OBJECT ORIENTED PROGRAMMING

OOP

Object Oriented Programming (OOP)
Assignment 1: Java Fundamentals

1. What is the difference between JDK and JRE?

JDK (Java Development Kit):

- The JDK is a full software development kit used to develop Java applications.
- It contains tools like the Java compiler (javac), debugger, and JRE (Java Runtime Environment).
- Developers use JDK to write, compile, and debug Java programs.

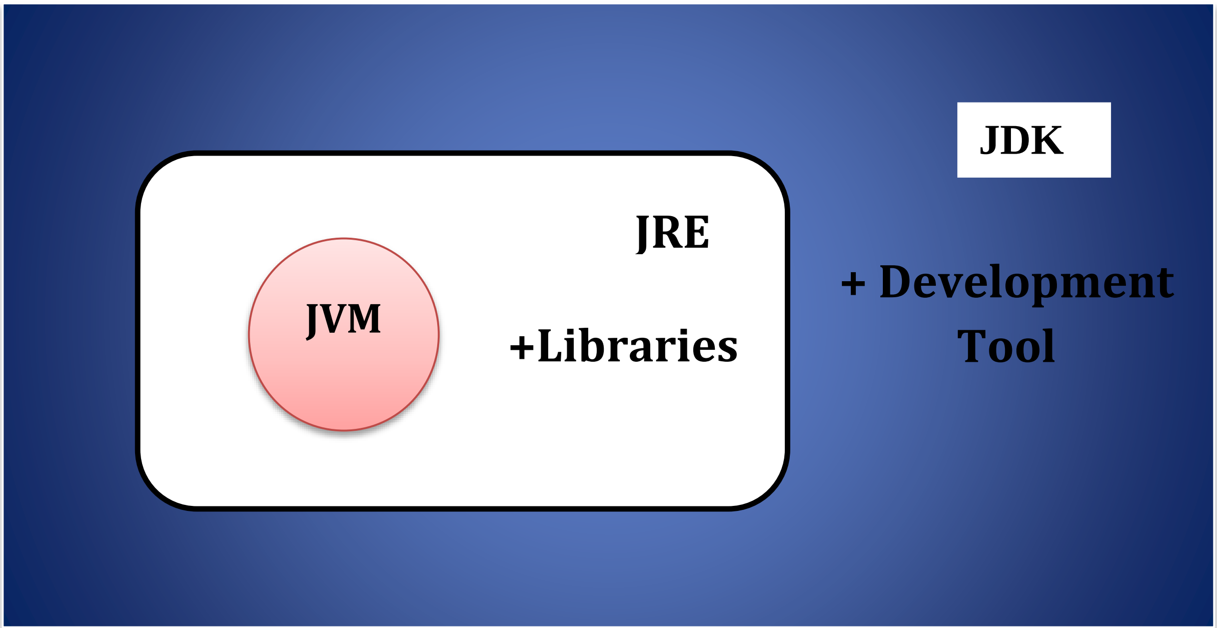
JRE (Java Runtime Environment):

- The JRE is a part of JDK and contains JVM (Java Virtual Machine), core libraries, and supporting files to run Java programs.

- However, it does not have development tools like compilers or debuggers — it's meant for end-users who only need to run Java programs.

Key Differences:
- JDK = JRE + Development tools (compiler, debugger, etc.)
- JRE = JVM + Libraries needed to run Java programs
- JDK is for developing and running Java programs, while JRE is just for running them.
Example: If you are writing and running Java programs, you need JDK. If you only want to run an existing Java program, JRE is enough.

Diagram



https://www.geeksforgeeks.org/differences-jdk-jre-jvm/

2. What is Java Virtual Machine (JVM)?

Java Virtual Machine (JVM) is an abstract machine that allows computers to run Java programs. JVM converts Java bytecode (compiled .class files) into machine code that the underlying operating system can execute. JVM provides platform independence by abstracting system-specific details.

Key Components of JVM:

- **ClassLoader:** Loads class files during runtime. It has three types:
  - **Bootstrap ClassLoader:** Loads core Java classes from java.lang package.
  - **Extension ClassLoader:** Loads classes from the ext directory.
  - **Application ClassLoader:** Loads classes from the classpath.
- **Memory Areas:**
  - **Method Area:** Stores class-level data such as method data, static variables, and runtime constant pool.
  - **Heap:** Used for dynamic memory allocation, storing objects and arrays.
  - **Stack:** Each thread has its own stack, which holds local variables, method call frames, and partial results.
  - **PC Register:** Contains the address of the current JVM instruction being executed.
  - **Native Method Stack:** Used for native methods written in other languages like C or C++.
- **Execution Engine:** Executes bytecode using either an interpreter or JIT compiler. It consists of:
  - **Interpreter:** Reads and executes bytecode line by line.
  - **JIT Compiler:** Translates bytecode into native machine code for better performance.
- **Garbage Collector:** Frees heap memory by removing unreferenced objects using algorithms like:
  - **Mark and Sweep:** Identifies and clears unused objects.
  - **Generational GC:** Divides memory into young, old, and permanent generations for optimized garbage collection.

