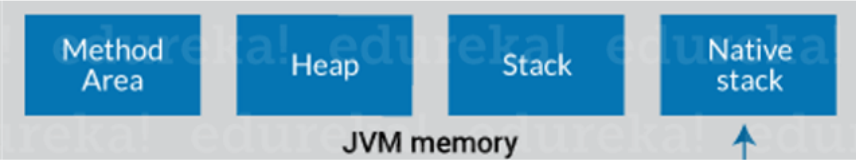
Week – 04

Memory allocation in java

Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations.

Memory allocation in java refers to the process where the computer programs and services are allocated dedicated to virtual memory spaces. The Java Virtual Machine divides the memory into

1. Method Area
2. Stack
3. Heap Memory.
4. Native Method Stack



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.

Stack Memory

The Stack Memory allocation in java is used for static memory and thread execution. The values contained in this memory are temporary and limited to specific methods as they keep getting referenced in Last-In-First-Out fashion.

As soon as the memory is called and a new block gets created in the stack memory, the stack memory then holds primitive values and references until the method lasts. After its ending, the block is flushed and is available for a new process to take place.

Java Heap Space

Mainly used by java runtime, Java Heap Space comes into play every time an object is created and allocated in it. The discrete function, like Garbage Collection, keeps flushing the memory used by the previous objects that hold no reference. For an object created in the Heap Space can have free access across the application.

It can be of fixed or dynamic size. When you 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.

We can break this memory model down into smaller parts, called generations, which are:

1. **Young Generation –** this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up.
2. **Old or Tenured Generation –** this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set, and when that threshold is reached, the object is moved to the old generation.
3. **Permanent Generation –** this consists of JVM metadata for the runtime classes and application methods.

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.

Garbage Collection

- In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors.**
- Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine(JVM). When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.
- But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**. The garbage collector is the best example of the *Daemon thread* as it is always running in the background.

How Does Garbage Collection in Java works?

- Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused or unreferenced object is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.
- The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

GC works in two simple steps, known as Mark and Sweep:

- **Mark –** this is where the garbage collector identifies which pieces of memory are in use and which aren't.
- **Sweep –** this step removes objects identified during the "mark" phase.

JVM has five types of *GC* implementations:

1. Serial Garbage Collector
   This is the simplest GC implementation, as it basically works with a single thread. As a result, **this *GC* implementation freezes all application threads when it runs**. Therefore, it's not a good idea to use it in multi-threaded applications, like server environments.

2. Parallel Garbage Collector
   It's the default *GC* of the *JVM*, and sometimes called Throughput Collectors. Unlike *Serial Garbage Collector*, it **uses multiple threads for managing heap space,** but it also freezes other application threads while performing *GC*.

3. Concurrent Mark Sweep (*CMS) Garbage Collector*
   The *Concurrent Mark Sweep (CMS)* implementation uses multiple garbage collector threads for garbage collection. It's designed for applications that prefer shorter garbage collection pauses, and can afford to share processor resources with the garbage collector while the application is running.

4. G1 Garbage Collector (G1GC)
   G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. It's available from the *JDK7 Update 4* and in later releases.

5. Z Garbage Collector (ZGC)
   ZGC has obtained the production status from Java 15 onwards, *ZGC* performs all expensive work concurrently, **without stopping the execution of application threads for more than 10 ms**, which makes it suitable for applications that require low latency.

Advantages:

- No manual memory allocation/deallocation handling because unused memory space is automatically handled by *GC*
- No overhead of handling ***Dangling Pointer***
- Automatic ***Memory Leak*** management

finalize() method in java

- The **finalize() method** is defined in **Object class** which is the super most class in Java. This method is called by **Garbage collector** just before they destroy the object from memory. The **Garbage collector** is used to destroy the object which is **eligible for garbage collection**.
- The **finalize() method** is a **protected** and **non-static method** of **Object** class. As you know Object is super most class and each class inherit the **Object class,** it means the **finalize() method** is available in each class.
- The **finalize() method** is called by the garbage collector for an object when **garbage collection** determines that there are no more references to the object.
- The garbage collection determines each object of the **class**, if there is no reference for the object it means it should be destroyed.

```
protected void finalize() throws Throwable { }
```

Practice session examples:

```
public class Data
{
	public static void main(String[] args)
	{
	Data obj = new Data();
	obj = null;
	System.gc();
	System.out.println("Done");
	}
	@Override
	protected void finalize()
	{
		System.out.println("finilize() method called");
	}
}
```

Output:
Done
finilize() method called

```
public class Data
{
	public static void main(String[] args)
	{
	String s = "Hello";
	s = null;
	System.gc();
	System.out.println("Garbage collector");
	}
	@Override
	protected void finalize()
	{
	System.out.println("finilize() method called");
	}
}
```

Output:
Garbage collector

As you can see the above program prints only **"Garbage collector".** You must think about why it's not printing the **"finalize() method called"**.
Because **Garbage Collector** makes a call to **finalize() method** of that class, whose object is eligible for Garbage collection. In the above example **s = null;** where 's' is the object of **String** class. So, Garbage collector called the **finalize() method** of **String** class.

```
public class Data
{
```

```
        public static void main(String[] args)
        {
        Data data = new Data();
        Data data1 = new Data();
        data1 = null;
        // Explicit call
        data.finalize();
        System.out.println("Garbage collector");
        System.gc(); //Implicit call to finilize() method
        }
        @Override
        protected void finalize()
        {
        System.out.println("finilize() method called");
        }
}
```

Output:
finilize() method called
Garbage collector
finilize() method called

If we are calling **finalize() method** explicitly, then **JVM** executes it as a normal method. It can't destroy the object or not able to remove the object from memory. This method is useful only with **Garbage Collector** to release the memory.

