CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

Lecture 0 – Course Introduction & Plan

Course Aims and Objectives

- Learn the basic programming with java (Java Architecture, Tokens, Data types, Operators, Conditional and Looping statements)
- Understand the Object-oriented programming with java using object-oriented concepts
- Learn the advanced java programming using exception handling, multithreading, file handling, and graphical applications
- Lean basic concepts about Arrays and methods in java
- Classes and Objects - Creating Classes and Objects, Constructors, Method overloading, "this" Reference, Garbage Collection
- Inheritance - Implementing Inheritance, Method overriding, "final" Keyword, Abstract Classes and Methods, Dynamic Binding
- Leann the Exception Handling
- Learn how to use Java collections
- GUI Applications with Java Swing
- Learn basic Network programming with java
- Learn basic multithreading concepts

Textbooks

- Paul Deitel, Harvey Deitel, Java: How to Program, , Early Objects, 11th Edition.
- Cay S. Horstmann, **Big Java: Late Objects**
- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- Java documentation https://docs.oracle.com/javase

Topic covered

Java basics	**Data Types, Methods, 1D array and 2D array, Scanner, Methods, Operators, If statement, loops.**
OOP	**Classes, objects, Inheritance, Encapsulation, Abstraction, Polymorphism**
Files	**Handling IO Text and Binary files**
Exceptions	**Handling Exceptions**
JavaSwing	**Graphical User Interface Elements based JAvaSwing**
Event Driven programming	**Mouse Listeners, Actions, Window listener**
Collections	**ArrayList, List, Vecto , HashSet ,…..etc**
DB	**JDBC - Java Database Connectivity**
Threads	**Threads and shared objects**
Network Programming	**Sockets Programming**

Grade Policy

- Final 50
- Midterm 20
- Practical Quizzes 20
- Project 15
- Note total semester work grade is 50
- If the student gets a mark above 50, he will only get 50.

Section

- Get involved with the story.

Practice.. Practice.. Practice..

Course Project Policy

01

Project groups: Each group will be 4 to 5 students.

02

Project Must be implemented based om MVC concept

03

GUI can be implemented using Swing or Javafx[Bounce]

Academic Integrity

You can discuss ideas and methodology for the homework (assignments / sheets) with other students in the course, but you must write your solutions completely independently.

We will be code-checking to assess similar submissions or submissions that use code from other sources.

Java Setting up

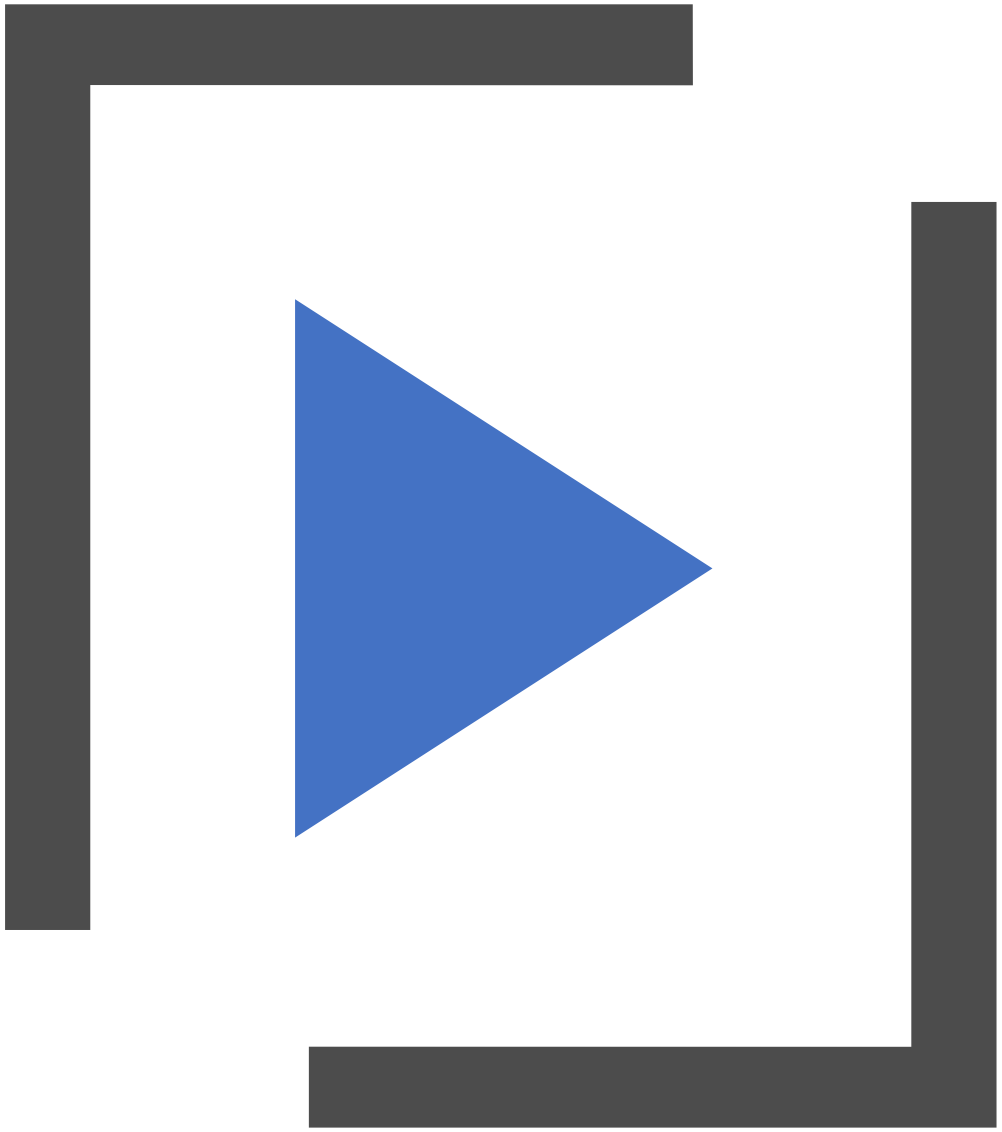
- Integrated Development Environment (IDE): IDE provides a sophisticated GUI and editor, with integrated facilities for compiling, running, and debugging programs
- Java 2 Standard Edition SDK (Software Development Kit): //What we need..
  - SDK includes everything you need to compile and run Java.
  - SDK tools are used for compiling and running Java programs on a variety of platforms
  - The Java compiler (**javac**) translates Java source into bytecode.
  - The Java interpreter (**java**) translates and executes Java bytecode. Java interpreters are available for many different computer systems.
- Java 2 Enterprise Edition (J2EE) → for large-scale systems
- Java 2 Micro Edition (J2ME) → for mobile phones
- Java 2 Standard Edition (J2SE) – Our Course -

