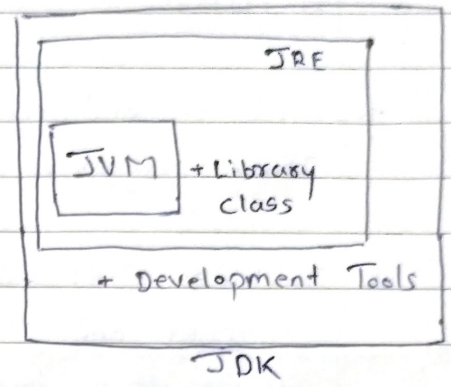
Assignment No: 3

→Q1 The Java Development Kit (JDK) is a software development environment that contains many components.

1. Java Compiler (javac)

Translates Java source code (written in .java files) into bytecode (save in .class files) which can be executed by Java Virtual Machine (JVM)



2. Java Runtime Environment (JRE)

Executes the Java bytecode. It acts a a virtual processor, converting bytecode into machine code for execution on the host system.

- The JRE provides the libraries and other components necessary to run Java application

3. Java Virtual Machine (JVM)

The JVM is responsible for executing the bytecode produced by the Java compiler. It provides platform independence by converting bytecode into platform specific machine code at runtime.

4. Java Documentation Tool (javadoc):

Generates HTML - formated documentation from Java source code comments.

5. Java class Libraries.

A collection of pre-built classes and packages that provide basic functionality needed to build Java applications.

Q2 Difference between JDK, JVM and JRE.

	JDK	JVM	JRE
1.	Develop and compile Java programs.	Execute Java bytecode	Run Java Application
2.	Includes JRE, JVM, compiler and development tools.	Bytecode interpreter class, loader, garbage collector, execution engin.	Includes JVM and core libraries.
3.	For developers to write, compile and debug Java program	For anyone running Java bytecode	For running Java application only
4.	JRE, compiler and debugging tool	Execution environment for bytecode	JVM, core libraries
5.	Java developers.	Both developers and end users	End user running Java programs

→ Q3 Role of the JVM in Java

The Java Virtual Machine (JVM) plays a central role in the Java platform, as it allows Java programs to be executed independently of the underlying operating system and hardware

1. Platform independence : Write Once, Run Anywhere
2. Bytecode Execution : Executing Java bytecode.
3. Memory Management

4: Security : Security model that ensures java program
5: Exception Handling. : Exception (runtime error)
6: Class Loading and verification
: loading class file and verifying their correctness

How the JVM Executes Java Code

1. Compilation to Bytecode :
First the java compiler (javac) compiles the java source code (.java files) into bytecode (.class files)

2. Class Loading :
The JVM class loader subsystem loads the complied .class file (bytecode) into memory when the application starts or when a class is first used. This dynamic class loading enables on-demand loading of classes.

3. Bytecode verification
The JVM verifies that ~~code~~ bytecode ensure it is well formed and adheres to Java security constraints.

4. Execution
The JVM the executes the bytecode through its Execution Engine.
- Interpretation
- Just-in-Time

5. Memory Management (Heap & Stack)
- Heap Memory : Used for dynamically created objects with new keyword
- Stack Memory : Used the storing method calls, local variable, and refereces

6. Garbage Collection

The JVM continuosly monitors object in memory and automatically cleans up unused objects through Garbage collector.

7 Execution of Machin Code

After the bytecode is either interpreted or compiled into native code by the JIT compiler, the processor executes the corresponding machine instructions

8. Handling Exceptions.

04 Memory management system of the JVM

JVM Memory Structure

a] Heap Memory

The heap is where all objects are dynamically allocated at runtime using the new keyword.

b] Stack Memory

The Stack memory is used for storing method calls local variable and method execution.
Each thread in the JVM has its own stack.
When a method is invoked, a new block (frame) is created in the stack.
It follows LIFO (Last in first out) principle

Non-Heap Memory

a] Method Area

The method area stores class-level metadata such as

- Class structure
- Method data
- Static variable
- Runtime constants.
- Class Loading
- Meta space

5] Program Counter (PC) Register

The PC register stores the address of the current instruction being executed by each thread.
It helps the JVM keep track of the program's control flow

c] Native Method Stack.

The Native Method Stack stores information for native (non-Java) methods invoked using Java Native Interface (JNI). It is used when Java programs call libraries written in other languages like C/C++.