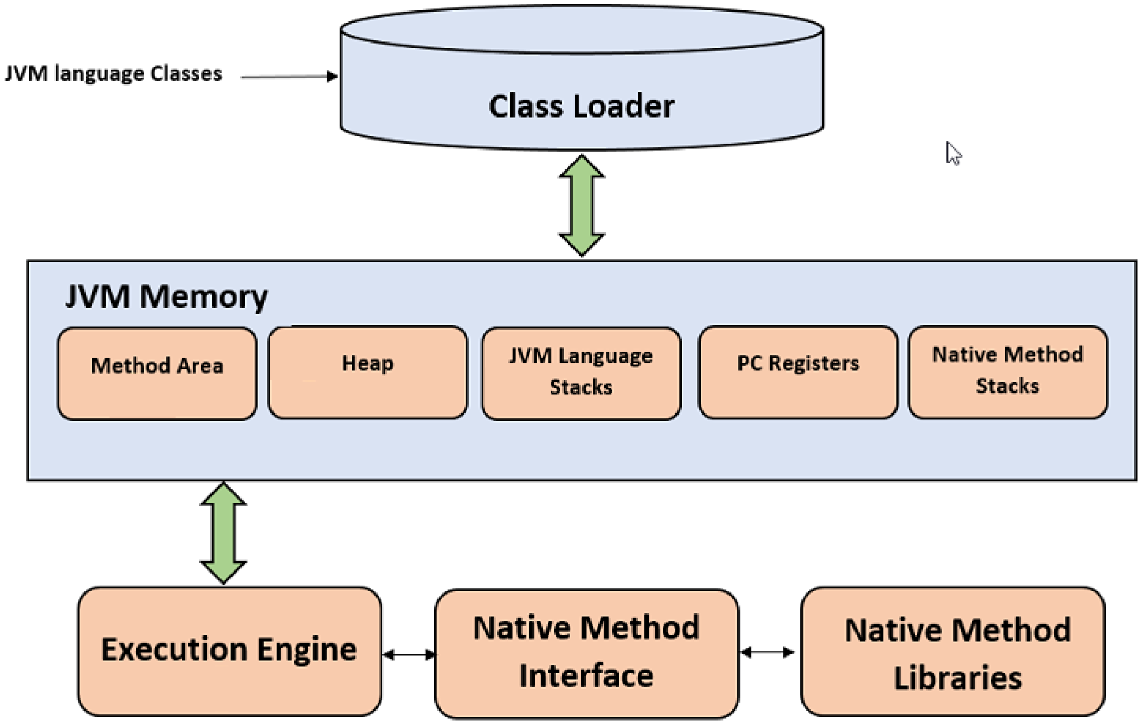
JVM Workflow:

1. Java source code (.java) is compiled by javac into bytecode (.class).

2. ClassLoader loads the bytecode into JVM.
3. Bytecode is verified for security.
4. Execution Engine interprets or compiles bytecode to native machine code.
5. Garbage Collector reclaims memory by removing unreferenced objects.

Example: When you compile a Java program, a .class file is generated. The JVM reads the bytecode and translates it into native machine code that runs on the host system.

Diagram:



Reference: https://www.javatpoint.com/jvm-java-virtual-machine

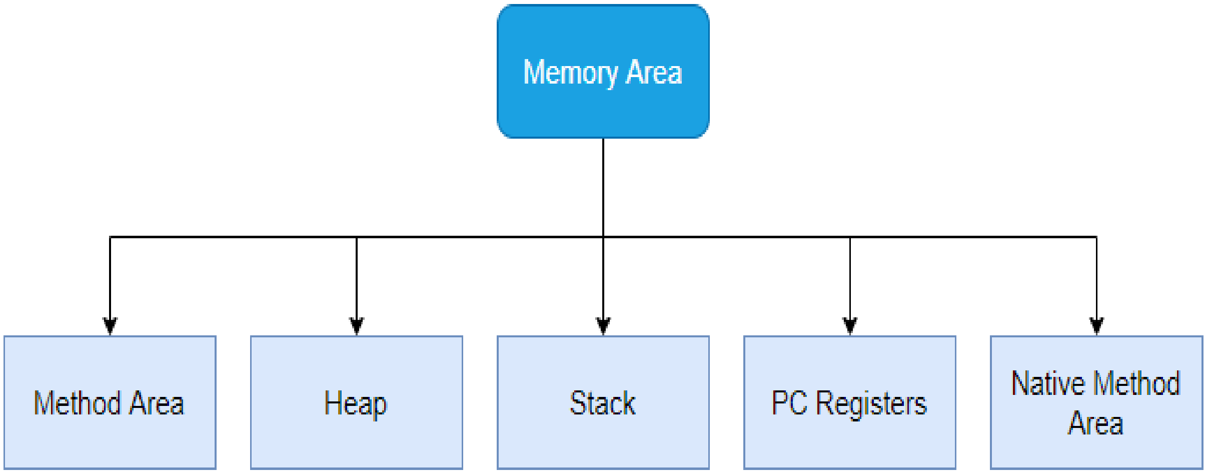
3. What are the different types of memory areas allocated by JVM?

JVM memory is divided into multiple areas to manage program execution efficiently:

- **Method Area:** Stores class-level data such as method data, static variables, and runtime constant pool.
- **Heap:** Used for dynamic memory allocation, storing objects and arrays.
- **Stack:** Each thread has its own stack, which holds local variables, method call frames, and partial results.
- **PC Register:** Contains the address of the current JVM instruction being executed.
- **Native Method Stack:** Used for native methods written in other languages like C or C++.