Java Setting up

- We can work with Netbeans/Eclipse/ IntelliJ
  - https://netbeans.org/downloads/
  - https://eclipse.org/downloads/
  - JDK http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - Java documentation https://docs.oracle.com/javase

Java Setting up For JavaFX

- Install first the JDK and We will use Zulu JDK-fx
- Java Download | Java 7, Java 8, Java 11, Java 13, Java 15, Java 17, Java 19 - Linux, Windows & macOS (azul.com)
- IntelliJ IDEA Community Edition
- Download IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains
- We will use Scene Builder for GUI (Drag & Drop)
- Scene Builder - Gluon (gluonhq.com)

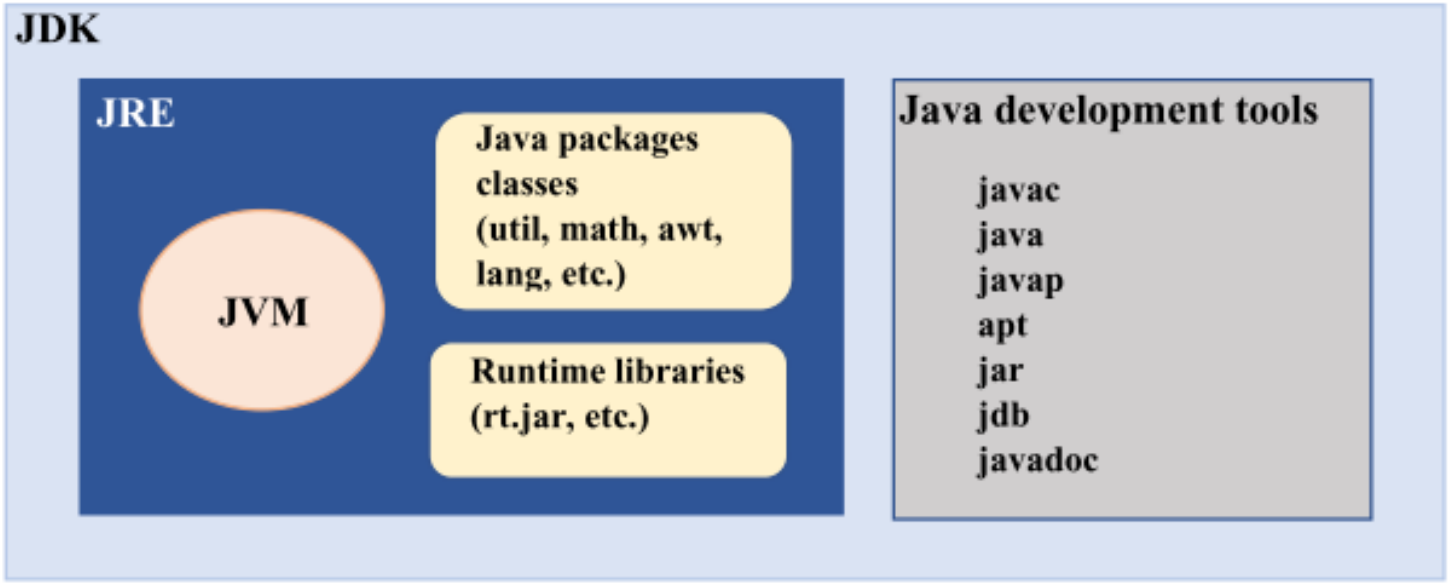


Video for Configuration

- https://youtu.be/LTgClBqDank

What Is JDK?

- JDK, or **Java Development Kit**, consists of tools that we use JDK to develop and run Java code. Before developing and running Java code, you should install it on your computer or system.
- The picture represents the structure of JDK



What Is JDK?

- The **Java Development Kit (JDK)** is a software development environment used for developing Java applications and applets.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

What Is JDK?

As you see, **JDK** consists of **JRE** and **Java development tools.**
JRE or ***Java Runtime Environment is*** a package that provides an environment to **only run (**not develop) the Java program(or application)on your machine. It is only used to run Java programs.

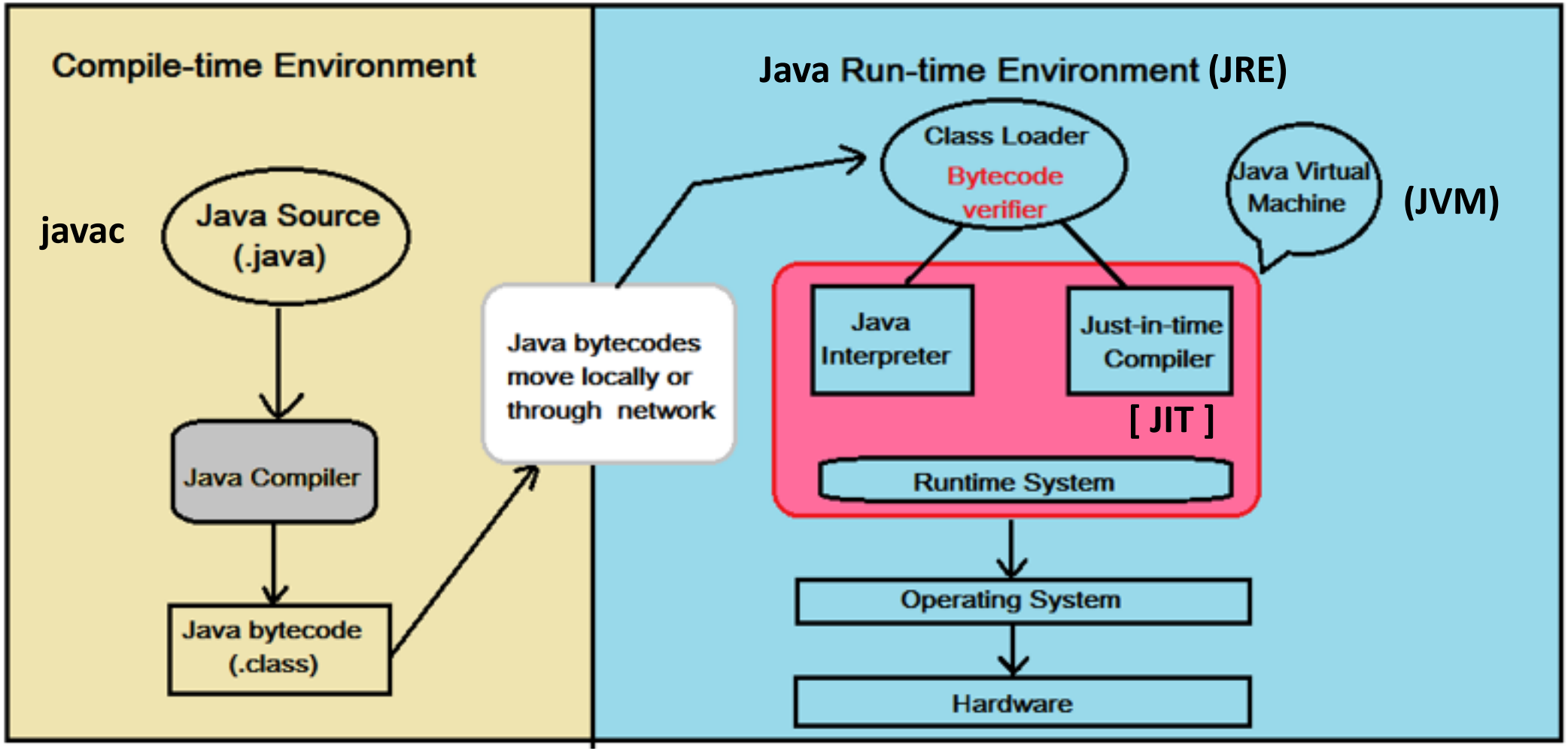
JDT or **Java Development tools** consist of many tools like compilers, debuggers, and other development tools.

