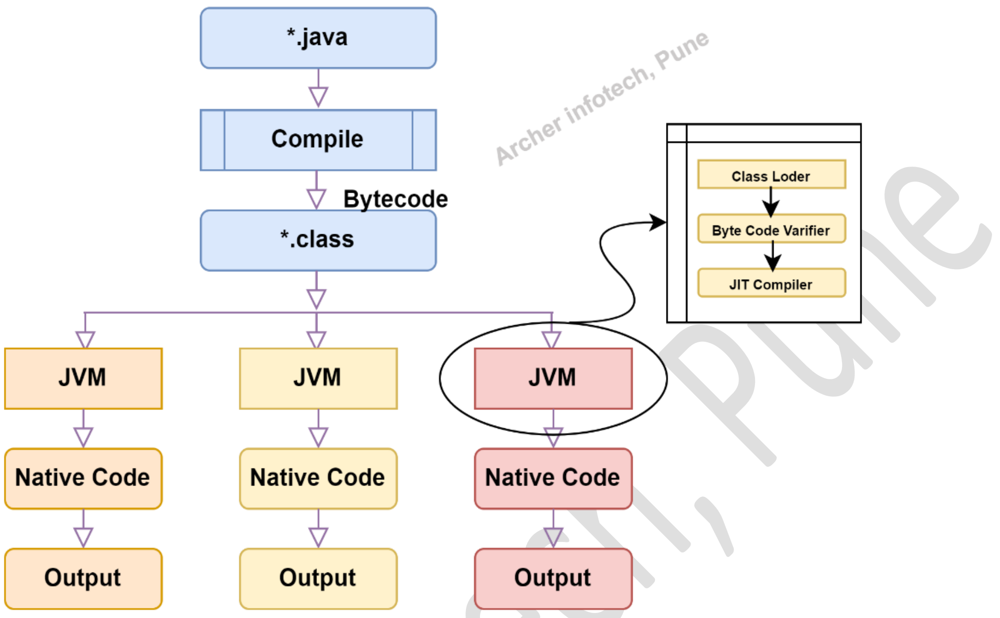
Java Program Execution:

The image provides a visual representation of the Java compilation process. Here's a breakdown of each step:



1. Java Source Code (.java):

- The process starts with a Java source code file written in the Java programming language. This file contains the code that you want to execute.

2. Compile:

- The Java compiler (javac) is used to compile the source code file.
- The compiler translates the human-readable source code into machine-readable bytecode.

3. Bytecode (.class):

- The output of the compilation process is a bytecode file with a .class extension.
- Bytecode is platform-independent, meaning it can be executed on any system that has a Java Virtual Machine (JVM).

4. Class Loader:

- The JVM's class loader is responsible for loading the bytecode into memory when it's needed.
- The class loader searches for the .class file on the classpath, which is a list of directories or JAR files where the JVM can find classes.

5. Bytecode Verifier:

- The bytecode verifier checks the bytecode for correctness and potential security threats.
- It ensures that the bytecode adheres to Java's syntax and semantics and doesn't contain malicious code.

6. JIT Compiler:

- The Just-In-Time (JIT) compiler is an optional component of the JVM that can improve performance.
- It translates frequently executed parts of the bytecode into native machine code, which can be executed more efficiently by the CPU.

7. JVM:

- The JVM is responsible for executing the bytecode or native code.
- It provides a runtime environment for Java applications, including memory management, garbage collection, and thread management.

8. Native Code:

- If the JIT compiler is used, some parts of the bytecode may be translated into native code, which is specific to the underlying hardware platform.
- This can improve performance by taking advantage of hardware-specific optimizations.

9. Output:

- The final output of the Java program is the result of the execution, which can be displayed on the screen, written to a file, or sent to another system.

In summary, the image shows the flow of a Java program from source code to output. The compilation process transforms the source code into bytecode, which is then loaded, verified, and potentially compiled into native code by the JVM. The JVM then executes the code and produces the output.

JVM Architecture

The JVM stands for **Java Virtual Machine**, so let's start with Virtual Machine first.

Virtual Machine (VM): A virtual machine is a software emulation of a physical computer. It provides an isolated environment where applications can run without interfering with other applications or the host operating system. VMs are commonly used for:

- **Running multiple operating systems on a single physical machine:** This allows for efficient resource utilization and isolation of different environments.
- **Testing applications in different environments:** VMs can be configured to simulate various hardware and software configurations.

