**Heap Memory**:

* The **heap memory** in the JVM is divided into **Young Generation**, **Old Generation**, and **Permanent Generation** (or **Metaspace** in later versions of Java).

1. **Young Generation**

* **Young Generation** serves to handle **short-lived objects** and uses **Minor GC** for cleanup.
* **Purpose**: The Young Generation is optimized for fast garbage collection because most objects are short-lived and do not survive long enough to be promoted to the Old Generation.
* **Eden Space**: When an object is first created, it is allocated in the **Eden space.**
* **Survivor Space 1 (S1)** and **Survivor Space 2 (S2)**:
* **Minor Garbage Collection**: When the Eden space fills up, a minor GC is triggered. This will only collect objects from the Young Generation and may promote surviving objects to the Survivor space or, if they are old enough, to the Old Generation.

2. **Old Generation (Tenured Space)**

* The **Old Generation** is where long-lived objects are stored and uses **Major GC** for cleaned up.
* **Major Garbage Collection**: A **full GC** or **major GC** occurs when the Old Generation fills up. This type of GC is more expensive because it involves both the Old and Young Generations. The garbage collector will clean up unreachable objects and may also compact memory in the Old Generation.
* **Purpose**: The Old Generation holds objects that are expected to live longer and are garbage collected less frequently.

3. **Permanent Generation** (Before Java 8) / **Metaspace** (From Java 8)

* **Permanent Generation (PermGen)**: The **PermGen** was a fixed-size memory area, and if it filled up, you would get an OutOfMemoryError: PermGen space.
* Metaspace: In Java 8, the **Permanent Generation** was removed and replaced with **Metaspace**.
* Metaspace stores JVM class metadata, and other JVM internals like class loaders and method data.
* **Purpose**: The **Metaspace** (or **PermGen** in older versions) holds JVM internal structures like class and method metadata, so it is crucial for the operation of the JVM itself.

4. **Other JVM Memory Areas**

* **Stack Memory:** Each thread has its own stack, which holds local variables and method call information. When a method is called, a new stack frame is created, and when the method exits, the stack frame is popped off the stack.
* **Code Cache:** Code cache helps improve performance by avoiding repeated compilation of the same bytecode.

A diagram of a computer program

AI-generated content may be incorrect.

**Method Area**

* All the class data is available in the method area.
* Corresponding static variables associated with the class are also part of the Method Area.
* Its size can be customized as per use.
* Apart from the class data, it also stores superclass name, interface name, and constructors information
* If the memory in the method area can’t satisfy an allocation request, JVM throws an **OutOfMemoryError**.

**Heap Area**

* All the object data is available in the Heap area.
* The instance variable will also be stored in Heap Area.
* It is very important for a programmer to understand the Heap Area as all the object-level information is present here and garbage collection keeps a check for this area only.
* Its size can be customized as per need.
* If more heap is required for the storage management system, JVM throws an **OutOfMemoryError**.

**3. Stack Area:**

* For every thread, a separate run-time stack is created and is stored in Stack Area.
* Thus, if we have "n" number of threads, "n" number of stacks will be created and will be stored in the stack area.
* Further, each entry in each stack contains three parts i.e. Local variable array, operand stack, and Frame data, and these three are collectively known as a stack frame.
* If more JVM stack memory is required than is permitted, JVM throws a **StackOverflowError**. On the other hand, if JVM tried to expand the Stack memory or if there is a shortage of memory during initial JVM stack creation for a new thread, JVM throws **OutOfMemoryError**.

**Relation between Stack and Heap:**

|  |  |
| --- | --- |
|  | String s = **new** String("abc"); |

* In the above syntax, an object of type String and value "abc" will be created in the Heap area.
* "s" will be created in Stack which will refer to the object created in the Heap area.

**4. PC Registers:**

* For each thread, a stack is created in the stack area and a PC register is created in PC Registers memory.
* Thus, if we have "n" number of threads, then again "n" number of stacks will be created in the stack area, and "n" number of PC registers will also be created.
* It holds the address of the currently executing instruction and it moves to the next instruction once the current instruction is executed.