The most important part of JDK and JRE is **JVM** (**Java Virtual Machine)** and its responsibility is the execution of code line-by-line. It's also known as an Interpreter. In the following, we will get more acquainted with it.

Java Virtual Machine (JVM)

- The Java Virtual Machine is a specification that provides a runtime environment in which java bytecode can be executed. It means JVM creates a platform to run Java bytecode(.class file) and converting into different languages (native machine language) which the computer hardware can understand. Actually, there is nothing to install as JVM. When the JRE is installed, it will deploy the code to create a JVM for the particular platform. JVMs are available for many hardware and software platforms.

How does Java Code Compile And Run?



How does Java Code Compile And Run?

- **Step 1:** Writing the source and save it with extinction **.java**
- **Step 2:** compile the code in the command prompt by using the command line or atomically with IDE javac filename.java).
- This invokes the Java Compiler. The compiler checks the code for syntax errors and any other compile time errors and if no error is found the compiler converts the java code into an intermediate code(**filename. class** file) known as **bytecode.**
- This **intermediate code is platform-independent** (you can take this bytecode from a machine running windows and use it in any other machine running Linux or macOS etc).
- Also, this bytecode is an intermediate code, hence it is only understandable by the **JVM (Java Virtual Machine)** and not the user or even the hardware /OS layer.

How does Java Code Compile And Run?

- **Step 3:** This is the start of the Run Time phase, where the bytecode is loaded into the JVM by the **class loader**(another inbuilt program inside the JVM).
- **Step 4:** Now the **bytecode verifier** checks the bytecode for its integrity and if no issues are found passes it to the interpreter. For example, if in the program, we use a variable that has not been declared, or if the run-time stack overflows, it will throw an Exception and the compiling process will stop.
- **Step 5:** Since java is both compiled and interpreted language, now the java **interpreter and Just in time Compiler (JIT)** inside the JVM convert the bytecode into executable machine code and passed it to the OS/Hardware i.e. the CPU to execute.

Just in time Compiler (JIT)

- The JIT compiler allows Java to be both a compiled and interpreted language. It combines the performance advantage of a compiled language with the flexibility of an interpreted language.
- While the JVM interprets bytecode, the JIT analyses execution and dynamically compiles frequently executed bytecode to machine code. This prevents the JVM from having to interpret the same bytecode over and over.