In the above example when we are calling the **finalize() method** explicitly it is not destroying any object. But when **System.gc()** it makes an implicit call to **finalize() method** it destroys the object **data1**.

Install memory monitoring tool and observe how JVM allocates memory [VisualVM Tool]

Java VisualVM can be used by Java application developers to troubleshoot applications and to monitor and improve the applications' performance. Java VisualVM can allow developers to generate and analyse heap dumps, track down memory leaks, and monitor garbage collection, and perform lightweight memory and CPU profiling.

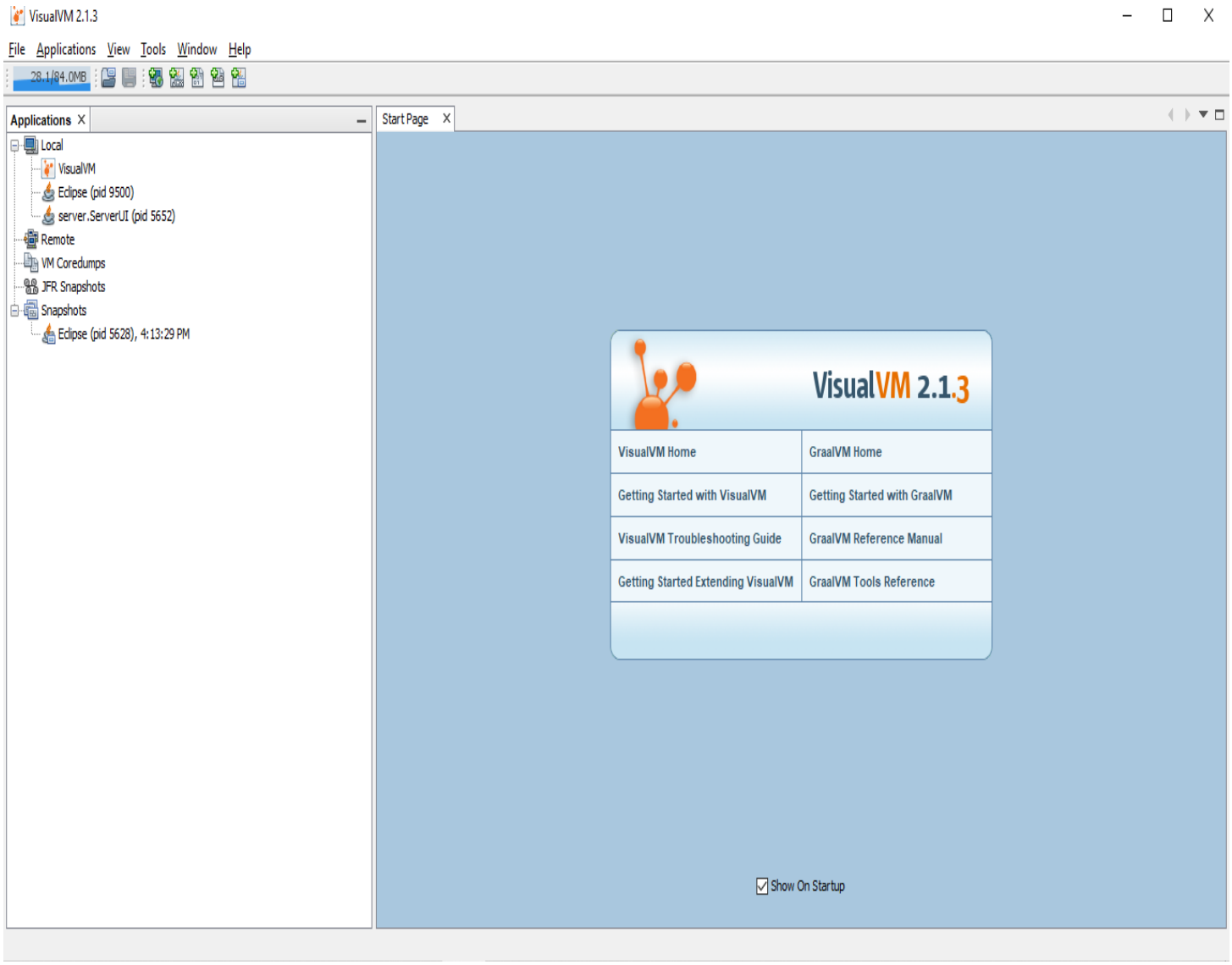
VisualVM is a powerful tool that **provides a visual interface to see deep and detailed information about local and remote Java applications while they are running on a Java Virtual Machine (JVM)**.