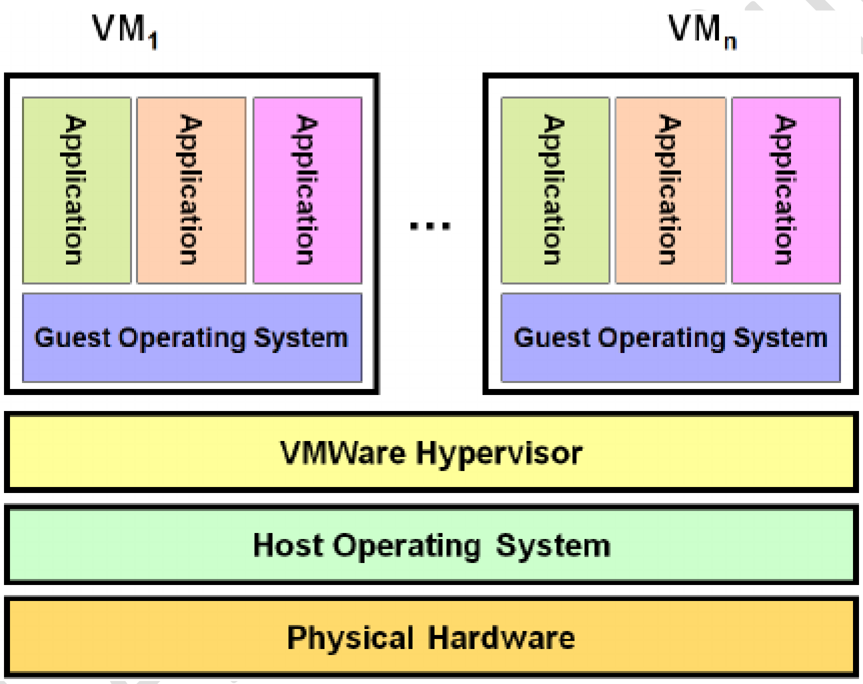
- **Providing a secure environment for running sensitive applications:** VMs can be isolated from the host system to protect against security threats.

Types of VM's:

There are primarily **two types of virtual machines** (VMs) based on their functionality and purpose: **System Virtual Machines** and **Process Virtual Machines**.

1. System Virtual Machines (System or Hardware Based)

A **System VM** provides a complete virtualized environment for an operating system (OS) to run. It allows for the complete emulation of a physical machine, enabling users to run multiple OS instances on a single physical machine, each OS running in isolation from others. These are widely used in data centers, cloud computing environments, and development/testing setups.



- **Examples**: VMware, VirtualBox, Microsoft Hyper-V, KVM (Kernel-based Virtual Machine).
- **Key Features**:
  - Allows multiple operating systems (OS) to run on the same physical hardware simultaneously.
  - Provides virtualized hardware resources (CPU, memory, storage, network) to each guest OS.
  - Hypervisors manage the system virtual machines.
    - **Type 1 Hypervisor**: Runs directly on the host's hardware (bare-metal), e.g., VMware ESXi, Microsoft Hyper-V.

  - **Type 2 Hypervisor**: Runs on the host OS as an application, e.g., VirtualBox, VMware Workstation.

- **Use Cases**:
  - Server consolidation: Running multiple virtual servers on a single physical machine.
  - Development and testing: Isolated environments for software testing.
  - Cloud computing: Provisioning virtual machines in public or private cloud environments.

2. Process Virtual Machines (Application Based)

A **Process VM** provides a runtime environment for executing a single program or process. It allows a program to be executed in a platform-independent manner by abstracting the underlying hardware and OS. These virtual machines run on top of an existing operating system and are terminated when the process (or application) they are running completes.

- **Examples**:
  - **Java Virtual Machine (JVM)**: Executes Java bytecode, making Java programs platform-independent.
  - **.NET CLR (Common Language Runtime)**: Executes .NET programs in a platform-independent way.
  - **Python Virtual Machine (PVM)**: Executes Python bytecode.
- **Key Features**:
  - Provides an abstraction layer for running platform-independent programs.
  - Exists only while the process or application is running.
  - Translates intermediate code (e.g., Java bytecode, .NET IL(MSIL Code)) into native machine code at runtime.
- **Use Cases**:
  - **Platform independence**: Allowing applications to run on any hardware or OS without modification.
  - **Portability**: Enabling cross-platform software development.
  - **Managed execution**: Features like garbage collection, security management, and exception handling.

Java Virtual Machine (JVM): The Java Virtual Machine (JVM) is a specific type of virtual machine designed to execute Java bytecode. It provides a platform-independent environment for running Java applications. This means that Java programs can be **compiled once and run on any system** that has a compatible JVM installed.

Key features of the JVM include:

- **Platform independence:** Java programs can run on various operating systems (Windows, macOS, Linux) and hardware architectures.
- **Memory management:** The JVM automatically manages memory allocation and deallocation using garbage collection.
- **Security:** The JVM has built-in security features to protect against malicious code.
- **Performance optimization:** The JVM uses techniques like Just-In-Time (JIT) compilation to improve performance.

In summary, a virtual machine is a general-purpose software emulation of a computer, while a JVM is a specialized virtual machine designed specifically for executing Java bytecode.