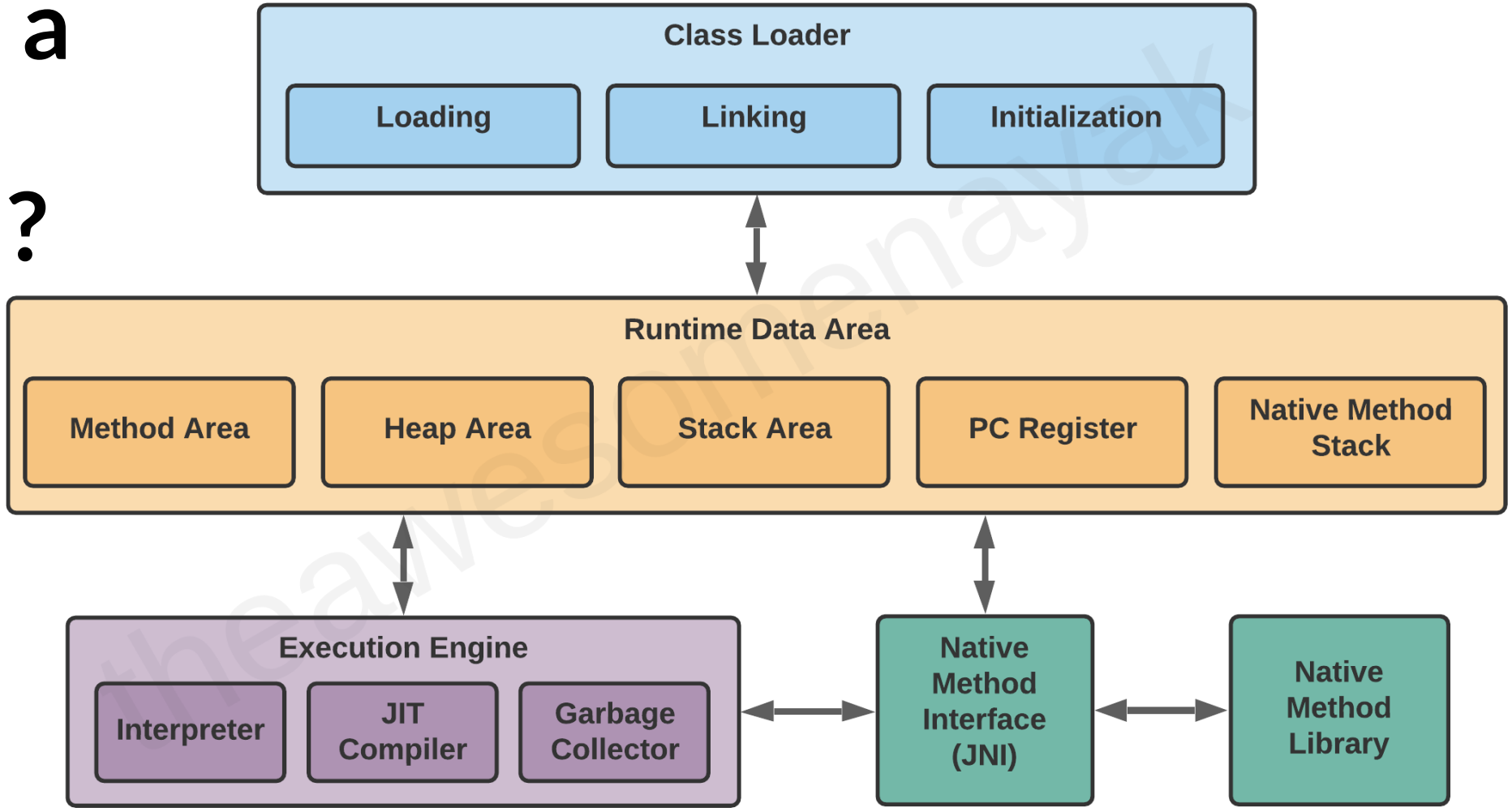
How the JIT Compiler Works in Java

- The source code you write (.java) is compiled by javac to bytecode. When your program executes, the JVM starts interpreting this bytecode.
- The interpreter translates the bytecode into machine code and feeds it to the CPU for processing.
- While this happens, the JIT profiles and analyses the code execution. It tracks the most frequently called methods and dynamically compiles these to machine code at runtime.

