

Garbage Collection (GC) in the Old Generation:

Major GC/ Full GC: These are more comprehensive and time-consuming events that occur when the Old Generation is full. They clean out both the Young and Old Generations and may involve memory compaction to optimize space usage, often leading to longer pause times.

Permanent Generation

The **Permanent Generation**, often referred to as **Perm Gen**. It is a dedicated area in the Java Virtual Machine (JVM) that holds metadata about the application's classes and methods. It is important to note that Perm Gen is separate from the Java Heap, where instance data is stored.

This region is dynamically filled by the JVM during runtime, depending on the classes that are actively used by the application, and it includes classes and methods from the Java SE libraries as well. Objects stored in Perm Gen are subject to garbage collection during a full garbage collection cycle, allowing for the clean-up of class and method metadata that is no longer needed.

Java Heap Memory Switches

1. -xms

Description: It sets the initial heap size when the JVM starts. It is the amount of memory the JVM attempts to allocate for its heap at startup.

Example: -xms512m starts the JVM with an initial heap size of 512 megabytes.

2. -xmx

Description: Sets the maximum heap size. It limits the maximum amount of memory that the JVM can allocate for its heap, affecting how much memory your Java application can use for all its objects and processes.

AD

Example: -xmx2g caps the heap size at 2 gigabytes.

3. -xmn

Description: It sets the size of the Young Generation. The remainder of the heap (outside of the Young Generation) is dedicated to the Old Generation.

Example: -xmn256m allocates 256 megabytes to the Young Generation.

4. -XX:PermGen

AD

Description: Sets the initial size of the Permanent Generation, which is used to store the JVM's metadata, such as classes and method data. Note that this setting is relevant only for JVMs before Java 8, after which PermGen was replaced by Metaspace.

Example: -XX:PermGen=128m sets the initial size of PermGen to 128 megabytes.

5. -XX:MaxPermGen

Description: Sets the maximum size of the Permanent Generation. Controlling this limit is crucial to prevent **OutOfMemoryError** that occurs when the JVM runs out of space in PermGen.

Example: -XX:MaxPermGen=256m caps the PermGen size at 256 megabytes.

6. -XX:SurvivorRatio

Description: it specifies the ratio between Eden and Survivor spaces within the Young Generation. If the Young Generation size is 10m and the Survivor Ratio is set to 2, then 5m will be reserved for the Eden Space, and 2.5m each will be reserved for the two Survivor spaces.

Example: -XX:SurvivorRatio=6 means the Eden space will be six times the size of each Survivor space.

7. -XX:NewRatio

Description: It provides a ratio of Old to New (Young) Generation sizes. A ratio of 2 means the Old Generation is twice the size of the Young Generation.

Example: -XX:NewRatio=3 sets the Old Generation to three times the size of the Young Generation.

AD

Reference Type

There are four types of references: **Strong, Weak, Soft**, and **Phantom reference**. The difference among the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria.

Strong reference: It is very simple as we use it in our daily programming. Any object which has Strong reference attached to it is not eligible for garbage collection. We can create a strong reference by using the following statement:

```
StringBuilder sb= new StringBuilder();
```

Weak Reference: It does not survive after the next garbage collection process. If we are not sure when the data will be requested again, in this condition, we can create a weak reference to it. In case, if the garbage collector processes, it destroys the object. When we again try to retrieve that object, we get a null value. It is defined in **java.lang.ref.WeakReference** class. We can create a weak reference by using the following statement:

```
WeakReference<StringBuilder> reference = new WeakReference<>
(new StringBuilder());
```

Soft Reference: It is collected when the application is running low on memory. The garbage collector does not collect the softly reachable objects. All soft referenced objects are collected before it throws an OutOfMemoryError. We can create a soft reference by using the following statement:

```
SoftReference<StringBuilder> reference = new SoftReference<>
(new StringBuilder());
```

Phantom Reference: It is available in **java.lang.ref** package. It is defined in **java.lang.ref.PhantomReference** class. The object which has only phantom reference pointing them can be collected whenever garbage collector wants to collect. We can create a phantom reference by using the following statement:

```
PhantomReference<StringBuilder> reference = new PhantomReference<>
(new StringBuilder());
```

Stack Area

Stack Area generates when a thread creates. It can be of either fixed or dynamic size. The stack memory is allocated per thread. It is used to store data and partial results. It contains references to heap objects. It also holds the value itself rather than a reference to an object from the heap. The variables which are stored in the stack have certain visibility, called scope.

Stack Frame: Stack frame is a data structure that contains the thread's data. Thread data represents the state of the thread in the current method.

- It is used to store partial results and data. It also performs dynamic linking, values return by methods and dispatch exceptions.
- When a method invokes, a new frame creates. It destroys the frame when the invocation of the method completes.
- Each frame contains own Local Variable Array (LVA), Operand Stack (OS), and Frame Data (FD).
- The sizes of LVA, OS, and FD determined at compile time.
- Only one frame (the frame for executing method) is active at any point in a given thread of control. The frame is called the current frame, and its method is known as the current method. The class of method is called the current class.
- The frame stops the current method, if its method invokes another method or if the method completes.
- The frame created by a thread is local to that thread and cannot be referenced by any other thread.