JDK, JRE and JVM:

JDK (Java Development Kit), **JRE (Java Runtime Environment)**, and **JVM (Java Virtual Machine)** are fundamental components of the Java programming environment. Each serves a specific role in the development and execution of Java applications. Let's explore each of them:

1. JDK (Java Development Kit)

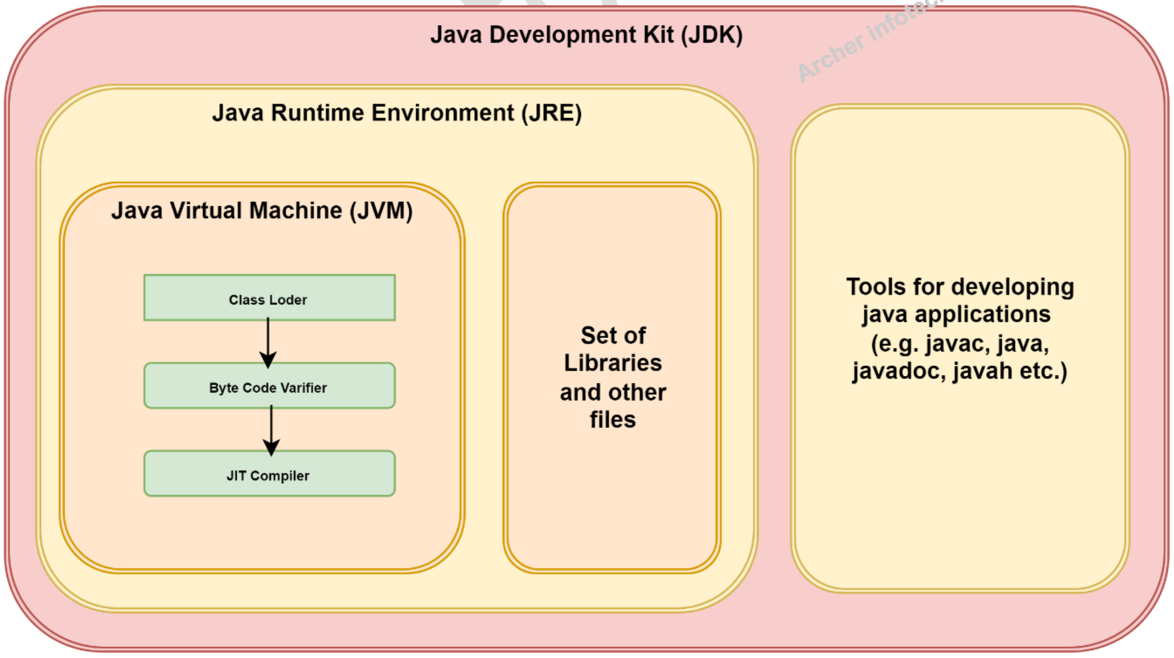
The **Java Development Kit** is a software development environment used to develop Java applications. It provides the necessary tools and libraries for writing, compiling, and debugging Java programs.

- **Key Components**:
  - **Javac (Java Compiler)**: Converts Java source code (.java files) into bytecode (.class files).
  - **Java Runtime Environment (JRE)**: A component of the JDK, which allows Java applications to run.
  - **Debugger**: A tool to debug and analyze Java programs.
  - **Additional Tools**: Includes tools like javadoc (to generate documentation), jar (to bundle files into a JAR), and others for developers.
- **Purpose**: The JDK is essential for Java developers as it includes the tools necessary to write and build Java programs.
- **Usage**:
  - Writing code
  - Compiling source files
  - Debugging applications
  - Packaging and distributing Java applications

2. JRE (Java Runtime Environment)

The **Java Runtime Environment** is a package that provides the environment needed to run Java programs. It includes the JVM, libraries, and other files necessary to execute Java bytecode.

- **Key Components**:
  - **JVM (Java Virtual Machine)**: The engine that executes Java bytecode.
  - **Java Class Libraries**: Pre-compiled libraries used by Java applications to perform various tasks.
  - **Other Runtime Components**: Such as configuration files and property settings needed by the JVM.
- **Purpose**: The JRE allows users to run Java programs. It does not contain tools for developing Java applications (like the compiler).
- **Usage**:
  - Running Java applications on any platform.
  - Typically used by end-users who need to execute Java programs but don't need development tools.


Java Development Kit (JDK)
Java Runtime Environment (JRE)
Java Virtual Machine (JVM)
Class Loder
Byte Code Varifier
JIT Compiler
Set of Libraries and other files
Tools for developing java applications (e.g. javac, java, javadoc, javah etc.)

3. JVM (Java Virtual Machine)

The **Java Virtual Machine** is a crucial part of both the JDK and JRE. It is an abstract machine that enables a computer to run Java programs by converting the platform-independent bytecode into machine-specific instructions.

