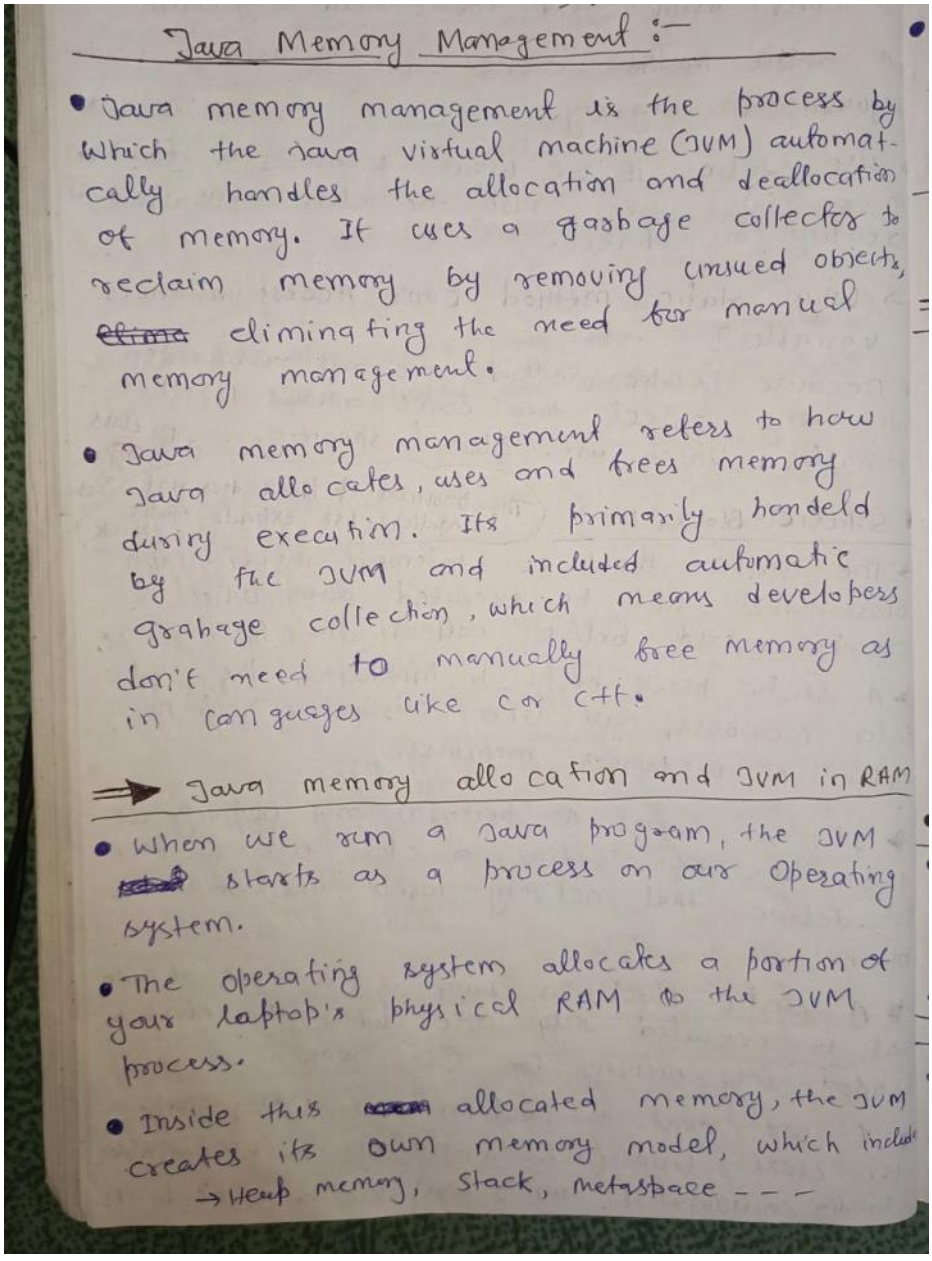
Memory Managements

31 October 2025 16:10



Java Memory Management :-

- Java memory management is the process by which the Java virtual machine (JVM) automatically handles the allocation and deallocation of memory. It uses a garbage collector to reclaim memory by removing unused objects, eliminating the need for manual memory management.
- Java memory management refers to how Java allocates, uses and frees memory during execution. Its primarily handeld by the JVM and included automatic garbage collection, which means developers don't need to manually free memory as in languages like C or C++.