Garbage Collection JVM

Java automatically handles memory deallocation through Garbage Collection (GC)

4. Memory Leaks and OutofMemory Error

Despite automatic memory management, memory leaks can still occur if objects are unintentionally held by references, even though they are no longer needed. This can lead to a OutofMemory Error when the heap memory is exhausted and the JVM cannot allocate more space.

Q5 What are the JIT compiler and its role in the JVM? What is the bytecode and why is it important for Java.

→ Just-in-Time (JIT) Compiler

The JIT compiler is an essential part of the Java Virtual Machine (JVM) that improves the performance of Java applications by converting bytecode (intermediate representation of Java code) into native machine code at runtime. This allows Java programs to execute faster by eliminating the overhead of interpreting bytcode repeatedly.

Role of the ~~JVM~~ JIT Compiler

Compitation at Runtime

Optimization

Execution

Bytecode

Bytecode is an intermediate representation of Java source code, generated by the Java compiler (javac) It is a low-level platform-independent code that the JVM can interpret into native machine code.

Q6 Describe the architecture of the JVM

→ 1. Class Loader System

The Class Loader Subsystem is responsible for loading .class files (Java bytecode) into the JVM for execution. Java programs are complied into bytecode by the Java compiler (javc) and stored in .class files.

2. Runtime Data Area

a) Method Area.

Stores class structure (metadata), static variables and method code.

- Hold information about loaded classes, including methods and variables. shared among all threads

b) Heap

The heap is used to store object instances and arrays.

- All new objects are allocated memory in the heap and when objects are no longer is use garbage collection

c) Stack

Each thread has its own stack memory which stores local variables, method arguments and frames for each method call.

d) PC (Program Counter) Register

Each thread ~~in~~ the JVM has its own PC register It holds the address of the current instruction being executed by the thread.

Once the instruction is executed the PC register is updated to point to the next instruction.

e) Native Method Stack

This stack is used for managing native method call (methods written in language like C and C++)

3. Execution Engine

~~The~~ ~~for~~ a) Interpreter

The interpreter executes bytecode instruction one by one

While it provide a quick startup it can be slower since the same instruction are repeatedly interpreted.

b) Just-in-Time (JIT) Compiler

The JIT compiler frequently executed bytecode (knows as "hot spots") into native machine code at runtime.

c) Garbage Collector

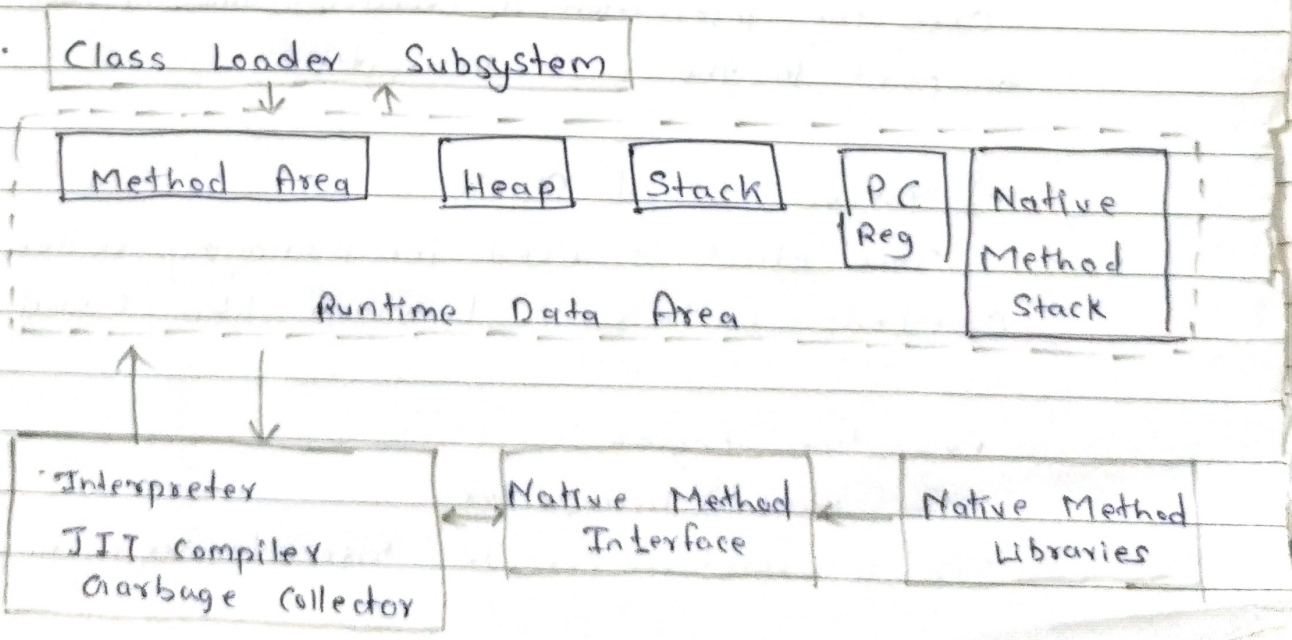
The Garbage Collector (GC) is responsible for automatically manging memory by identifying and deallocating memory that is no longer in use by objects.