- **Key Components**:
    - **Class Loader**: Loads class files into the memory.
    - **Execution Engine**: Interprets or compiles the bytecode into machine code for the underlying hardware.
    - **Memory Management**: Manages memory allocation and garbage collection for Java objects.
- **Purpose**: The JVM provides platform independence by executing bytecode on any operating system without needing changes to the program.
- **Usage**:
    - Running and managing the lifecycle of Java applications.
    - Managing resources like memory and threads during execution.

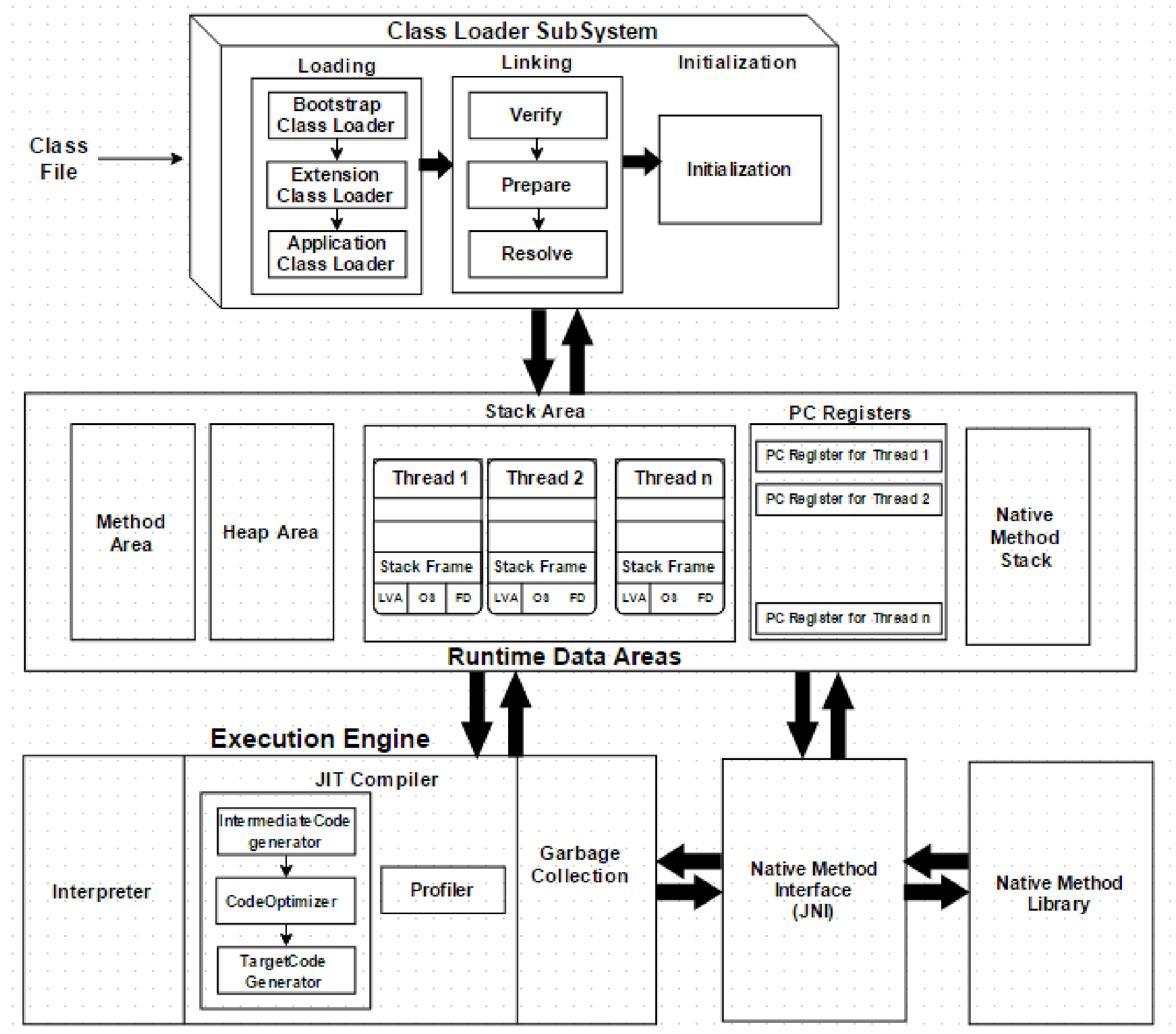
Relationship Between JDK, JRE, and JVM:

- **JDK** = **JRE** + Development tools (like compiler, debugger)
- **JRE** = **JVM** + Libraries for running Java programs
- **JVM** = Core component of JRE that executes the bytecode

The **Java Virtual Machine (JVM)** is an abstract computing machine that enables a computer to run Java programs. It is responsible for converting Java bytecode into machine-specific instructions, allowing Java programs to be platform-independent. Here's an explanation of the JVM architecture.