→ Java memory allocation and JVM in RAM

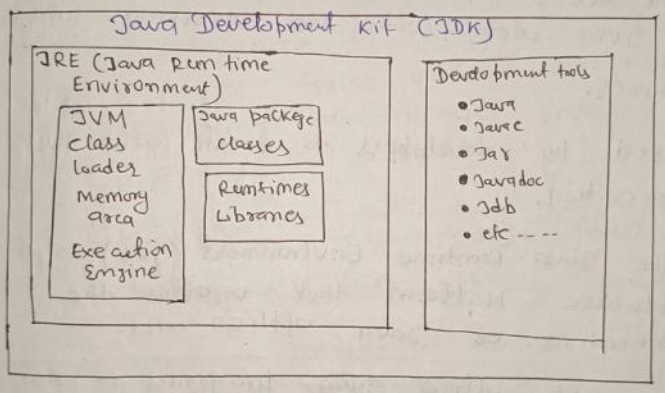
- When we run a Java program, the JVM starts as a process on our Operating system.
- The operating system allocates a portion of your laptop's physical RAM to the JVM process.
- Inside this allocated memory, the JVM creates its own memory model, which include
  → Heap memory, Stack, Metaspace - - -

All heap and stack memory of the JVM exists within the physical RAM of our laptop – the JVM doesn't create separate hardware memory.

- The operating system manages the overall RAM, while the RAM divides its allocated portion into heap, stack and other area of execution.

→ Both stack and Heap is created by JVM and stored in RAM.

⟹ JVM Memory Structure :–



- JDK (Java Development Kit) :–
- The JDK is a software development kit used to develop application in the Java programming language.
- It includes :–

→ various tools which essential for compiling, running, documenting, packaging and debugging Java application, such as the Java Application launcher (java), Java compiler (javac), Documentation generator (javadoc) etc. and

2025.10.31 16:14

- a runtime environment (Java Runtimes Environment or JRE).

JDK = 1-Development tools + 2-JRE

- JRM :-

provides the libraries, Java virtual machine (JVM) and other components to run Java application.

- The JRE is part of the JDK, and it includes the JVM along with the necessary runtime libraries.
- used by developers to create and compile Java application.
- The Java Runtime Environment (JRE) is a software platform that enables the execution of Java applications.
- The JRE allow Java programs to run on any device or operating system without modification by translating Java bytecode into native machine code that the JVM executes.
- These are 4 components of JRE :-

1) Java libraries :-

Java libraries are collections of pre-defined written code that provide essential functionality for development. They includes base

libraries (Java.lang, Java.util).

2) User interface toolkits :-

User interface toolkits in JRE, like AWT, Swing and Java2D provides tool for building GUI.

3) Deployment :- Deployment in JRE involves packaging and distributing Java application for execution on client machine.

4) Java Virtual Machine :-

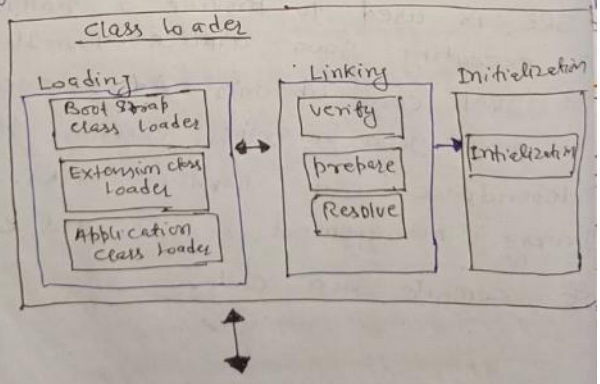
The Java virtual machine (JVM) is an engine that executes Java byte code. It provides platform independence by allowing Java programs to run on any device with a JVM, handling tasks like memory management, garbage collection and code optimization.

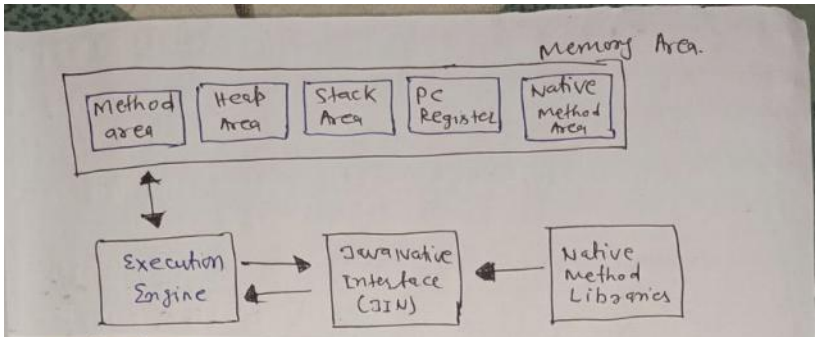
→ JRE = Java libraries + User Interface toolkit + Deployment + JVM

- JRE is used to provide a runtime platform for executing Java applications. It includes the JVM, core libraries and resources required to run Java programs, ensuring platform independence and handling tasks like memory mangement, security and execution of compile Java code.