Native Method Stack

It is also known as C stack. It is a stack for native code written in a language other than Java. Java Native Interface (JNI) calls the native stack. The performance of the native stack depends on the OS.

PC Registers

Each thread has a Program Counter (PC) register associated with it. PC register stores the return address or a native pointer. It also contains the address of the JVM instructions currently being executed.

AD

Working of Garbage Collector

Garbage Collector Overview

When a program executes in Java, it uses memory in different ways. The heap is a part of memory where objects live. It iss the only part of memory that involved in the garbage collection process. It is also known as garbage collectible heap. All the garbage collection makes sure that the heap has as much free space as possible. The function of the garbage collector is to find and delete the objects that cannot be reached.

Object Allocation

When an object allocates, the JRockit JVM checks the size of the object. It distinguishes between small and large objects. The small and large size depends on the JVM version, heap size, garbage collection strategy, and platform used. The size of an object is usually between 2 to 128 KB.

The small objects are stored in Thread Local Area (TLA) which is a free chunk of the heap. TLA does not synchronize with other threads. When TLA becomes full, it requests for new TLA.

On the other hand, large objects that do not fit inside the TLA directly allocated into the heap. If a thread is using the young space, it directly stored in the old space. The large object requires more synchronization between the threads.

What does Java Garbage Collector?

JVM controls the garbage collector. JVM decides when to perform the garbage collection. We can also request to the JVM to run the garbage collector. But there is no guarantee under any conditions that the JVM will comply. JVM runs the garbage collector if it senses that memory is running low. When Java program request for the garbage collector, the JVM usually grants the request in short order. It does not make sure that the requests accept.

The point to understand is that "**when an object becomes eligible for garbage collection?**"

EEvery Java program has more than one thread. Each thread has its execution stack. There is a thread to run in Java program that is the main() method. Now we can say that an object is eligible for garbage collection when no live thread can access it. The garbage collector considers that object as eligible for deletion. If a program has a reference variable that refers to an object, that reference variable available to live thread, this object is called **reachable**.

Here a question arises that "**Can a Java application run out of memory?**"

The answer is yes. The garbage collection system attempts to objects from the memory when they are not in use. Though, if we are maintaining many live objects, garbage collection does not guarantee that there is enough memory. Only available memory will be managed effectively.

Java Garbage Collection Monitoring

Monitoring garbage collection in Java applications is crucial for performance tuning and resource management. For a practical example, here we will demonstrate how to monitor GC activities using a demo application available from the Java SE downloads. The application I am using is **Java2Demo.jar**, located in the **jdk1.7.0_55/demo/jfc/Java2D** directory after downloading JDK 7 and JavaFX Demos and Samples. However, we can apply these monitoring techniques to any Java application.

Starting the Demo Application

To run the Java2Demo application with specific memory settings, use the following command:

```
pankaj@Pankaj:~/Downloads/jdk1.7.0_55/demo/jfc/Java2D$ java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar Java2Demo.jar
```

jstat

The jstat command-line tool is a valuable resource for monitoring JVM memory and garbage collection activities. Included with the standard JDK installation, jstat provides detailed insights without the need for additional software. To use jstat, you'll first need to find the process ID of the Java application you want to monitor. You can easily retrieve this information by executing the ps -eaf | grep java command in your terminal. This step will help you identify the specific Java process to monitor with jstat.

```
pankaj@Pankaj:~$ ps -eaf | grep Java2Demo.jar
  501 9582  11579   0  9:48PM ttys000    0:21.66 /usr/bin/java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseG1GC -jar Java2Demo.jar
  501 14073 14045   0  9:48PM ttys002    0:00.00 grep Java2Demo.jar
```

Now that we have identified the process ID for the Java application, which is 9582, we can proceed to monitor it using the jstat command. Here's how you can run the command:

```
pankaj@Pankaj:~$ jstat -gc 9582 1000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       PC     PU    YGC     YGCT    FGC    FGCT     GCT
1024.0 1024.0  0.0    0.0    8192.0   7933.3   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654
1024.0 1024.0  0.0    0.0    8192.0   8026.5   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654
1024.0 1024.0  0.0    0.0    8192.0   8030.0   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654
1024.0 1024.0  0.0    0.0    8192.0   8122.2   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654
1024.0 1024.0  0.0    0.0    8192.0   8171.2   42108.0    23401.3   20480.0 19990.9    157    0.274  40      1.381    1.654
1024.0 1024.0  48.7   0.0    8192.0   106.7    42108.0    23401.3   20480.0 19990.9    158    0.275  40      1.381    1.656
1024.0 1024.0  48.7   0.0    8192.0   145.8    42108.0    23401.3   20480.0 19990.9    158    0.275  40      1.381    1.656
```

The final parameter for the **jstat** command specifies the frequency of output, so setting it to 1000 milliseconds means it will report memory and garbage collection data every second. Let's explore what each column in the output represents:

- **S0C and S1C**: These columns indicate the current size of the Survivor0 and Survivor1 spaces in kilobytes.
- **S0U and S1U**: These columns show the current usage of the Survivor0 and Survivor1 spaces in kilobytes. It's important to note that typically one of the survivor spaces is always empty.
- **EC and EU**: These columns represent the current size and usage of the Eden space in kilobytes. Observe that the EU (Eden Usage) size increases and when it surpasses the EC (Eden Capacity), a Minor GC is triggered, which then reduces the EU size.
- **OC and OU**: These columns display the current size and usage of the Old Generation in kilobytes.
- **PC and PU**: These columns reflect the current size and usage of the Perm Gen in kilobytes.