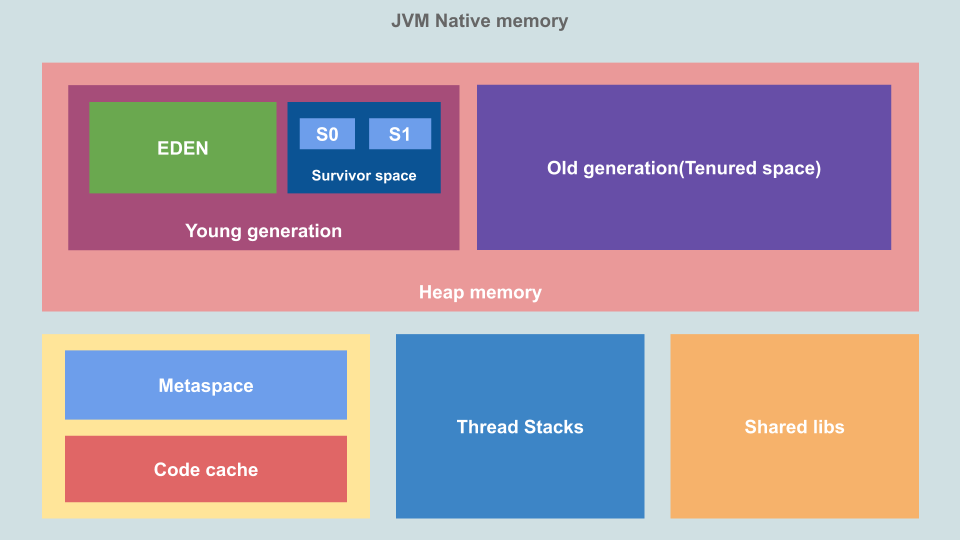
**Relation between Stack Area and PC Registers:**

* For each thread created in Stack Area, a program counter is associated with it.
* PC register stores the address of the current running instruction.

**5. Native Method Stack**

* It is similar to Stack Area but the difference is that the native methods of JAVA are stored in this area.
* If we deep dive into the architecture of JVM, we will find that there is a JNI (Java Native Interface) which deals with all the native methods.
* If more JVM native stack memory is required than is permitted, JVM throws a StackOverflowError. On the other hand, if JVM tried to expand the native stack memory or if there is a shortage of memory during the initial creation for a new thread, JVM throws OutOfMemoryError.

public class MemoryAreasExample {  
  
 // Static variable stored in the Method Area  
 static int staticVariable = 10;  
  
 public static void main(String[] args) {  
 // Local variable stored in the Stack Memory  
 int localVar = 20;  
  
 // Creating an object, and the object itself is stored in the Heap Memory  
 MyClass obj = new MyClass("Example Object");  
  
 // Method invocation, and the method and local variables are stored in the Stack Memory  
 obj.printMessage("Hello, Java!");  
  
 // Static method invocation, and the method is stored in the Method Area  
 staticMethod();  
  
 // Creating another object, and the object itself is stored in the Heap Memory  
 MyClass anotherObj = new MyClass("Another Object");  
  
 // Local variable stored in the Stack Memory  
 int anotherLocalVar = 30;  
 }  
  
 // Static method stored in the Method Area  
 static void staticMethod() {  
 System.out.println("Inside staticMethod");  
 }  
}  
  
class MyClass {  
 // Instance variable stored in the Heap Memory (part of the object)  
 String message;  
  
 // Constructor stored in the Heap Memory  
 public MyClass(String message) {  
 this.message = message;  
 }  
  
 // Instance method stored in the Heap Memory (part of the object)  
 void printMessage(String additionalMessage) {  
 System.out.println(message + ": " + additionalMessage);  
 }  
}



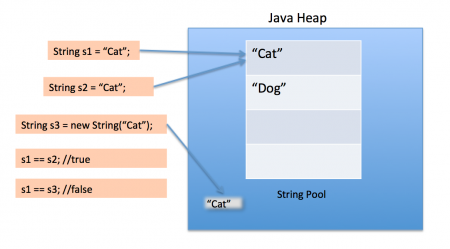
**Metaspace:**

* PermGen Space has been removed. So where will the metadata information be stored now? This metadata is now stored in a native memory are called as "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.