What is JVM :-

- The JVM is a part of the Java platform that executes Java bytecode, converting it into machine code for the host system. It provides an abstraction layer blw Compiled Java code and the operating system.
- It means JVM creates a platform to run Java bytecode (.class file) and converting into different language (native machine language) which the computer hardware can understand.
- The main use of the JVM is to enables platform-independent execution of Java programs by interpreting or compiling bytecode to machine code, managing memory, ensuring security and optimizing performance.





class loader :-

- class Loader is responsible for loading class into the memory area. The classloader is an abstract class generates the data which consitiutes a definition for the class using a binary name which is constituted of the package name, and a class name.

→ **Loading :-** Whenever JVM loads file, it will load and read,

- fully qualified class name.
- variable information (instance variables).
- Immediate parent information
- whether class or interface or enum.

When the class is loaded to the JVM, it creates an object of class type and is put into the heap area. This class type object will be created only the very first time the class is loaded to the JVM.

Linking :- This is the process of linking the data in the class file into the memory area. It begins with verification to ensure this class file and the compiler.

1) Make sure this compiler is vaild.
2) The class file has the correct formating.
3) The class file has the correct structure.

Initialization :-

This is the final stage of class loading. In this stage, the actual values are assigned to all static and instance variables. These is a rule that every class must be initialized before doing any active use

→ These 4 ways to initialize a class in Java.

1) New Keyword :- normal Initialization process.

2) Reflection API :- getInstance() method

3) clone method :- Gets the information from the source object.

4) IO. ObjectInputStream :- Gets data from non-transient variable passed in the parameter.

Types of class Loader :-

- Bootstrap Class Loader :- Load classes from JRE/lib/rt.j
- Extension ClassLoader :- Load classes from JRE/lib/ext
- System/Application classLoader :- Load classes from class path, -cp, Manifest.

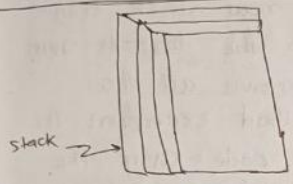
⇒ Method Area :-

- Method area is a logical part of the heap and it is created when the JVM starts.
- Method area is used to store class-level information such as class structures, method, bytecode, static variables, constant pool, interfaces
- Method area can be fixed or dynamic size depending on the system's configuration.
- Static variables in Java are stored in the method area.
- Grabage collection of the method area is not guaranteed and depends on JVM implementation.

- Method area is logically a part of heap, many JVM like HotSpot uses a separate space known as metaspace outside the heap to store it.

Note
- Static variables are stored in the Method area.
- Instances variable are stored in the Heap.
- Local variable are stored stack.