  Example: When an object is created (`Person p = new Person () ;`), the reference variable `p` is stored in the stack, while the actual `Person` object is allocated in the heap.

Diagram:



https://www.baeldung.com/java-jvm-memory-types

4. What is JIT compiler?

The Just-In-Time (JIT) compiler is part of JVM that boosts performance by compiling bytecode into native machine code at runtime. Instead of interpreting bytecode every time, JIT compiles frequently executed code (hotspots) into native code, speeding up execution.

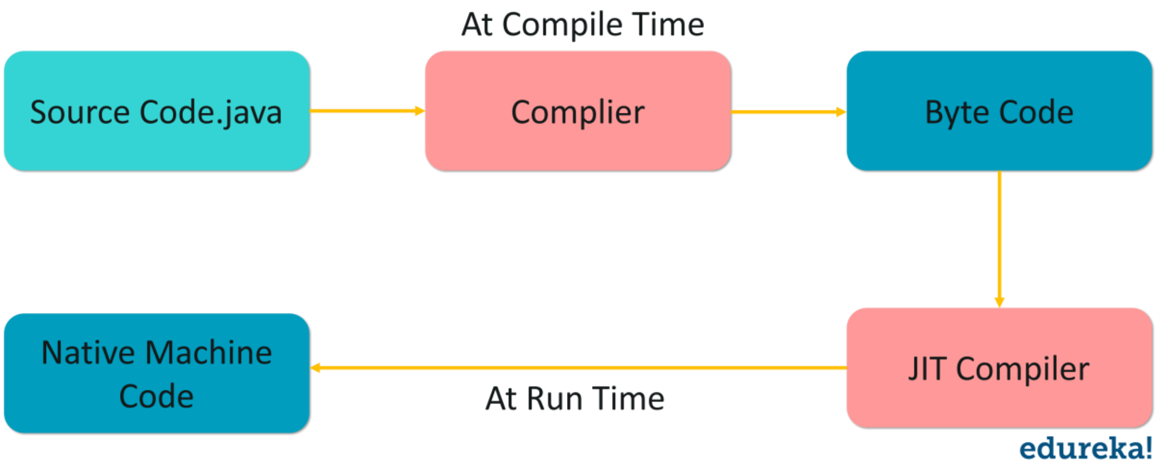
How JIT Works:

- JVM loads bytecode into memory.
- JIT identifies frequently used code.
- JIT compiles this bytecode into native machine code.
- The compiled code is executed directly, bypassing interpretation.

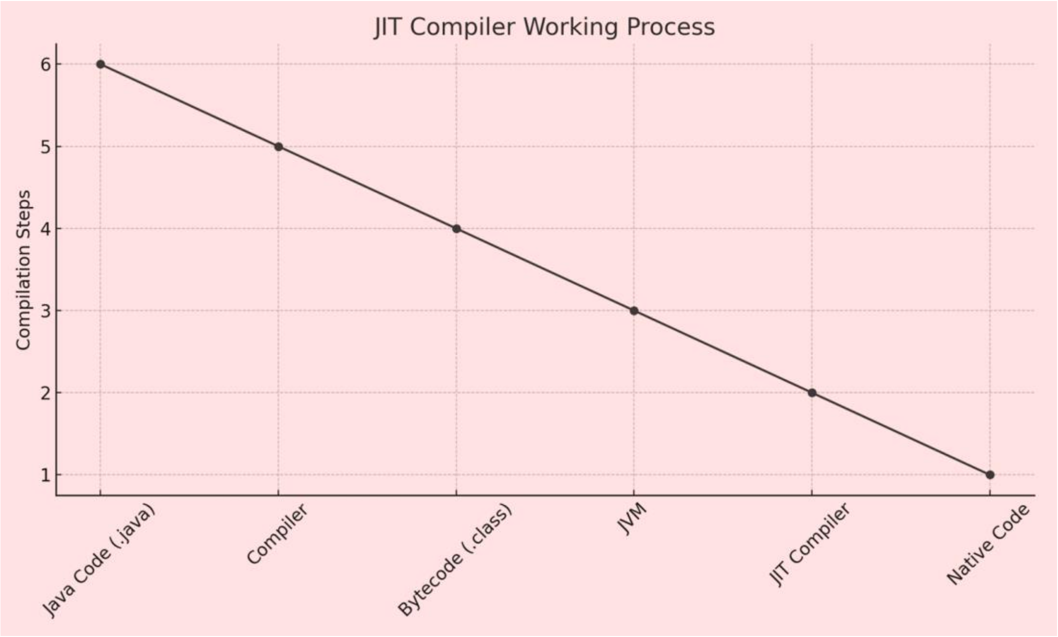
Benefits: Faster execution, reduced overhead, and optimized code for repeated tasks.

Example: If a loop runs 1000 times, JIT compiles the loop code once, so JVM doesn't re-interpret it every time.