JVM(Java Virtual Machine)

- **JVM(Java Virtual Machine)** runs Java applications as a run-time engine. JVM is the one that calls the **main** method present in a Java code. JVM is a part of JRE(Java Runtime Environment).
- Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

What is a JVM in Java ?



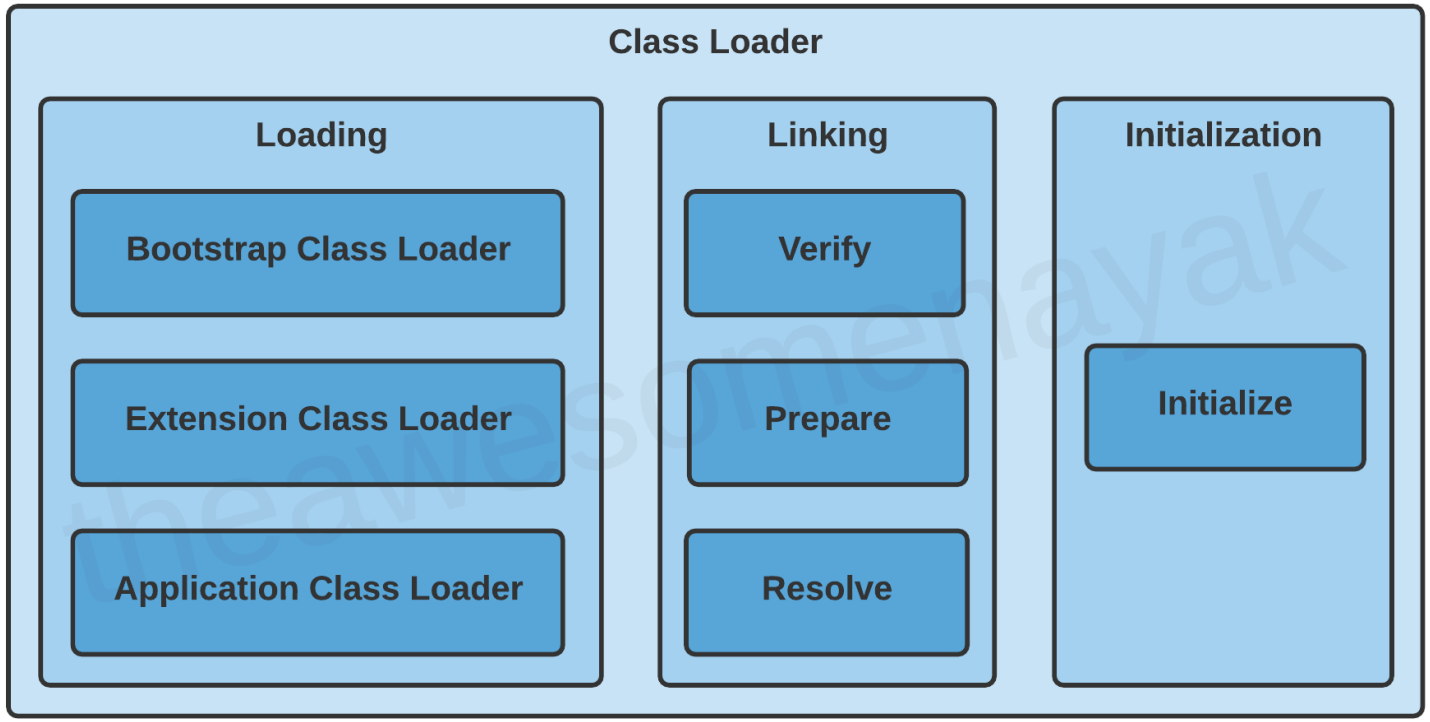
What is a JVM in Java?