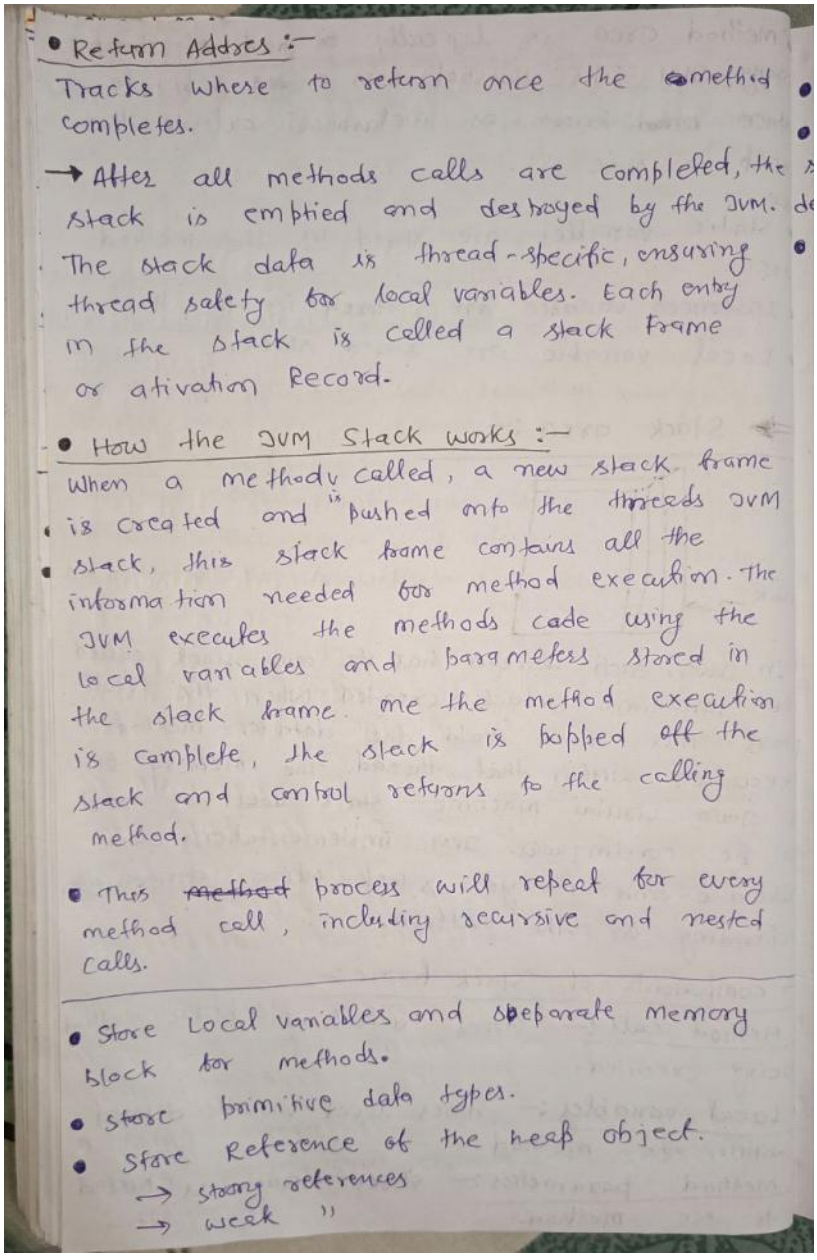
⟹ Stack area :-



- In Java, each thread has its own stack called the run-time stack, created when the fread start. This stack hold the data for method execution within that thread. The memory for a Java visutal machine stack does not need to be continguous. JVM implementations can allocate and manage stack memory dynamically depending on the paltform.

→ components of stack frame :-

- Method call :- Stores details about the method being executed.
- Local variables :- Stores local variables defined within the method.
- Method parameters :- Stores parameters passed to the Method.



• Return Address :-

Tracks where to return once the method completes.

→ After all methods calls are completed, the stack is emptied and destroyed by the JVM. The stack data is thread-specific, ensuring thread safety for local variables. Each entry in the stack is called a stack frame or ativation Record.

• How the JVM Stack works :-

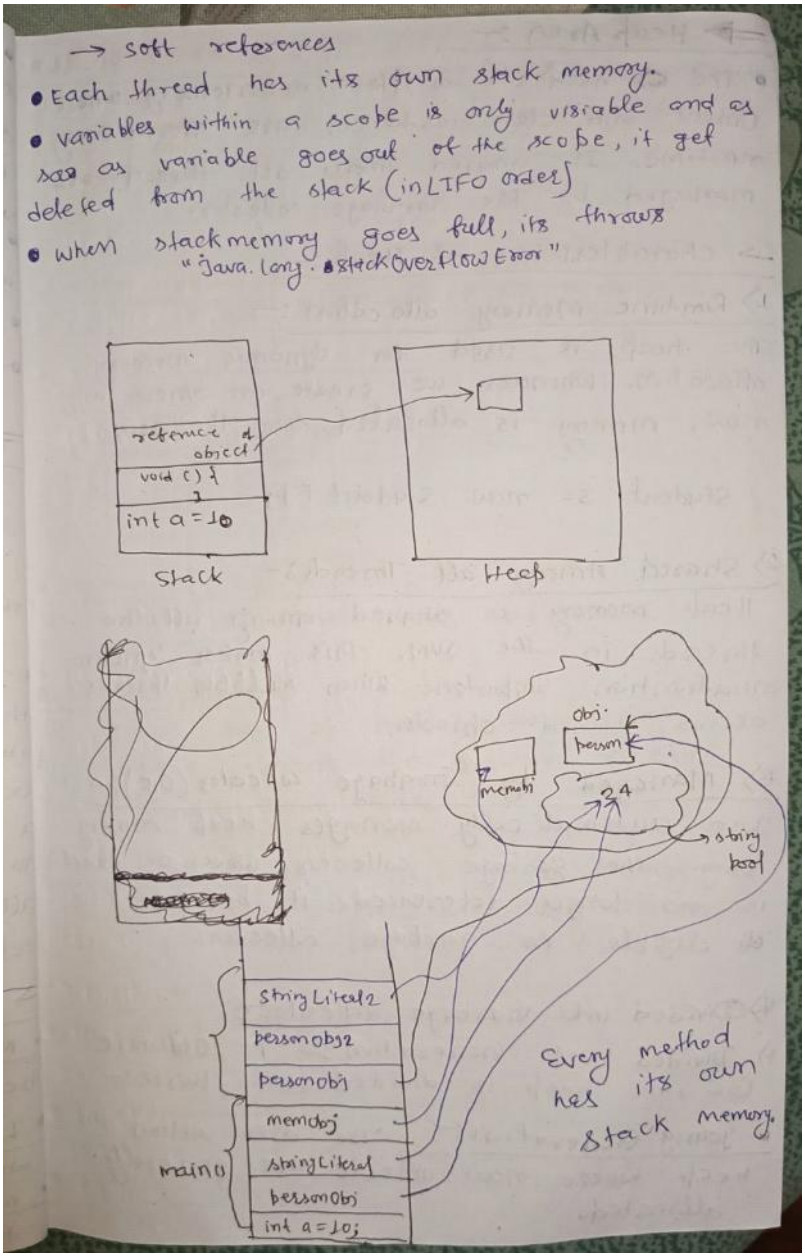
When a methody called, a new stack frame is created and is pushed onto the threads JVM stack, this stack frame contains all the information needed for method execution. The JVM executes the methods code using the local variables and parameters stored in the stack frame. one the method execution is complete, the stack is popped off the stack and control returns to the calling method.