Diagram:



https://www.baeldung.com/java-jvm-memory-types



https://www.freecodecamp.org/news/just-in-time-compilation-explained/

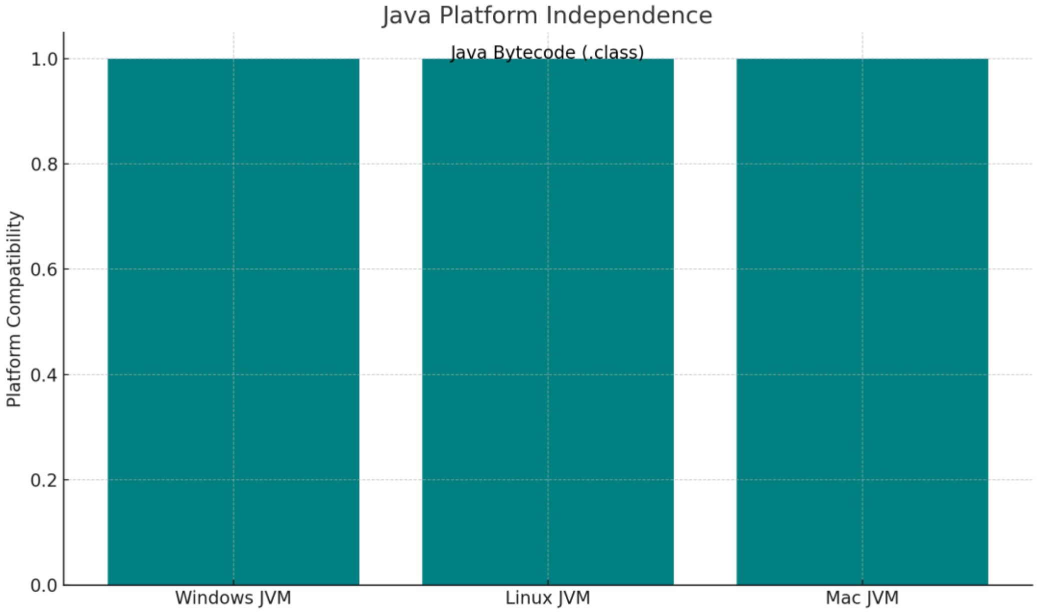
5. How is Java platform different from other platforms?

Java is platform-independent because it compiles source code into bytecode (.class files), which JVM executes. Unlike languages like C/C++ that compile directly to machine code (specific to OS), Java bytecode runs on any system with JVM.

Key Differences:
-**Java:** Compiles to byte code, then runs on any OS with JVM.
- C/C++: Compiles directly to machine code, making programs OS-specific.

Example: A Java program compiled on Windows will run on Mac or Linux without recompilation because JVM on each OS converts the same bytecode into platform-specific machine code.

Diagram:

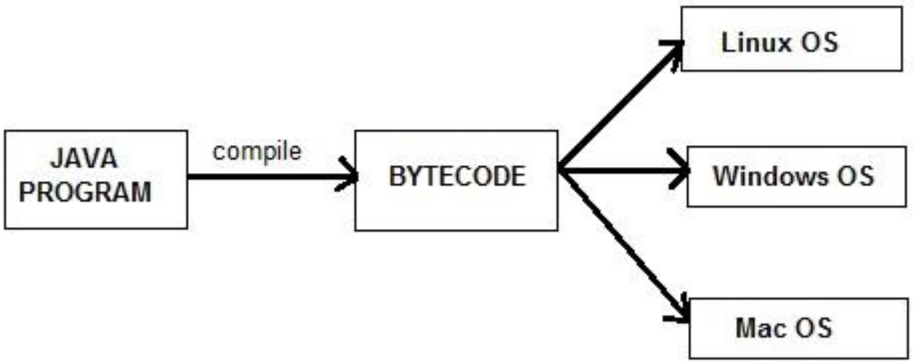


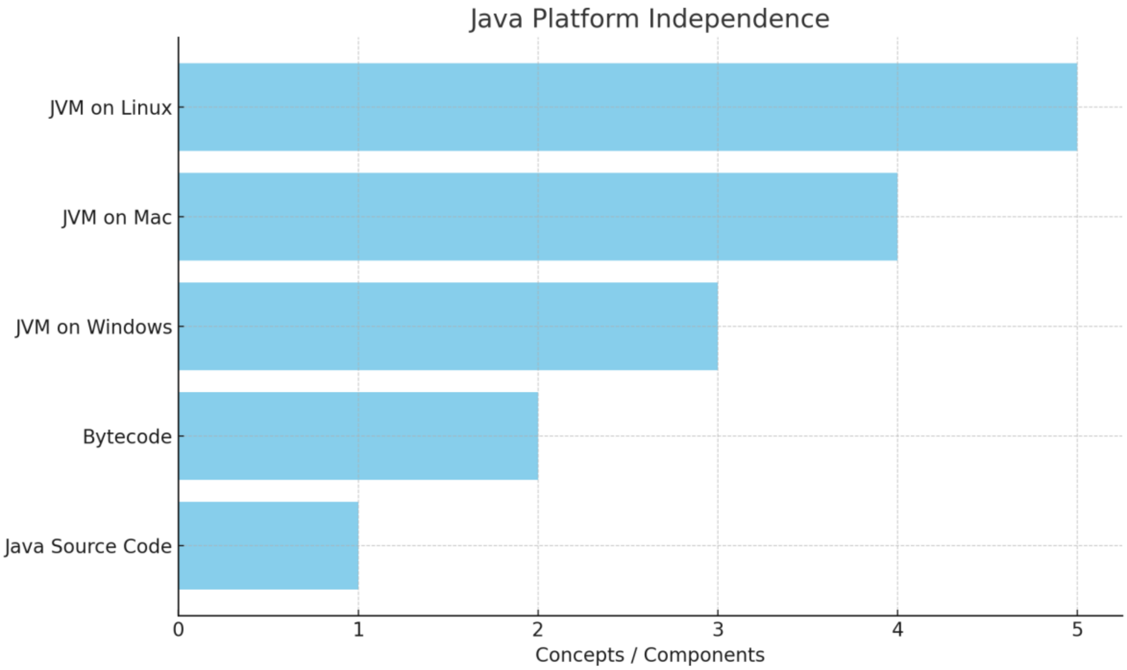
https://www.freecodecamp.org/news/just-in-time-compilation-explained/