- There are three main mechanisms inside the JVM as shown in the above diagram.
- ClassLoader
- Memory Area
- Execution Engine

Next Lecture

Class Loader

- The first class to be loaded into memory is usually the class that contains the main() method.
- There are three phases in the class loading process: loading, linking, and initialization.



Class Loader - Loading

- Bootstrap Class Loader - This is the root class loader. It is the superclass of Extension Class Loader and loads the standard Java packages like java.lang, java.net, java.util, java.io, and so on. These packages are present inside the rt.jar file and other core libraries present in the $JAVA_HOME/jre/lib directory.
- Extension Class Loader - This is the subclass of the Bootstrap Class Loader and the superclass of the Application Class Loader. This loads the extensions of standard Java libraries which are present in the $JAVA_HOME/jre/lib/ext directory.
- Application Class Loader - This is the final class loader and the subclass of Extension Class Loader. It loads the files present on the classpath. By default, the classpath is set to the current directory of the application. The classpath can also be modified by adding the -classpath or -cp command line option.

Class Loader - Loading

- The JVM uses the ClassLoader.loadClass() method for loading the class into memory.
- If a parent class loader is unable to find a class, it delegates the work to a child class loader. If the last child class loader isn't able to load the class either, it throws NoClassDefFoundError or ClassNotFoundException.

Class Loader - Linking - Verification

- After a class is loaded into memory, it undergoes the linking process. Linking a class or interface involves combining the different elements and dependencies of the program together.
- Linking includes the following steps:
- Verification: This phase checks the structural correctness of the .class file by checking it against a set of constraints or rules. If verification fails for some reason, we get a VerifyException.
- For example, if the code has been built using Java 11, but is being run on a system that has Java 8 installed, the verification phase will fail.

Class Loader - Linking - Preparation

- Preparation: After a Java virtual machine has loaded a class and performed whatever verification it chooses to do up front, the class is ready for preparation. During the preparation phase, the Java virtual machine allocates memory for the class variables and sets them to default initial values. The class variables are not initialized to their proper initial values until the initialization phase. (No Java code is executed during the preparation step.) During preparation, the Java virtual machine sets the newly allocated memory for the class variables to a default value determined by the type of the variable.