• This ~~method~~ process will repeat for every method call, including recursive and nested calls.

• Store Local variables and separate memory block for methods.
• Store primitive data types.
• Store Reference of the heap object.
  → strong references
  → week "

→ soft references

- Each thread has its own stack memory.
- variables within a scope is only visiable and as soon as variable goes out of the scope, it get deleted from the stack (in LIFO order)
- when stack memory goes full, its throws "java.lang.StackOverFlowError"



⟹ Heap Area :-

- The heap is a place in memory where Object and class instances are stored during runtime. It shared among all threads and managed by the Grabage collector.

→ Characteristics of Heap :-

1) Runtime Memory allocation :-

The heap is used for dynamic memory allocation. Whenever we create an object using new, memory is allocated from the heap.

```
Student s= new Student ();
```

- **2) Shared Among all Threads :-**

Heap memory is shared among all the threads in the JVM. This makes synchronization important when multiple threads access shared objects.

3) Managed by Grabage collector (GC) :-

Java automatically manages heap memory using the garbage collector. When an object is no longer referenced, it becomes eligible for garbage collection.

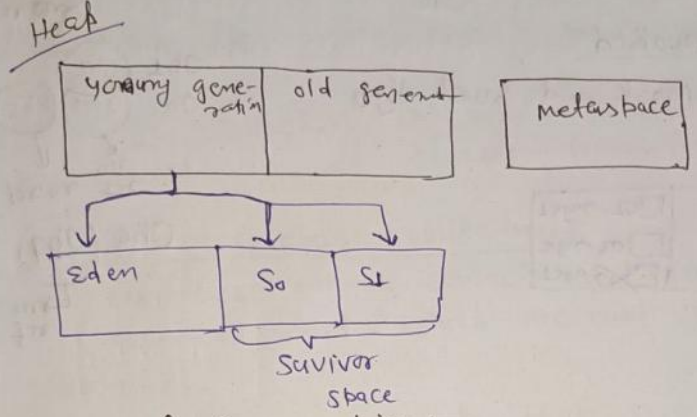
~~4) Divided into Grabage collector~~

4) Divided into Generation :- To optimize GC, the heap is divided into parts :-

- **young Generation :-** The area within the heap where new objects are generally allocated.

• Old generation :-

The area within the heap where long lived objects are stored after surviving multiple garbage ~~Collector~~ cycle.
Collection.