6. Why Java called a "write once and run anywhere" language?

Java source code is compiled into byte code (.class files).
This byte code can be executed on any operating system that has JVM.
- As a result, Java programs do not need to be recompiled for different platforms.

Example: A Java program compiled on Windows can run on Mac or Linux without modification.





https://en.wikipedia.org/wiki/Write_once,_run_anywhere

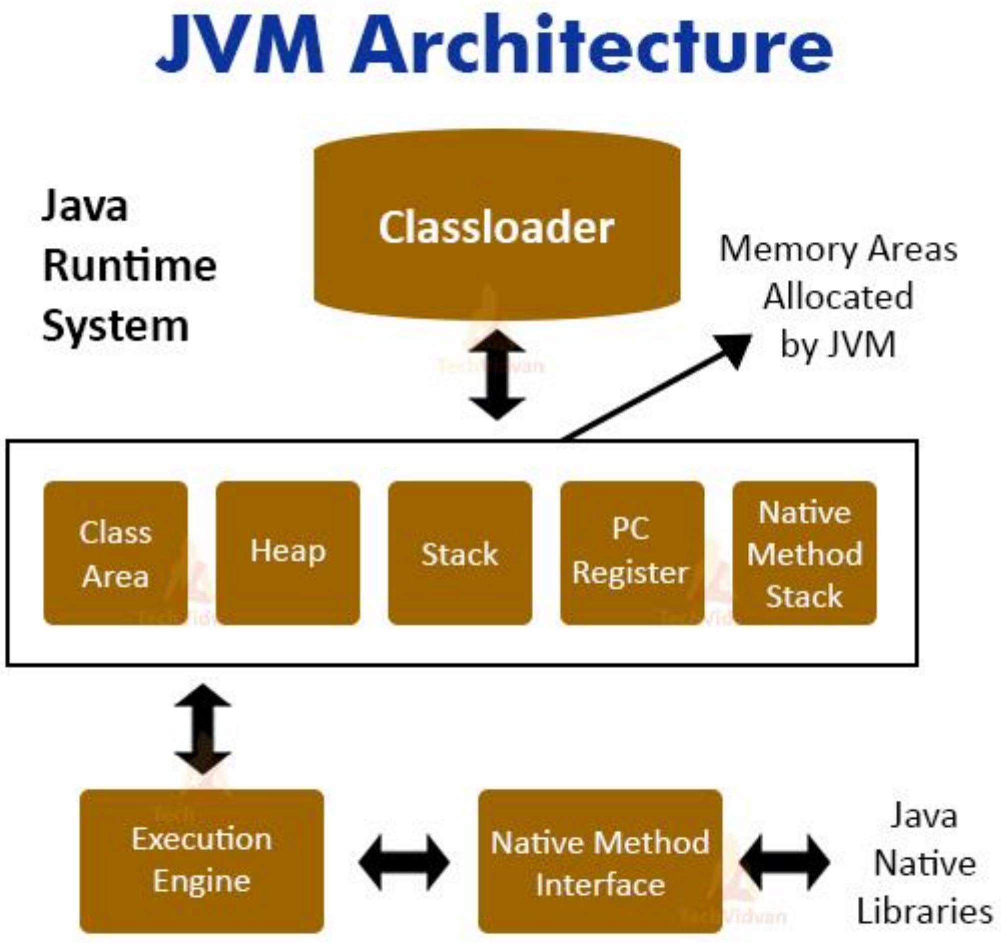
7. How does ClassLoader work in Java?

ClassLoader is a subsystem of JVM responsible for loading class files. It loads bytecode (.class files) into memory when a class is referenced for the first time during runtime.

Types of Class Loaders:

1. **Bootstrap ClassLoader:** Loads core Java classes from java.lang package and other essential classes.
2. **Extension ClassLoader**: Loads classes from the ext directory (java.ext.dirs).
3. **Application ClassLoader:** Loads classes from the classpath, specified by the CLASSPATH environment variable or -cp option.
4. **Custom ClassLoader**: Developers can extend ClassLoader class to create their own ClassLoaders to load classes in a customized way.