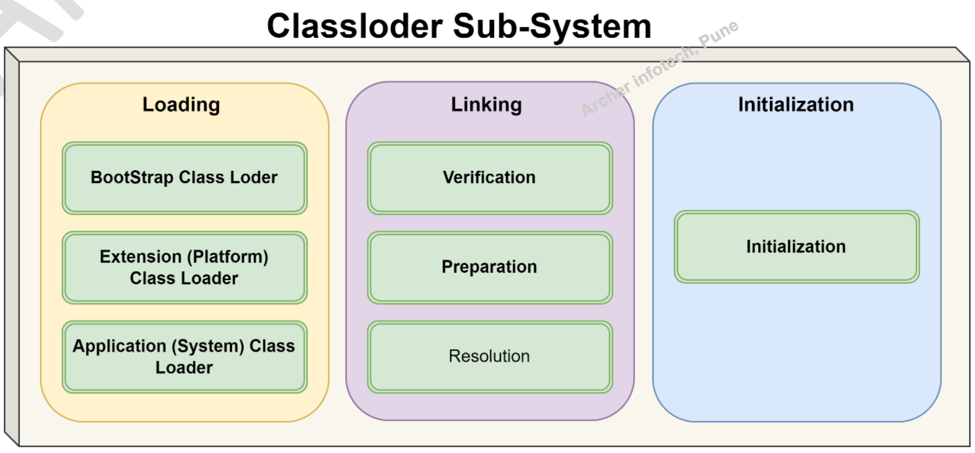
Mainly there are five different components of JVM:

1. Class Loader Subsystem
2. Memory Area / Runtime Data Areas
3. Execution Engine
4. Native Method Interface (JNI)
5. Native Method Libraries



1. Class Loader Subsystem

The **Class Loader Subsystem** in the **JVM (Java Virtual Machine)** is responsible for dynamically loading Java classes into memory during runtime. It plays a crucial role in managing the lifecycle of Java classes, ensuring they are loaded, linked, and initialized properly. The class loading process can be divided into three main steps: **Loading**, **Linking**, and **Initialization**. We'll focus on **Loading**, which is the first and essential step in the lifecycle of a Java class.



A. Loading Phase:

The **Loading phase** in the Class Loader Subsystem is responsible for finding and loading the .class files (compiled Java bytecode) into the JVM. In this phase, the following key operations occur:

- **Locating the .class file**: The class loader searches for the class file by its fully qualified name (package and class name). It can search for classes in different sources like:
  - Local file systems
  - Remote locations (network resources)
  - JAR files
  - Custom locations if you're using a custom class loader
- **Loading the bytecode into memory**: Once the .class file is located, the class loader reads the bytecode from the file and loads it into the method area of the JVM memory. At this point, the bytecode itself is not yet executed, but it's made available for further processing.
- **Creating an in-memory representation**: After loading the class file, the JVM creates an in-memory representation of the class. This includes:
  - Class structure (fields, methods, interfaces)
  - Constant pool (a table of constants, strings, and method references)
  - Metadata related to the class.

What happened when the class is loaded into memory:

1. The reading of *.class and is loaded in the Method Area of memory.
2. In the method area it is represented in the binary form.
3. In the next step it will create the object of class Class, in the Heap Area to represent the loaded class information.
4. The Class object can be used by the programmer to get the Class level information like fully qualified name of class, Parent name, methods and variable information.

Let's see the example below,

```
import java.lang.reflect.*;
class Demo
{
   private int x;
   private double y;
   public void in()   {  }
   public void out() {  }
}

public class Main
{
     public static void main(String[] args) throws Exception {
         Class c=Class.forName("Demo");
         Method []m=c.getDeclaredMethods();
```

```
            for(Method m1 : m)
            {
                        System.out.println(m1);
            }
            Field []f=c.getDeclaredFields();
            for(Field f1 : f)
            {
                        System.out.println(f1);
            }
        }
    }
```

Types of Class Loaders:

The JVM provides a hierarchical delegation model for class loading, which means that class loaders are organized in a parent-child hierarchy. There are **three main types of class loaders**:

1. **Bootstrap Class Loader**:
    - The root class loader in the hierarchy.
    - It is responsible for loading core Java classes from the rt.jar file (i.e., classes from java.lang.*, java.util.*, etc.).
    - It is written in native code (platform-specific) and is part of the core JVM.
2. **Extension (Platform) Class Loader**:
    - Child of the Bootstrap Class Loader.
    - It loads classes from the extensions directory (usually the JDK's lib/ext directory) or the JAVA_HOME/jre/lib/ext directory.
    - It is responsible for loading optional packages or libraries that are provided with the Java runtime.
3. **Application (System) Class Loader**:
    - Child of the Extension Class Loader.
    - It loads classes from the application classpath (set by the CLASSPATH environment variable or command-line options).
    - It loads all user-defined classes and third-party libraries.