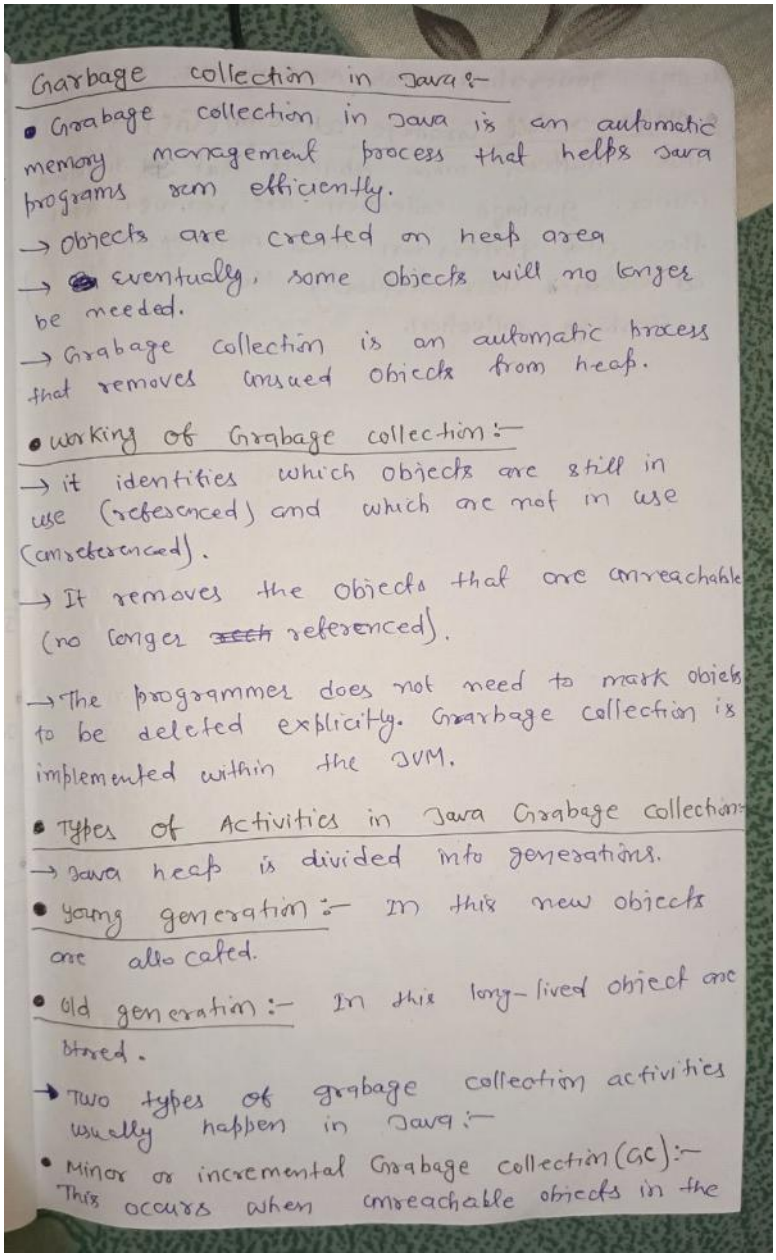
• permanent Generation :-

It stores Meta data about classes and method.

• survior space :-

If is a part of heap memory young generation where objects the survive garbage collection in Eden space and moved.

• code cache :- It is the special memory outside the heap where JVM stores optimized code to make programs run faster.



Garbage collection in Java :-

- Grabage collection in Java is an automatic memory management process that helps Java programs run efficiently.

→ Objects are created on heap area

→ eventually, some objects will no longer be needed.

→ Grabage collection is an automatic process that removes unused objects from heap.

- working of Grabage collection :-

→ it identifies which objects are still in use (referenced) and which are not in use (unreferenced).

→ It removes the objects that are unreachable (no longer referenced).

→ The programmer does not need to mark objects to be deleted explicitly. Grarbage collection is implemented within the JVM.

- Types of Activities in Java Grabage collection :-

→ Java heap is divided into generations.

- young generation :- In this new objects are allocated.
- Old generation :- In this long-lived object are stored.

→ Two types of grabage collection activities usually happen in Java :-

- Minor or incremental Grabage collection (GC) :- This occurs when unreachable objects in the

Writing, Compiling and Executing :-

- **Step 1:-**

→ Writing the Java program.

→ The program is saved with a .Java extension or MainApp.java.

→ The .Java file contains our Java source code, including a class with a main method (the entry point of the program).

- **Step 2:-**

In compilation phase, we open CMD, navigate to the director where our .Java file is located and run the javac command.