Diagram:



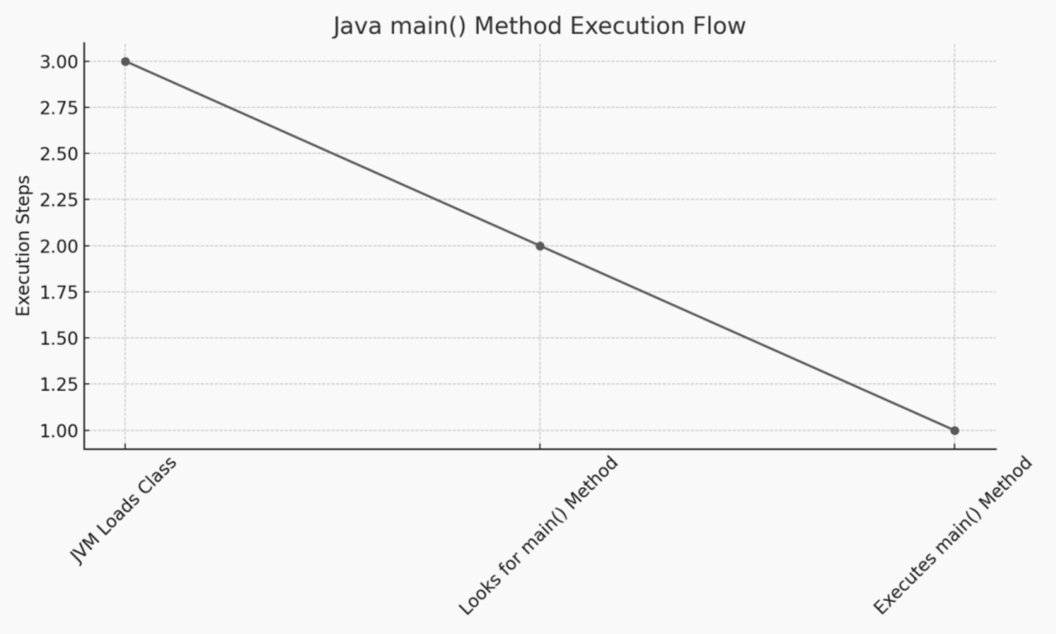
Digital Ocean (https://www.digitalocean.com/)

8. Is "main" used for main method a keyword in Java?

- No, "main" is not a Java keyword.
- It is just the name of the method that JVM looks for as the entry point of a Java program.
- The main ( ) method must have the correct signature to be recognized by JVM.

Correct signature: public static void main (String [ ] args)

The following diagram shows how JVM executes the main () method:



https://stackoverflow.com/questions/19404666/is-main-a-keyword-in-java

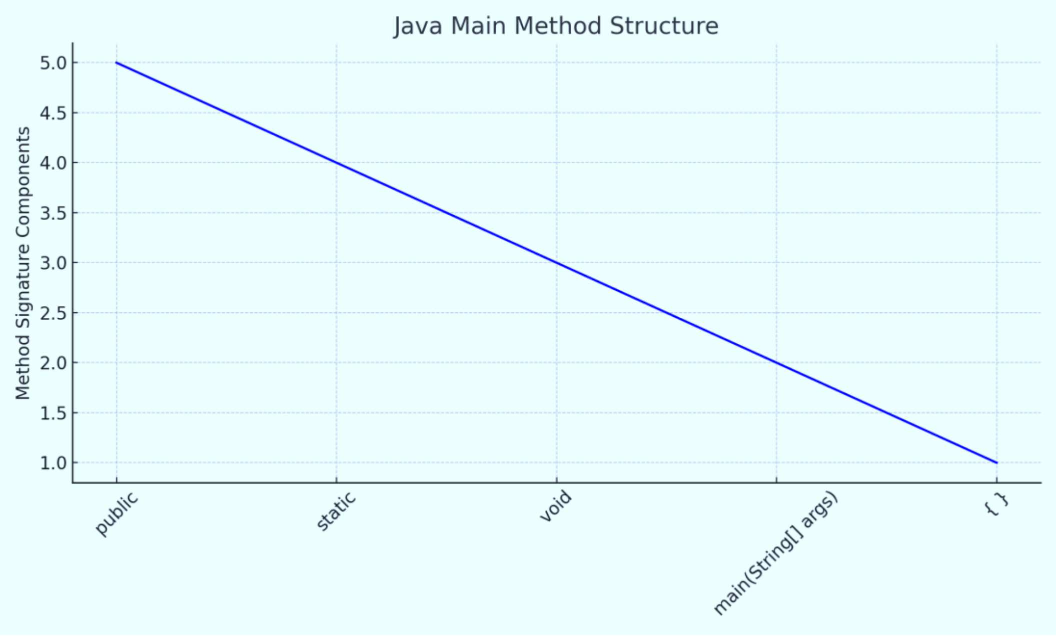
9. Can we write main method as "public void static" instead of "public static void"?

- No, the correct order is "public static void main (String [] args)".
- The compiler expects this exact signature to recognize the main method.
- Changing the order will cause a compilation error.