You can download the tool from the link→ https://drive.google.com/drive/folders/18vn-9fAMC1Cy-HMRYIb0skzCVs3dhF_I?usp=sharing

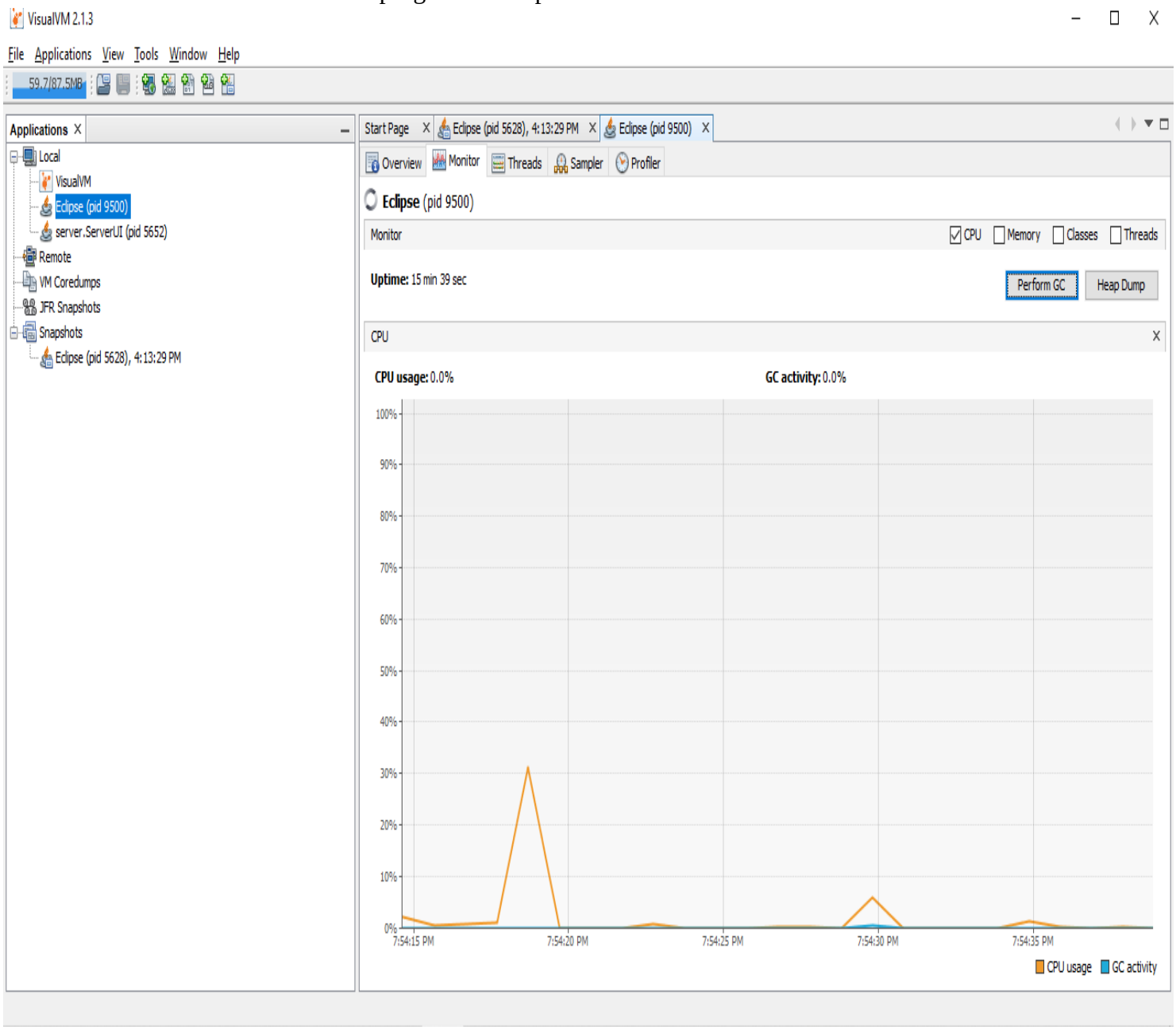
1. Once you download the folder of visualvm_213 go to bin folder inside visualvm_213 folder.
2. Click on visualvm Application file
3. Open the eclipse software and type the java program

```
public class HelloWorld
{
   int x=10;
        public static void main(String args[])
   {
            HelloWorld h1 = new HelloWorld();
            HelloWorld h2 = new HelloWorld();
            HelloWorld h3 = new HelloWorld();
          int a = (20*5)+(10/2)-(3 * 10);
           System.out.println("value of a is="+ a);
           System.out.println("value of x is="+ h1.x);
          h1=null;
          h2=null;
          h3=null;
   }
}
```

4. Open the visualvm



5. Now run the program in eclipse and watch the behaviour in visualvm tool



6. Now you can monitor the CPU, Memory, Classes and threads where the java applications running on JVM using VisualVm