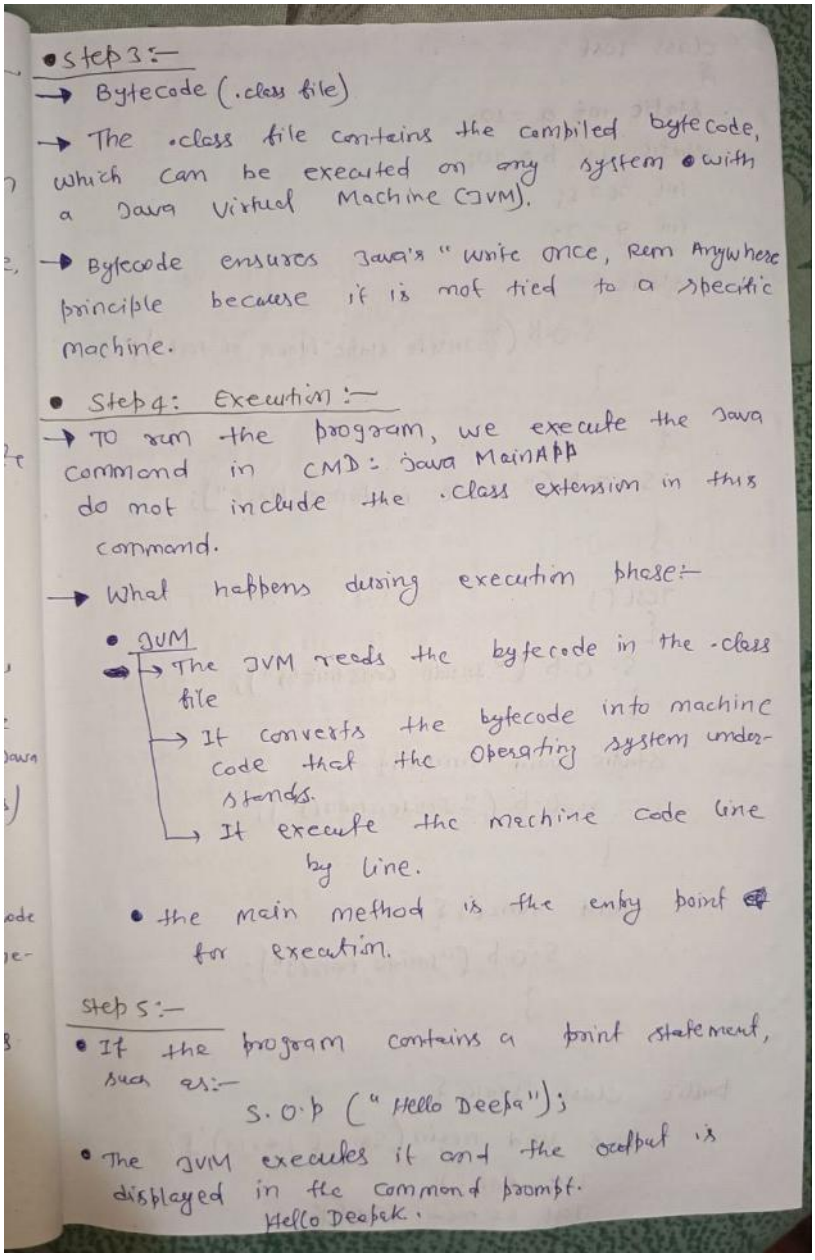
javac MainApp.java

→ ~~Javac is activated the compiler,~~ ~~decide~~

→ The javac command activates the Java compiler to convert our .Java source file into bytecode (.class) files.

→ If there are no errors, it compiles the code into bytecode a platform-independent intermediate representation.

→ The bytecode is saved as .class file (e.g MainApp.class)



• Step 3:—

→ Bytecode (.class file)

→ The .class file contains the compiled bytecode, which can be executed on any system with a Java Virtual Machine (JVM).

→ Bytecode ensures Java's "write once, Run Anywhere" principle because it is not tied to a specific machine.

• Step 4: Execution :—

→ To run the program, we execute the Java command in CMD : java MainApp
do not include the .class extension in this command.

→ What happens during execution phase:—

• JVM
  → The JVM reads the bytecode in the .class file
  → It converts the bytecode into machine code that the Operating system understands.
  → It execute the machine code line by line.

• the main method is the entry point for execution.

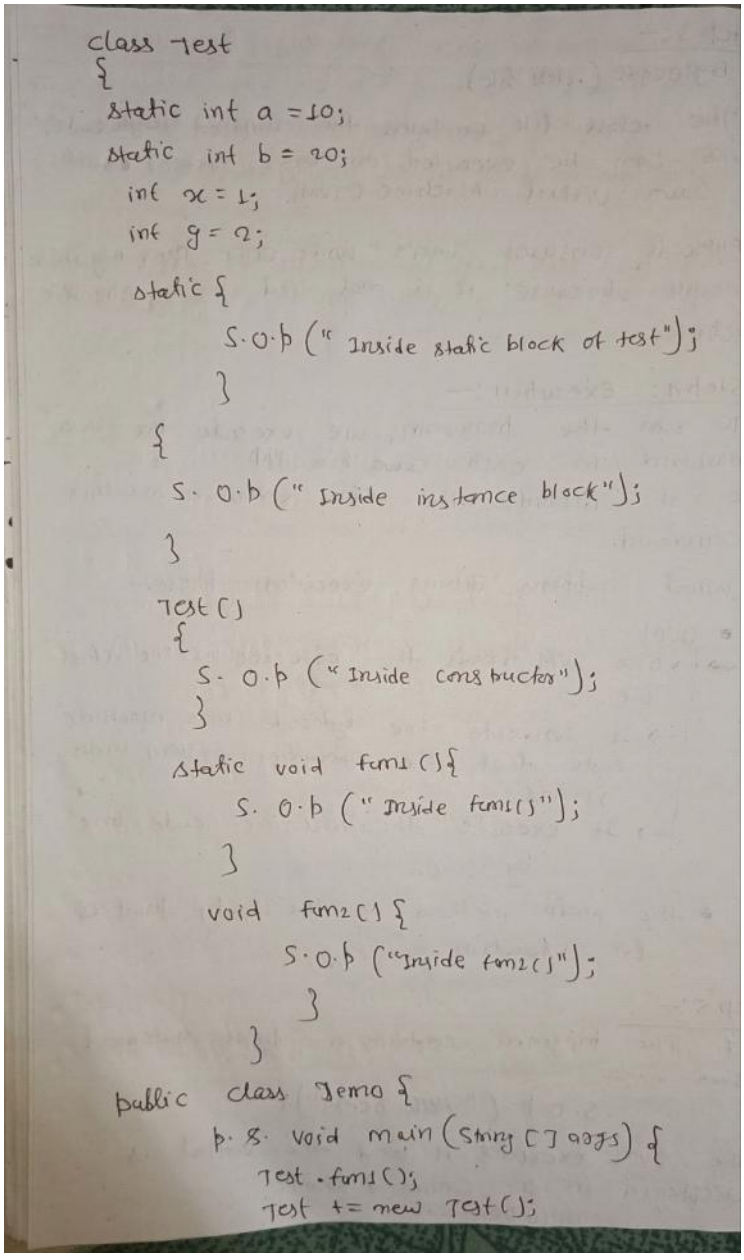
Step 5:—

• If the program contains a print statement, such as:—

```
S.O.P ("Hello Deepa");
```