Example:

```
public static void main (String [] args) {
   System.out.println ("Hello, World!");
}
```

The following diagram explains the correct structure of the main method:



https://stackoverflow.com/questions/19404666/is-main-a-keyword-in-java

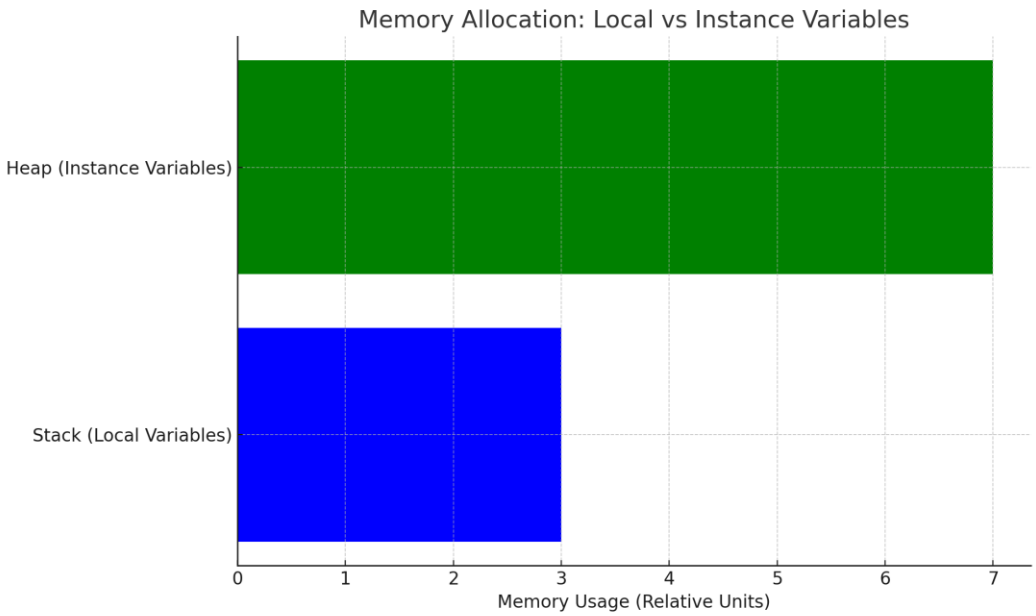
10. What is the default value of local variables in Java?

Local variables do not have default values.
- They must be initialized before use.
- Trying to access an uninitialized local variable will result in a compilation error.

Example:

```
public class Test

 {
   public static void main (String [] args)

{
      int num; // Compilation error
   }
}
```

The following diagram explains how local variables are stored:



https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

11. Value of String array of arguments in Main method when no arguments are passed?

In Java, the main () method accepts a single argument: a String array, commonly named args. This array holds the command-line arguments passed to the program during execution.

- If **no arguments** are passed when executing the program, the array is initialized but **empty**.
- The **length** of the array will be **zero**.

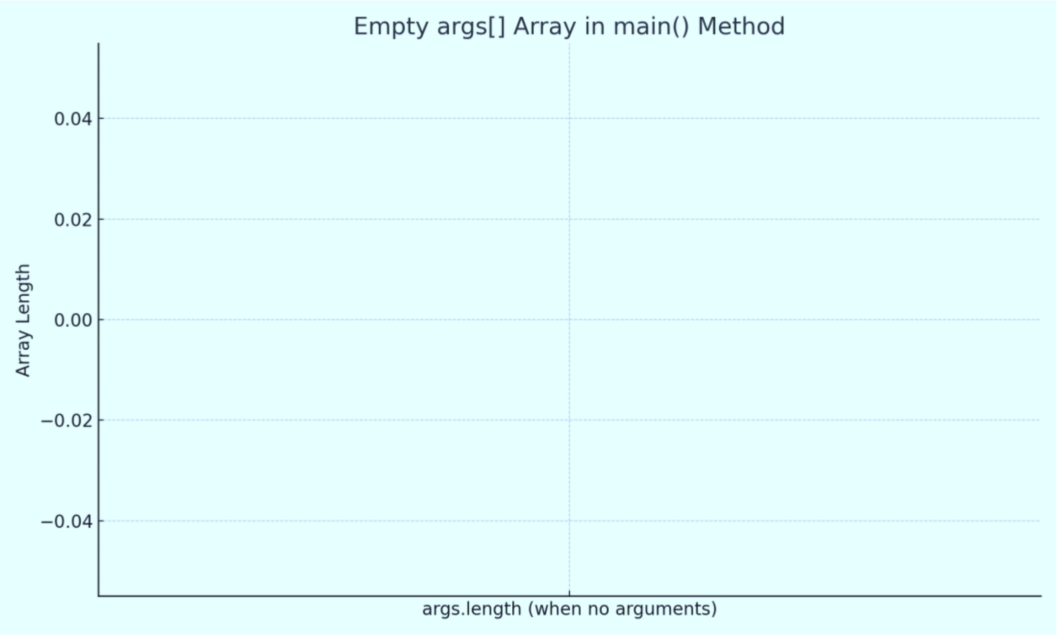
Key Points:

- String [] args is never null — it is always initialized.
- Accessing an index without checking its length results in ArrayIndexOutOfBoundsException.