- **YGC and YGCT**: The YGC column counts the number of garbage collection events that have occurred in the Young Generation, while the YGCT column measures the total time spent on these operations. Both metrics typically increase together, especially when a drop in EU indicates a Minor GC event.
- **FGC and FGCT**: The FGC column tallies the number of Full GC events, and the FGCT column tracks the accumulated time spent on these operations. Note that the time for Full GC events is generally much longer than for Young Generation GC.
- **GCT**: This column sums the total accumulated time spent on all garbage collection operations, combining both YGCT and FGCT times.

Types of Garbage Collection

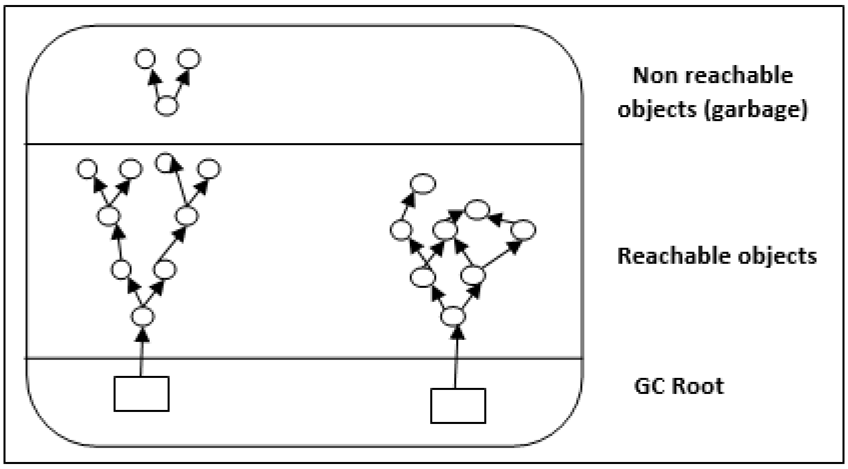
There are five types of garbage collection are as follows:

- **Serial GC:** It uses the mark and sweeps approach for young and old generations, which is minor and major GC.
- **Parallel GC:** It is similar to serial GC except that, it spawns N (the number of CPU cores in the system) threads for young generation garbage collection.
- **Parallel Old GC:** It is similar to parallel GC, except that it uses multiple threads for both generations.
- **Concurrent Mark Sweep (CMS) Collector:** It does the garbage collection for the old generation. You can limit the number of threads in CMS collector using **XX:ParalleCMSThreads=JVM option**. It is also known as Concurrent Low Pause Collector.
- **G1 Garbage Collector:** It introduced in Java 7. Its objective is to replace the CMS collector. It is a parallel, concurrent, and CMS collector. There is no young and old generation space. It divides the heap into several equal sized heaps. It first collects the regions with lesser live data.

Mark and Sweep Algorithm

JRockit JVM uses the mark, and sweep algorithm for performing the garbage collection. It contains two phases, the mark phase, and the sweep phase.

Mark Phase: Objects that are accessible from the threads, native handles, and other GC root sources are marked as live. Every object tree has more than one root objects. GC root is always reachable. So any object that has a garbage collection root at its root. It identifies and marks all objects that are in use, and the remaining can be considered garbage.



Sweep Phase: In this phase, the heap is traversed to find the gap between the live objects. These gaps are recorded in the free list and are available for new object allocation.

There are two improved versions of mark and sweep:

- **Concurrent Mark and Sweep**
- **Parallel Mark and Sweep**

Concurrent Mark and Sweep

It allows the threads to continue running during a large portion of the garbage collection. There are following types of marking:

- **Initial marking:** It identifies the root set of live objects. It is done while threads are paused.
- **Concurrent marking:** In this marking, the reference from the root set are followed. It finds and marks the rest of the live objects in a heap. It is done while the thread is running.
- **Pre-cleaning marking:** It identifies the changes made by concurrent marking. Other live objects marked and found. It is done while the threads are running.
- **Final marking:** It identifies the changes made by pre-cleaning marking. Other live objects marked and found. It is done while threads are paused.

Parallel Mark and Sweep

It uses all available CPU in the system for performing the garbage collection as fast as possible. It is also called the parallel garbage collector. Threads do not execute when the parallel garbage collection executes.

Pros of Mark and Sweep

- It is a recurring process.
- It is an infinite loop.
- No additional overheads allowed during the execution of an algorithm.

Cons of Mark and Sweep

- It stops the normal program execution while the garbage collection algorithm runs.
- It runs multiple times on a program.

Next Topic Java Tutorial

← prev next →

AD

Learn Important Tutorial