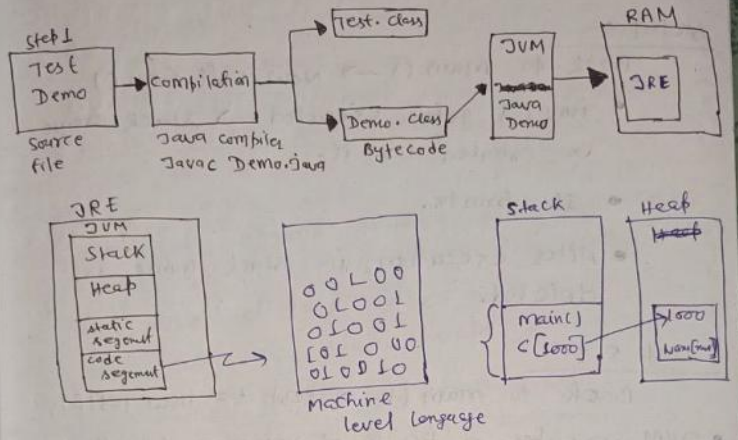
• The JVM executes it and the output is displayed in the command prompt.

Hello Deepak.



```
class Test
{
 static int a =10;
 static int b = 20;
  int x = 1;
  int y = 2;

  static {
        S.O.P (" Inside static block of test");
       }

   {
    S. O.P (" Inside instance block");
   }

   Test ()
    {
     S. O.P (" Inside constructor");
     }

    static void fun1 (){
        S. O.P (" Inside fun1()");
       }

      void fun2 () {
           S.O.P ("Inside fun2()");
            }
       }
public class Demo {
     P. S. void main (String [] args) {
        Test . fun1 ();
        Test t= new Test ();
```

t.fun2();
}
}



→ Step 1 :-
JVM load main() method.
→ The class Demo loaded first
→ JVM checks for static variables and static blocks in Demo. There are none.
→ main() gets called → stack frame is created for main().

→ Step 2 :-
→ First line in main() → Test.fun1();
- JVM see you're calling fun1() of Test class.
- But Test class isn't loaded yet, so JVM uses class loader to load it.

→ Step 3 :-
When Test is loaded :-

JVM looks for :-

- static variables → find a and b → memory allocated in static segment.
- static block → It runs and prints.

Step 4 :-

Back to main() → Now call fun1()

- fun1() gets executed → stack frame is created for it.
- It prints.
- After execution, its stack frame is deleted.

Step 5. :-

Back to main() → Test t = new Test();

- JVM creates a block of memory for the new object(t) → allocates space for x and y.
- Then JVM looks for instance block → runs it.

After that, Constructor called :-

Step 6 :-

t. fun2();

- JVM call fun2() → creates new stack frame
- prints
- Then deletes fun2()'s stack frame.

Heap Area :-

- The heap is a place in memory where Object and class instances are stored during runtime. It shared among all threads and managed by the Grabage collector.

→ Characteristics of Heap :-

1) Runtime Memory allocation :-

The heap is used for dynamic memory allocation. Whenever we create an object using new, memory is allocated from the heap.

```
Student s= new Student ();
```

- **2) Shared Among all Threads :-**

Heap memory is shared among all the threads in the JVM. This makes synchronization important when multiple threads access shared objects.

3) Managed by Grabage collector (GC) :-

Java automatically manages heap memory using the garbage collector. When an object is no longer referenced, it becomes eligible for garbage collection.

~~4) Divided into Grabage collector~~

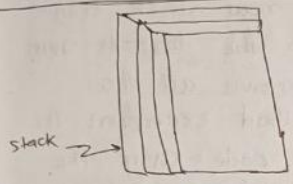
4) Divided into Generation :- To optimize GC, the heap is divided into parts :-

- **young Generation :-** The area within the heap where new objects are generally allocated.

- Method area is logically a part of heap, many JVM like HotSpot uses a separate space known as metaspace outside the heap to store it.

Note
- Static variables are stored in the Method area.
- Instances variable are stored in the Heap.
- Local variable are stored stack.

⟹ Stack area :-



- In Java, each thread has its own stack called the run-time stack, created when the fread start. This stack hold the data for method execution within that thread. The memory for a Java visutal machine stack does not need to be continguous. JVM implementations can allocate and manage stack memory dynamically depending on the paltform.

→ components of stack frame :-

- Method call :- Stores details about the method being executed.
- Local variables :- Stores local variables defined within the method.
- Method parameters :- Stores parameters passed to the Method.