Custom Class Loaders:

Developers can create custom class loaders by extending the ClassLoader class. This allows for the customization of how and from where classes are loaded (e.g., loading from a network, databases, or encrypted files).

B. Linking Phase:

In the JVM (Java Virtual Machine), after the Loading phase in the Class Loader Subsystem, the next important phase is Linking. The Linking phase prepares the class for execution by verifying, preparing, and resolving its dependencies. This phase ensures that the class's bytecode is compatible with the JVM's rules and that the memory required for its fields and methods is allocated. Linking bridges the gap between loading the bytecode into memory and running it in the JVM.

The Linking phase can be broken down into three sub-phases: Verification, Preparation, and (Optional) Resolution.

1. Verification:

The **Verification** step ensures that the bytecode loaded in the JVM is valid and does not violate any rules of the Java Language Specification (JLS). This step is critical for maintaining the security and integrity of the JVM, especially when running bytecode from untrusted sources (e.g., when downloaded over the internet).

- **What is Verified**:
  - **Class File Format**: Ensures the class file is well-formed and adheres to the structure defined by the JVM specification.
  - **Bytecode Validity**: Checks that the instructions in the bytecode are valid and will not corrupt the memory or violate access controls.
  - **Data Types and Operand Stack**: Ensures that data types used in bytecode instructions match expected types, and that the operand stack is properly utilized.
  - **Method and Field Access**: Ensures that the code adheres to Java's access control rules (e.g., you can't access private fields from outside the class).
- **Verification Example**: For instance, the verifier checks if a method is trying to return a value of the wrong type. If the method declares it will return an int, but the bytecode tries to return a String, the verifier will detect this and throw an exception (VerifyError).
- **Result**: If verification succeeds, the class can proceed to the next phase. If it fails, a VerifyError is thrown, and the class is not loaded into the JVM.

2. Preparation:

The **Preparation** phase involves allocating memory for static fields (class variables) of the class and initializing them to their **default values**.

- **What Happens**:
  - **Memory Allocation**: During preparation, memory is allocated in the JVM's **method area** for all the class's static fields (but not instance fields, which are allocated later when objects are created).
  - **Default Initialization**: Static fields are initialized to default values according to their types:
    - Integer types (e.g., int, long) are initialized to 0.
    - Floating-point types (e.g., float, double) are initialized to 0.0.
    - Boolean fields are initialized to false.
    - Reference types (e.g., objects) are initialized to null.

3. Resolution (Optional):

The **Resolution** phase is responsible for converting symbolic references in the bytecode into direct references. A symbolic reference is a reference in the bytecode that uses the **names** of classes, fields, methods, or interfaces rather than their actual memory addresses. During resolution, the JVM replaces these symbolic names with direct references (actual memory addresses or method locations).

- **Types of Symbolic References Resolved**:
    - **Class References**: Resolving references to other classes (e.g., java.lang.String).
    - **Field References**: Resolving references to static or instance fields.
    - **Method References**: Resolving method calls, including virtual and interface method calls.
- **Lazy Resolution**: In many JVM implementations, resolution is done **lazily**, meaning that symbolic references are resolved only when they are first used during execution (not immediately after the class is loaded). This improves performance because not all symbolic references may be needed during execution.

C. Initialization

In the JVM (Java Virtual Machine), the Initialization phase is the final step of the Class Loader Subsystem. It occurs after the Loading, linking (which includes verification, preparation, and optional resolution), and is responsible for executing the code that initializes class variables and static blocks. This step ensures that all static fields are assigned their proper values (including explicit initialization) and any static blocks (static initializers) are executed in the order they appear in the class.

Overview of Initialization in the Class Loader Subsystem

1. **Static Field Initialization:** Assigns explicit values to static variables.
2. **Static Block Execution:** Executes static initialization blocks, which can contain any logic needed to initialize the class.
3. **Order of Execution:** The class variables and static blocks are executed in the order they appear in the source code.
4. **Triggered by Class Use:** Initialization occurs when the class is first actively used (such as when an instance is created or a static method is invoked).

```
class ClassLoaderExample
{
    public static void main(String[] args)
    {
            System.out.println(String.class.getClassLoader());
            System.out.println(ClassLoaderExample.class.getClassLoader());
    }
}
```

**** The Bootstrap ClassLoader is implemented in native code, so it's not represented by a Java object. For core Java classes loaded by the Bootstrap ClassLoader, null is returned to indicate the absence of a Java-level class loader.***

2. Memory Area / Runtime Data Areas

The JVM uses several memory areas to manage program execution, collectively called **Runtime Data Areas**. These include:

- **Method Area**: Stores class-level data like class structure, fields, methods, and constant pool.
- **Heap**: Stores objects and their instance variables at runtime. The heap is shared by all threads.
- **Stack**: Each thread has its own stack, which stores local variables, partial results, and function call information (frames).
- **PC (Program Counter) Register**: Each thread has its own PC register, which holds the address of the JVM instruction currently being executed.
- **Native Method Stack**: Holds the data for native method calls (methods written in other languages like C/C++).