d) Native Method interface

The JNI enables the JVM to call function in other programming languages like C and C++.

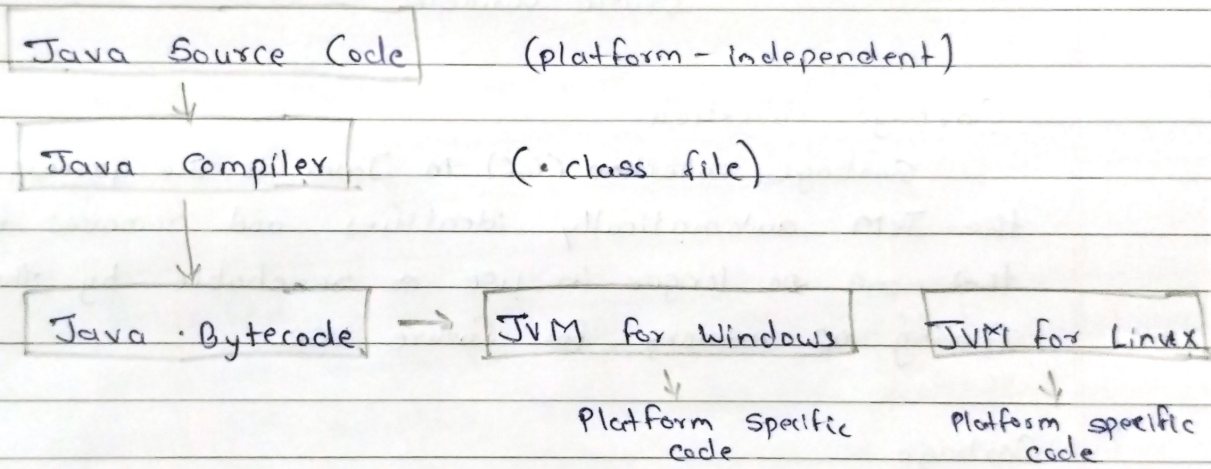
4. Native Method Libraries.

Native libraries are external libraries that are specific to the platform on which the JVM is running (e.g Windows, Linux)



Q7 How does Java achieve platform independence through the JVM

→ Java achieves platform independence by compiling source code into platform neutral bytecode.
This bytecode is then executed by the JVM, which abstracts away the details of the underlying operating system and hardware.
As long as there is a JVM available for a platform the same bytecode can be executed, enabling Java program to "Write Once, Run Anywhere".



Q8 What is the significance of the class loader in Java What is the process of garbage collection in Java.

→ Significance of the Class loader in Java
The Class Loader in Java is a critical component of the Java Virtual Machine (JVM) that dynamically loads Java classes into memory.
The JVM doesn't know about the classes of a Java program until they are loaded at runtime by the class loader.

Class Loading Process

1. Loading : The class loader locates the class file and loads it into memory.

2. Linking :
   - Verification : Ensures the bytecode is valid.
   - Preparation : Allocate memory for class variables and initializ
   - Resolution : Converts symbolic reference into direct references.

3. Initialization : Executes static blocks and initializes static variable.

Garbage Collection

Garbage collection (GC) in Java is the process by which the JVM automatically identifies and removes objects that are no longer in use or reachable by the application freeing up memory for future use.

Garbage

1. Marking : The garbage collector first marks all object reachable
2. Sweeping : Unmarked objects which no longer reachable and remov
3. Compacting : After sweeping the memory is compacted (defragmented) to make it contiguous, which allows efficient allocation of new objects.