Class Loader - Linking - Resolution

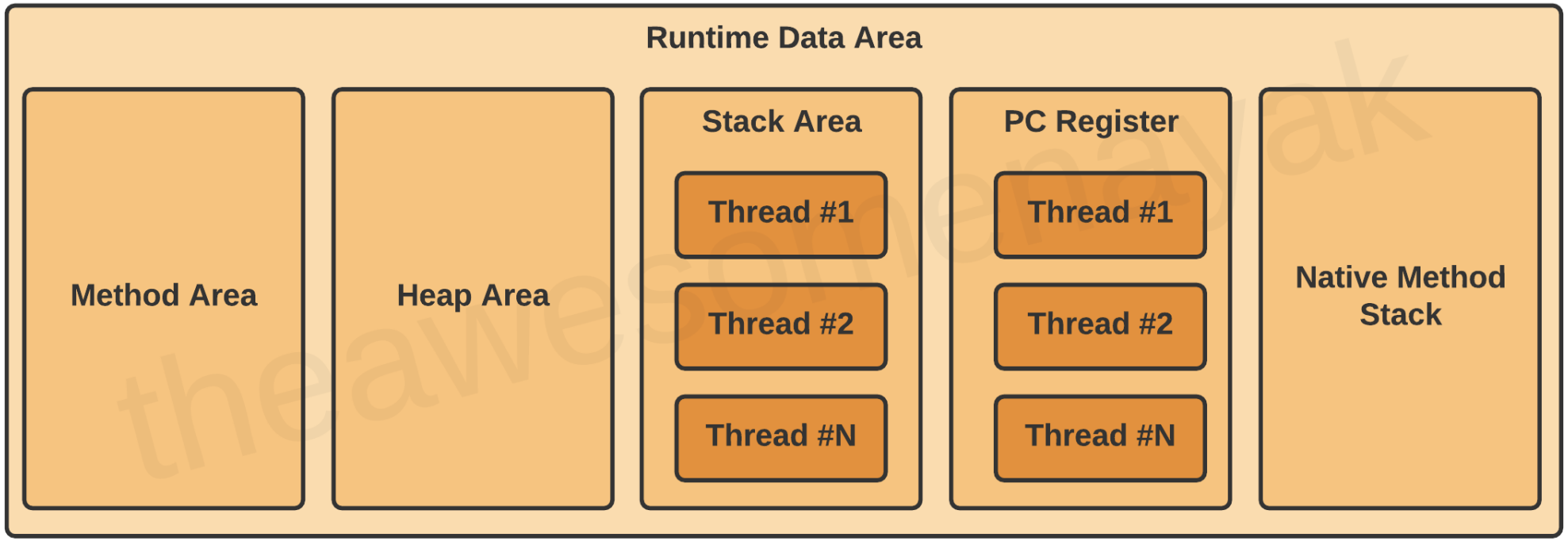
- **Resolution**
- Resolution involves replacing symbolic references in the bytecode with actual references to memory locations. In the class file, references to other classes, methods, or fields are stored symbolically (using names or constants), and during resolution, these are mapped to actual addresses in memory.
- The JVM resolves symbolic references like class names, field names, and method signatures into direct references that point to actual memory addresses.
- It links classes with their superclasses and interfaces.
- The symbolic references are replaced with concrete references, allowing the JVM to call methods and access fields directly.

Class Loader - Initialization

- Initialization involves executing the initialization method of the class or interface (known as <clinit>).
- This can include calling the class's constructor, executing the static block, and assigning values to all the static variables. This is the final stage of class loading.

Runtime Data Area

- There are five components inside the runtime data area:
-



Runtime Data Area - method area

- method area is shared among all Java Virtual Machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in an operating system process. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface initialization.
- In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Runtime Data Area - *heap* area

- The Java Virtual Machine has a *heap* that is shared among all Java Virtual Machine threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated.
- All the objects and their corresponding instance variables are stored here.
- **Note:** Since the Method and Heap areas share the same memory for multiple threads, the data stored here is not thread safe.

Runtime Data Area - Stack area

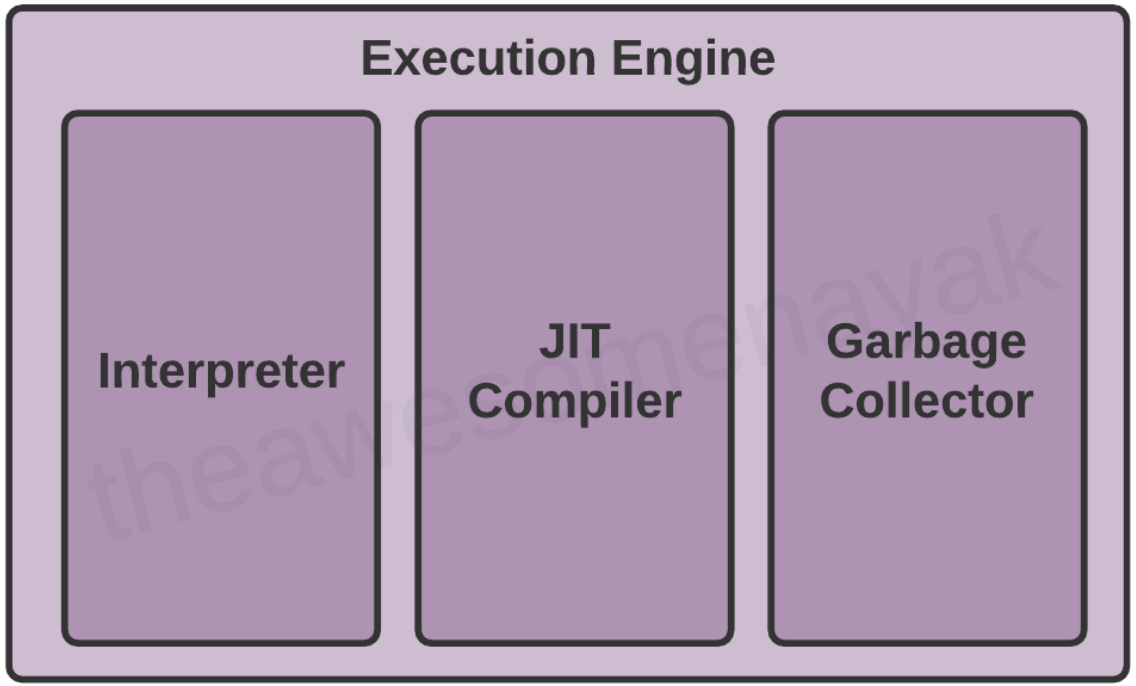
- **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
- **Note:** Since the Stack Area is not shared, it is inherently thread safe.
-