Let’s discuss each in detail:

- **Method Area:**

The **Method Area** is a crucial part of the JVM's memory structure. It stores class-level information necessary for the execution of Java programs, and it's shared among all threads in the JVM, meaning it is **thread-safe**.

The Method Area holds data that describes class structure and details needed for the execution of methods. Since Java is class-based, the JVM requires information about loaded classes (such as fields, methods, and constants) to execute the program. This is the primary role of the Method Area, as **it provides a place for storing class metadata, bytecode, and constant values** that remain in use throughout the lifetime of a Java application.

The Method Area stores several different types of data:

Class Information (Metadata): It refers to the detailed structural information about a class that the JVM stores in the Method Area once a class is loaded. This includes the fully qualified name of the class (its package and class name), the name of its superclass (from which it inherits), and the interfaces it implements. It also holds the access modifiers of the class (such as public, abstract, final), and information about its constructors, methods, and fields. The class metadata allows the JVM to know how to instantiate objects of the class, how to invoke methods, and how to access or modify its fields. This metadata is crucial for the JVM to manage classes dynamically during program execution.

Field Information: It refers to the data that describes the variables defined in a class, and it is stored in the Method Area when the class is loaded. This information includes the name of the field, its data type (e.g., int, String, or a reference to another object), and its access modifiers (such as public, private, static, final). Fields can be either instance variables (specific to each object of the class) or class variables (marked as static, shared across all instances). The JVM uses this information to determine how to allocate memory for objects and how to access and modify these fields during execution. In cases where a field is marked as final, the field's value is stored in the constant pool, ensuring its immutability after initialization.

Method Information: It refers to the detailed metadata about methods in a class, crucial for the JVM to execute the methods during runtime. This includes the method's signature, which uniquely identifies it through its name, parameter types, and return type. The access modifiers (such as public, private, static, final) define the visibility and behavior of the method. Each method is compiled into bytecode, a platform-independent code that the JVM executes, stored in the Method Area. During execution, the JVM uses a local variable table to store parameters and local variables for the method, along with an operand stack to hold intermediate computation results. Methods also have an exception table that specifies how to handle exceptions thrown during execution. Each time a method is invoked, a stack frame is created in the Java Stack to manage the method's local variables, operand stack, and the return address, allowing the JVM to keep track of the method call and where to resume execution after it completes. Static methods, which belong to the class itself, and non-static methods, which are tied to instances, are distinguished by the JVM using method metadata. For native methods, implemented in languages like C/C++, the JVM uses the Native Method Stack. All of this information enables the JVM to execute methods efficiently, manage memory, and handle control flow in Java programs.

The Runtime Constant Pool (RCP): It is a dynamic data structure within the JVM's Method Area that holds constants and symbolic references used during the execution of a Java program. When a class is loaded, its constant pool is created, storing a variety of elements such as literals (e.g., numeric constants like int and float, string literals) and symbolic references to fields, methods, and classes. Symbolic references allow the JVM to resolve links between different parts of the program, such as method invocations or field accesses, without directly using memory addresses until runtime. Additionally, the constant pool helps manage string literals efficiently by storing them in a String Pool, enabling reuse and reducing memory overhead. The Runtime Constant Pool serves as a key resource for the JVM to access class-level data and resolve method calls and object references, supporting efficient program execution.

Static variables: Static variables belong to the class rather than any specific instance, and they are stored in the Method Area. This ensures that static variables persist and are accessible across multiple objects and threads.

- **Heap:**

The Heap in the JVM's memory model plays a central role in managing dynamic memory allocation, specifically for objects and arrays. It is the largest part of the Runtime Data Area and is shared by all threads in a Java application. The Heap is where Java objects (created using the new keyword) and arrays are stored, making it the key area for managing objects' lifecycle and memory management in Java. The Heap supports dynamic memory allocation and works in conjunction with the garbage collection system to ensure efficient memory use.

When an object is created in the Heap, the JVM stores a reference (a memory address) to the object in the Java Stack (for local variables or method arguments). The actual object data, including its fields, is located in the Heap. Access to the object's fields is done through the reference, meaning that while the reference resides on the stack, the object itself is accessed via the Heap.