→ Four acess modifiers

1. Private : Only accessible within the same class where it is defined.

- The member cannot be accessed from outside the class.

```
Class Data {
    private int data;
}
```

2. Default (Package - Private)
- Accessible within the same package (i.e. any class in the same package can access it) but not from classes in different packages.

```
Class Data {
    int data;
}
```

3. Protected

Accessible within the same package and by subclasses even if the subclass is in different package.

Used when a class member should be accesible to its subclasses, but not the general public.

```
Class Data {
    protected int data;
}
```

4. Public

Accessible from anywher There are no restriction on visibility

Any class in any package can access the member

```
public class Data {
    public int data;
}
```

Q10 Difference between public, protected and default access mod

→

Visibility	Default	Public	protected	Private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Non-subclass outside the same package	No	Yes	No	No
Subclass outside the same packag	No	Yes	Yes	No

→ Q11 Can you overribe a method with a different acess modifier in subclass? For example can a protected method in a superclass by overridden with a private method in a subclass? Explain.

→ No you cannot override a method with a more restrictive access modifier in a subclass.

~~You Method can~~ Override a method with the same or less restrictive access modifier but not a more restrictive one

~~Ma~~ For example, if a method in a superclass is protected, it can be overridden with protected or public in the subclass but not with private, as private is more restrictive than protected.

2. The overriding method must be accessible wherever the original (superclass) method is accessible Overriding with a more restrictive access modifier would break this rule and make the subclass method less accessible.

Example

```
Class SuperClass {
    protected void show() {
        System.out.println("SuperClass -show method")
    }
}
Class SubClass extends SuperClass {
    // This will cause a compile-time error
    private void show() {
        System.out.println("SubClass show method");
    }
}
```

Example of Correct

```
Class SuperClass {
    protected void show() {
        System.out.println("SuperClass show method")
    }
}
class SubClass extends SuperClass {
    @Override
    public void show() {
        System.out.println("SubClass show method")
    }
}
```

Q12 Difference between protected and default (package - private)

Access Modifier	Same Class	Same Package	Subclass in Same Package	Subclass in Different Package	Other packages
Protected	Yes	Yes	Yes	Yes (through inheritance)	No
Default	Yes	Yes	Yes	No	No

Q13 Is it possible to make a class private in Java? If yes where can it be done and what are the limitations?

→ Private classes are allowed, but only as inner or nested classes. We can have a private inner or nested class, then access is restricted to the scope of that outer class.
If we have a private class on its own as a top level then you can't get access to it from anywhere.

```
public class OuterClass {
    private class InnerClass {
        public void display() {
            System.out.println("This is a private")
        }
    }
}
public void useInnerClass() {
    InnerClass inner = new InnerClass();
    inner.display(); // Accessing private inner class
}
}
```

```
public class TestClass {
    public static void main ( String[] args) {
        Outer Class outer = new Outer Class ();
        outer. useInner Class (); // Not allowed
        }
    }
```

Q14 Can a top-level class in Java be declared as protected or private ? Why or why not?

→ No, a top-level class in java cannot be declared as protected or private. This is because top-level classes in some form to other parts of the program and the purpose of protected and private access modifiers is to restrict access more tightly than what a top-level class allows.

Q15 What happens if you declare a variable or method as privated in a class and try to access it from another class within the same package?

→ Declare a variable or method as private in a class and try to access it from another class within the same package, the code with result in a compile-time-error This is because the private access modifier restricts access to the variable or method to within the same class only meaning no other even those in the same package can access it.

```
class MyClass {
    private int privateVariable = 10; // private variable

    private void privateMethod() {     // private method
        System.out.println("This is private")
    }
}
public class AnotherClass {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Trying to access private
        System.out.println(obj.privateVariable); // Compile error
        obj.privateMethod();                     // Compile-time error
    }
}

class MyClass {
    int privateVariable = 10;  // now package private

    void privateMethod() {     // now package private
        System.out.println("This is package-private")
    }
}
public class Another {
    public static void = new MyClass();

    // Now accessible
    System.out.println(obj.privateVariable);  // work fine
    obj.privateMethod();                      // work fine
```

Q16 Explain the concept of package private or default acess How does it affect the visibility of class members?

→ The concept of package-private (also referred to as default" access) in Java refers to the access level of class members (fields, methods, constructors) when no explicit access modifier (such as public, protected or private) is specified.

Visibility within the Same Package:

Package private members are visible to all classes within the same package but hidden from any class outside the package.

Not Inherited Outside the Package:

Even if a class is inherited by a class located outside the package, package-private members are not accessible to the subclass.