Runtime Data Area

- **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
- **Native method stacks:** The JVM contains stacks that support *native* methods. These methods are written in a language other than the Java, such as C and C++. For every new thread, a separate native method stack is also allocated..

Execution Engine

- Once the bytecode has been loaded into the main memory, and details are available in the runtime data area, the next step is to run the program. The Execution Engine handles this by executing the code present in each class.



Execution Engine

- **Interpreter**
- The interpreter reads and executes the bytecode instructions line by line. Due to the line by line execution, the interpreter is comparatively slower.
- Another disadvantage of the interpreter is that when a method is called multiple times, every time a new interpretation is required.
- **JIT Compiler**
- The JIT Compiler overcomes the disadvantage of the interpreter. The Execution Engine first uses the interpreter to execute the byte code, but when it finds some repeated code, it uses the JIT compiler.
- The JIT compiler then compiles the entire bytecode and changes it to native machine code. This native machine code is used directly for repeated method calls, which improves the performance of the system.

Execution Engine

- The JIT Compiler has the following components:

1. **Intermediate Code Generator -** generates intermediate code
2. **Code Optimizer -** optimizes the intermediate code for better performance
3. **Target Code Generator -** converts intermediate code to native machine code
4. **Profiler -** finds the hotspots (code that is executed repeatedly)

Execution Engine

- An interpreter will fetch the value of sum from memory for each iteration in the loop, add the value of i to it, and write it back to memory. This is a costly operation because it is accessing the memory each time it enters the loop.
- However, the JIT compiler will recognize that this code has a HotSpot, and will perform optimizations on it. It will store a local copy of sum in the PC register for the thread and will keep adding the value of i to it in the loop. Once the loop is complete, it will write the value of sum back to memory.

```
int sum = 10;
for(int i = 0 ; i <= 10; i++) {
    sum += i;
}
System.out.println(sum);
```

Garbage Collector

- Garbage collection makes Java memory efficient because because it removes the unreferenced objects from heap memory and makes free space for new objects. It involves two phases:
  - Mark - in this step, the GC identifies the unused objects in memory
  - Sweep - in this step, the GC removes the objects identified during the previous phase
- Garbage Collections is done automatically by the JVM at regular intervals and does not need to be handled separately. It can also be triggered by calling System.gc(), but the execution is not guaranteed.

Java Native Interface (JNI)

- At times, it is necessary to use native (non-Java) code (for example, C/C++). This can be in cases where we need to interact with hardware, or to overcome the memory management and performance constraints in Java. Java supports the execution of native code via the Java Native Interface (JNI).
- JNI acts as a bridge for permitting the supporting packages for other programming languages such as C, C++, and so on. This is especially helpful in cases where you need to write code that is not entirely supported by Java, like some platform specific features that can only be written in C.
- You can use the native keyword to indicate that the method implementation will be provided by a native library. You will also need to invoke System.loadLibrary() to load the shared native library into memory, and make its functions available to Java.

Native Method Libraries

- Native Method Libraries are libraries that are written in other programming languages, such as C, C++, and assembly.
- These libraries are usually present in the form of .dll or .so files. These native libraries can be loaded through JNI.

References

- Java: How To Program, Early Objects, 11th edition
- Setup IntelliJ IDEA (2021) for JavaFX & SceneBuilder and Create Your First JavaFX Application – YouTube
- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- Java documentation https://docs.oracle.com/javase
- https://www.youtube.com/watch?v=LTgClBqDank&feature=youtu.be
- Understanding Java Virtual Machine (JVM) Architecture | by Jalitha Dewapura | Java For Beginners | Medium
- JVM Tutorial - Java Virtual Machine Architecture Explained for Beginners (freecodecamp.org)
- How JVM Works - JVM Architecture – GeeksforGeeks
- Chapter 2. The Structure of the Java Virtual Machine (oracle.com)
- JVM Internals (jamesdbloom.com)

Usage of native in Java - Stack Overflow

Java Type Loading, Linking, and Initialization (artima.com)

Thanks