|  |  |
| --- | --- |
| **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 inefficient Garbage collection. Frequent GC pauses and no concurrent deallocation. | Comparatively efficient Garbage collection. Deallocate class data concurrently and not during GC pause. |

A diagram of a computer program

Description automatically generated with medium confidence

**String Pool in Java**  


String Pool is possible only because [String is immutable in Java](https://www.journaldev.com/802/string-immutable-final-java) and it’s implementation of [String interning](https://en.wikipedia.org/wiki/String_interning) concept. String pool is also example of [Flyweight design pattern](https://www.journaldev.com/1562/flyweight-design-pattern-java).

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to other String object from string pool having same value.

String str = new String("Cat");

// str points to a new object on the heap, even though "Cat" is in the string pool

In above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so total 2 string objects will be created.

String str = new String("Cat").intern()

If we want to store the string literal in the String pool we should use the intern() method.

//returns false because s1 occupies space in string pool and s3 occupies space in Java heap

String s1 = "Cat";  
String s2 = "Cat";  
  
String s3 = new String("Cat");  
String s4 = new String("Cat").intern();  
  
System.*out*.println(s1 == s2);//true  
System.*out*.println(s1 == s3);//true  
System.*out*.println(s1 == s4);//false  
System.*out*.println(s2 == s4);//true  
System.*out*.println(s1 == s4);//true

**Interning** is the process of creating a string object in String Constant Pool which will be exact copy of string object in heap memory.

**Java Profiler**

* A **Java Profiler** is a tool used to analyze and monitor the performance of a Java application
* A **Java profiler** is an essential tool for identifying performance bottlenecks, memory leaks, and inefficiencies in your Java application
* **CPU Profiling**: Tracks the amount of CPU time consumed by each method or class in your application.
* **Memory Profiling**: Helps in identifying memory leaks or excessive memory usage.
* **Thread Profiling**: Monitors the execution of threads and how they are scheduled.
* **Garbage Collection Analysis**: Helps identify unnecessary memory allocations that might lead to excessive GC pauses.
* **Database Profiling**: Useful for identifying slow or unnecessary database calls within your application.
* **Popular Profiles**: VisualVM, YourKit, JProfiler and Flight Recorder

**Memory Leakage and Application performance**

AppPerfect Java Profiler provides you a unified view of all resources that is memory, methods, threads and monitors to help you understand your JVM's behavior, performance characteristics etc. to reach to bottom of problem quickly. AppPerfect Java Profiler is a comprehensive Java performance tuning solution with support for Heap Analysis, Thread Analysis, Monitor analysis and Deadlocks detection. It is one of the best and finest profiler available in the large market of profilers.

Profiling is the process of monitoring various JVM level parameters such as Method Execution, Thread Execution, Object Creation and Garbage Collection. It will provide you with a finer view of your target application execution and its resource utilization.

1. **Memory Profiling :** you can view all the classes/packages currently loaded in JVM

a) Class wise Allocations

b) Method wise Allocations

c) Garbage Collection

**2) CPU Profiling :** you can view complete summary about CPU Profiling

a) Methods

b) Invocation/Call Tree

**3) Thread Profiling :** you can view summary of all the threads running in your application

a) Threads

b) Monitors

c) Deadlocks

**Micro Services**

* Do not try to solve complex problems at once
* Break the big thing into several smaller parts
* You will end up with a number of small loosely coupled applications

**Benefits**

* It is easier to design a small application that only does one thing
* Testing is easier
* as well as deployment and maintenance
* individual parts could have different pages

**Garbage Collection:**

* Nullifying the reference variable
* Re Assigning the reference variable
* Objects created inside a method
* Island of isolation

**OutOfMemoryError in Java**

**1️ Java Heap Space Error**

📌 **Cause:**

* Heap memory is **exhausted** due to excessive object creation.
* Memory leaks prevent objects from being garbage collected.
* Large objects like **caching data, reading large files** into memory.