Example:

```
public class Test {

    public static void main (String [] args) {

        System.out.println(args.length); // Output: 0
    }
}
```

Diagram:



https://www.w3schools.com/java/java_main.asp

12. What is the difference between byte and char data types in Java?

In **Java**, both `byte` and `char` are used to store data, but they are used for different types of values. Let's go over the differences in a way that's easy to understand:

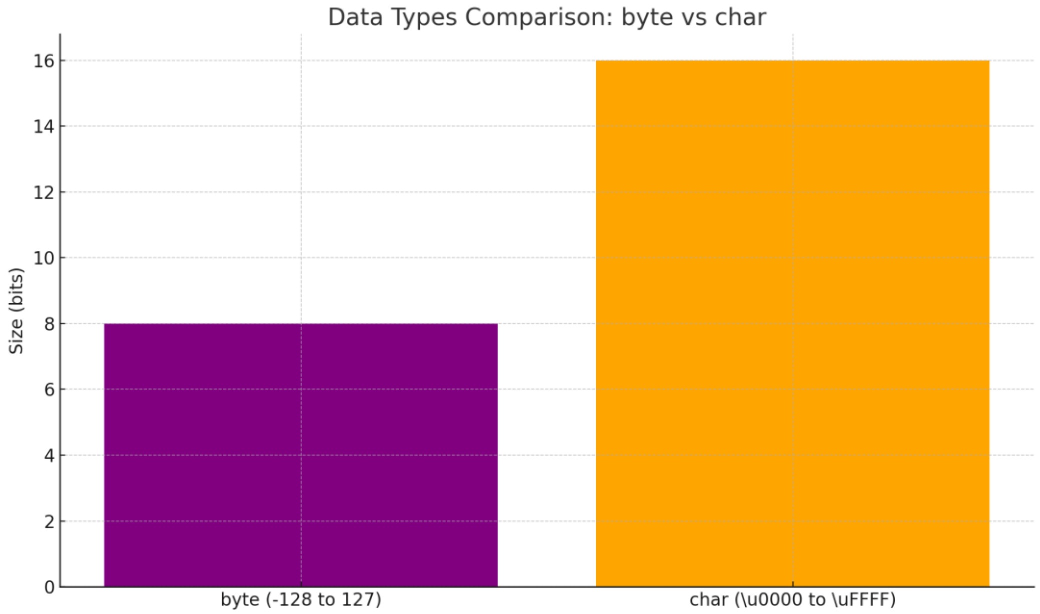
1. **Size (How much space they take in memory):**
   - A **byte** takes **1 byte** (8 bits) of memory.
   - A **char** takes **2 bytes** (16 bits) of memory.
2. **Range (What values they can hold):**
   - A **byte** can store **small numbers** from **-128 to 127**.
   - A **char** can store **Unicode characters** (like letters, symbols, and numbers) from **0 to 65,535**.

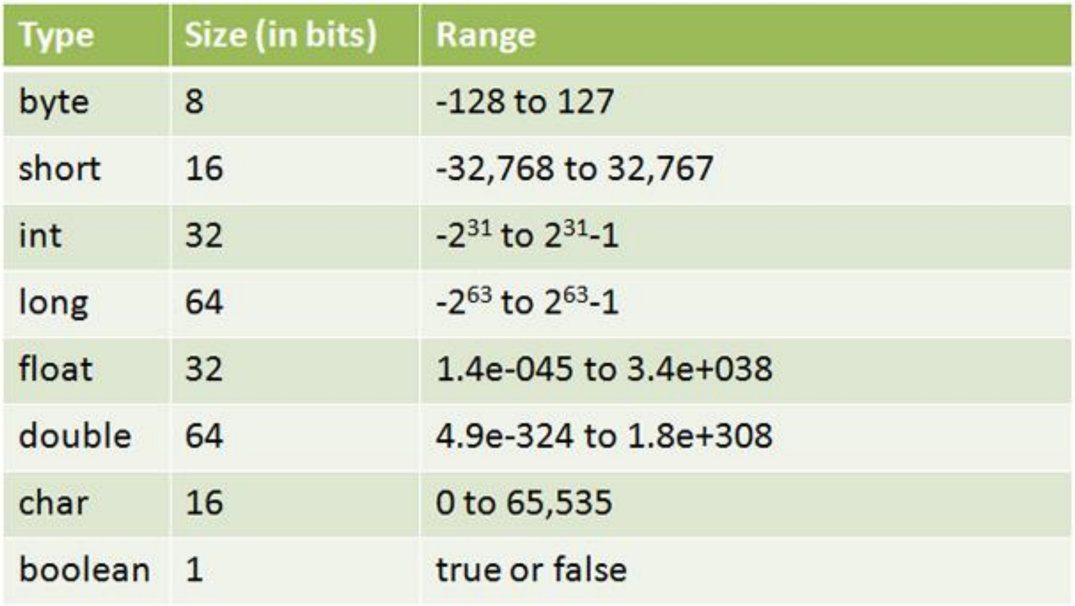
3. **Purpose (What they are used for):**
    - A **byte** is used for **small integer numbers**.
    - A **char** is used to store **a single character** like `'A'`, `'5'`, or `'@'`.
4. **Signed vs. Unsigned (Positive and negative values):**
    - **byte** is **signed** — it can have both **positive and negative** values.
    - **char** is **unsigned** — it only has **positive values** because it's used for characters, not negative numbers.
5. **Default value (What they are if you don’t set a value):**
    - A **byte's** default value is **0**.
    - A **char's** default value is **'\u0000'** (which means an empty character or **null character**).

Example:

```
byte b = 100;

char c = 'A';

System.out.println (b); // 100

System.out.println(c); // A
```

Diagram:





Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to 2^{31}-1
long	64	-2^{63} to 2^{63}-1
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Startertutorials.com

https://www.baeldung.com/java-char-byte-convert