While the JVM can expand the Heap dynamically, it is bounded by the maximum size (-Xmx parameter). If the Heap becomes full and there's no space for new objects, and the garbage collector cannot reclaim enough memory, the JVM throws an **OutOfMemoryError**.

- **Stack:**

In the Java Virtual Machine (JVM), the **stack** is a critical component of the data access area, primarily used for managing method execution and local variables. Each thread in a JVM has its own stack, meaning that stacks are thread-specific and not shared among threads. When a method is called, a **stack frame** is created and pushed onto the thread's stack. This frame contains local variables (both primitive data types and object references), operand stacks (used for executing bytecode instructions), and the return address, which points to the next instruction in the calling method after the method completes.

The **JVM stack** is essential for method invocation and execution, as it stores information required during method calls and returns. Once a method finishes executing, its corresponding stack frame is popped from the stack, and the program resumes execution from where the method was invoked. The stack operates in a last-in, first-out (LIFO) manner, ensuring that the most recently called method is executed first.

Unlike the heap, memory management in the stack is straightforward because each method's frame is automatically freed after the method completes. This makes stack memory management fast and efficient. However, the size of the stack is limited and can be adjusted with JVM options like -Xss. If a program has deep recursion or too many nested method calls, it can lead to a **StackOverflowError**, indicating that the stack has exceeded its memory limit.

Stack frame contains three primary components:

1. **Local Variable Array (or Local Variable Table):** This is where the method's local variables are stored, including both primitive types (like int, float, etc.) and references to objects in the heap. For instance, method parameters are also stored in this table.
2. **Operand Stack:** This is used as a workspace to store intermediate values during the method execution. JVM instructions (bytecode) operate on this stack by pushing operands onto it and popping them off to perform operations (like adding two numbers or comparing values). The operand stack grows and shrinks as the method's instructions are executed.
3. **Frame Data (including Return Address):** This includes metadata about the method's execution, such as the return address, which indicates where the execution should resume after the method call completes. It may also store additional information needed for debugging, exceptions, or stack unwinding.

3. Execution Engine

The Execution Engine is the core of the JVM responsible for executing bytecode instructions. It consists of the following:

- **Interpreter**: Reads and executes bytecode one instruction at a time. This can be slow since each instruction is interpreted at runtime.
- **Just-In-Time (JIT) Compiler**: To improve performance, the JIT compiler translates bytecode into native machine code for faster execution. Once a method is compiled, the native code is used for subsequent calls.
- **Garbage Collector (GC)**: Automatically manages memory by reclaiming memory used by objects that are no longer referenced.

4. Native Method Interface (JNI)

The JNI allows the JVM to interact with native applications (code written in other programming languages like C/C++). This interface enables Java applications to call or be called by native libraries.

5. Native Method Libraries

These libraries contain native code that is required by the application. The JVM can load these libraries as needed during execution.

Interview Questions:

Basic Questions:

1. What is the JVM, and why is it important in Java?
2. What are the main components of the JVM?
3. What is the role of the Class Loader in the JVM?
4. Explain the difference between JDK, JRE, and JVM.
5. What are runtime data areas?
6. What is the difference between stack memory and heap memory in the JVM?
7. Explain the JVM memory model, including the different runtime data areas.
8. What is object slicing in the JVM?
9. What are Java class loaders? What are the key features of Java class loader.
10. Describe the process of class loading in the JVM.
11. What is garbage collection, and why is it important in Java?
12. What are the different types of garbage collectors in the JVM?
13. What is method area (permanent generation) in the JVM? How is it different in Java 8?

Advanced Questions:

14. What are JVM stack frames, and how are they managed?
15. How does the JVM manage thread safety with shared resources?
16. Explain the difference between the static_cast and dynamic_cast in terms of JVM type conversion.
17. How does the Just-In-Time (JIT) compiler work in the JVM?
18. Explain the difference between stop-the-world garbage collection and concurrent garbage collection.
19. What is adaptive optimization in the JVM?
20. What is class loader delegation in the JVM, and why is it important?

21. What is a memory leak in the context of JVM, and how does it happen?
22. What are soft, weak, and phantom references in the JVM, and how are they used?
23. What is the difference between PermGen and Metaspace, and why was Metaspace introduced in Java 8?
24. How does the JVM handle multithreading, and what is the role of the Program Counter (PC) Register?
25. What is the role of the volatile keyword in the JVM memory model?
26. How does the JVM implement thread synchronization at the bytecode level?
27. How can you monitor and optimize JVM performance?
28. What are the common performance tuning options for the JVM?
29. How does the JVM optimize frequently executed code?
30. Explain escape analysis in the JVM.
31. What tools can you use to analyze JVM memory issues?
 (e.g., memory leaks, excessive garbage collection)
32. What does the OutOfMemoryError mean in the context of the JVM, and how would you handle it?
33. How would you investigate high CPU usage by the JVM?