📌 **Solution:**

* **Increase Heap Size:** -Xms1g -Xmx4g
* **Use Memory-Efficient Collections** (e.g., WeakHashMap).
* Close **database connections, streams, and sockets** properly.
* Use **try-with-resources (AutoCloseable)** to prevent resource leaks.
* **Enable GC Logging:** -Xlog:gc\*
* **Analyze with JVisualVM or Heap Dump (-XX:+HeapDumpOnOutOfMemoryError)**

**2️ GC Overhead Limit Exceeded**

📌 **Cause:**

* JVM spends **more than 98% of CPU time on Garbage Collection** but reclaims **less than 2%** of memory.
* Occurs due to memory leaks or excessive object creation.

📌 **Solution:**

* **Increase Heap Size:** -XX:+UseG1GC -Xmx4g
* **Profile and Optimize Memory Usage:** Use **JProfiler**, **Eclipse MAT**, or **JVisualVM**.
* **Identify Memory Leaks** (e.g., unused objects, circular references).
* **Pass by Value (Ex: Java)**: The method parameter values are copied to another variable and then the copied object is passed, that’s why it’s called pass by value.
* **Pass by Reference**: An alias or reference to the actual parameter is passed to the method, that’s why it’s called pass by reference.

**Out of Memory Error (OOM)**

* The **OutOfMemoryError** occurs when the JVM **runs out of heap memory** and cannot allocate more objects.
* **Solution:** Increase heap size (-Xmx option), optimize memory usage, use weak references (WeakReference).

**Causes of OutOfMemoryError**

1. **Memory Leak:** Unreleased objects continue to occupy memory.
2. **Large Object Allocation:** Huge arrays or data structures.
3. **Excessive Heap Usage:** Holding large collections (List, Map) in memory.
4. **Improper JVM Heap Configuration:** Insufficient **-Xmx** (maximum heap size) setting.
5. **Infinite Loops / Unbounded Data Processing:** Running processes that keep accumulating objects.

**Types of OutOfMemoryError**

| **Type** | **Cause** |
| --- | --- |
| **Java heap space** | Objects exceed allocated heap memory. |
| **GC overhead limit exceeded** | JVM spends too much time in Garbage Collection. |
| **Metaspace out of memory** | Too many loaded classes (JVM Metaspace full). |
| **Direct buffer memory** | Exceeding allocated **off-heap** memory (e.g., NIO buffers). |
| **Requested array size exceeds VM limit** | Trying to create an extremely large array. |

**Stack Overflow Error**

* **StackOverflowError** occurs when a program **exceeds the stack memory** due to **deep recursion or infinite method calls**.

**Causes of StackOverflowError**

1. **Infinite Recursion:** A method calls itself without a proper termination condition.
2. **Excessively Deep Recursion:** Too many nested method calls.
3. **Large Stack Memory Usage:** Too many local variables or deeply nested loops.

**Solution:**

* Use **iterative** solutions instead of recursion where possible.
* Optimize recursive functions (e.g., **tail recursion**).
* Increase **stack size** using -Xss JVM option.

**OutOfMemoryError and StackOverflowError**

| **Feature** | **OutOfMemoryError** | **StackOverflowError** |
| --- | --- | --- |
| **Cause** | Heap memory exhausted | Stack memory exhausted |
| **Common Scenario** | Large objects, memory leaks | Deep recursion, infinite method calls |
| **Recovery** | Can be handled via catch block | Usually unrecoverable |
| **Solution** | Increase -Xmx, optimize memory | Reduce recursion, increase -Xss |

**Difference Between getInstance() and new**

| **Aspect** | **new Keyword** | **getInstance() Method** |
| --- | --- | --- |
| **Object Control** | Always creates new instance | May reuse existing instance |
| **Design Pattern** | Direct instantiation | Singleton/Factory pattern |
| **Initialization** | Immediate | Can be lazy |
| **Thread Safety** | Not inherently thread-safe | Can be made thread-safe |
| **Flexibility** | Less flexible | More flexible (can add logic) |
| **Use Case** | Regular objects | When controlled access is needed |