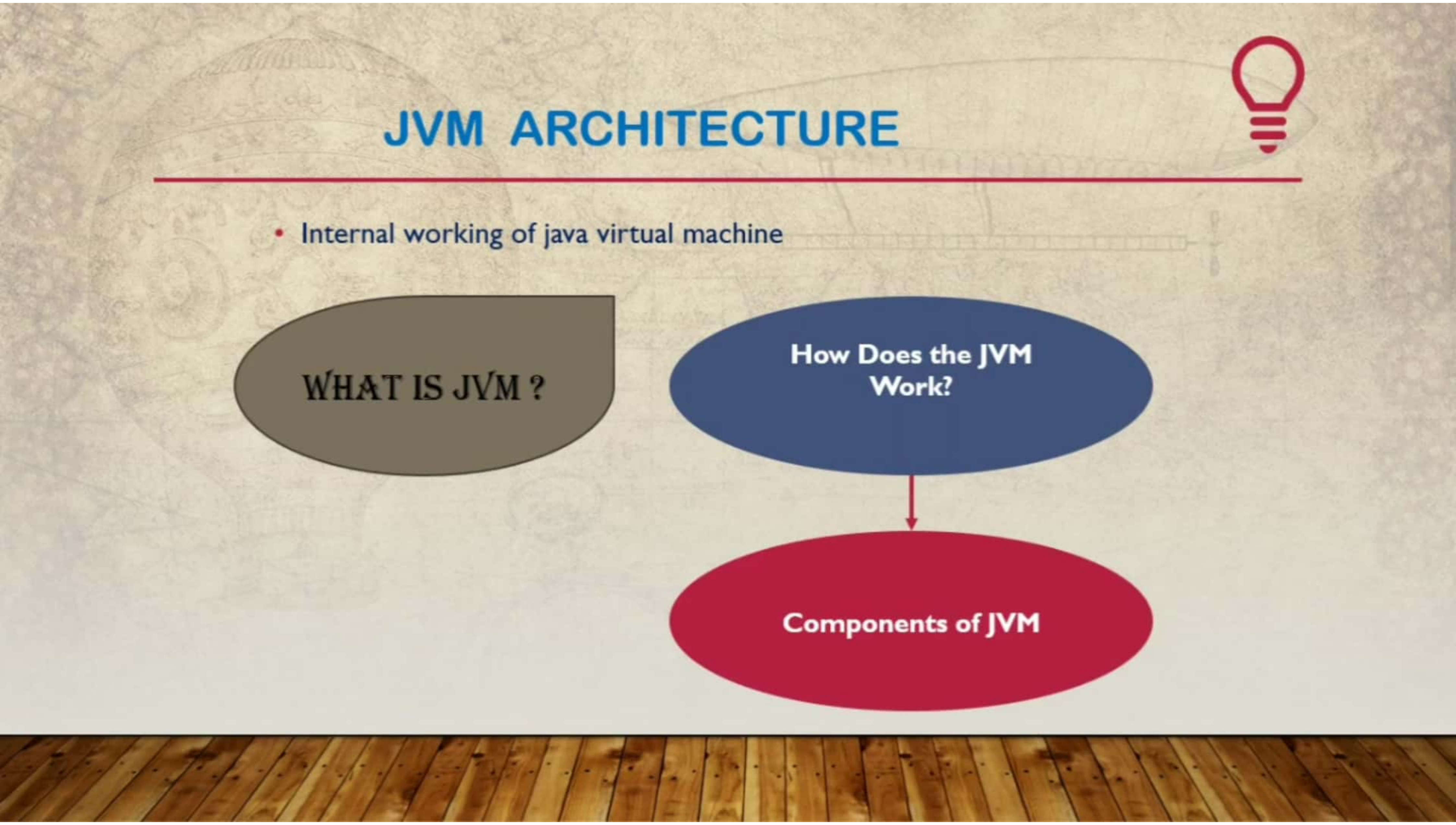


JVM ARCHITECTURE

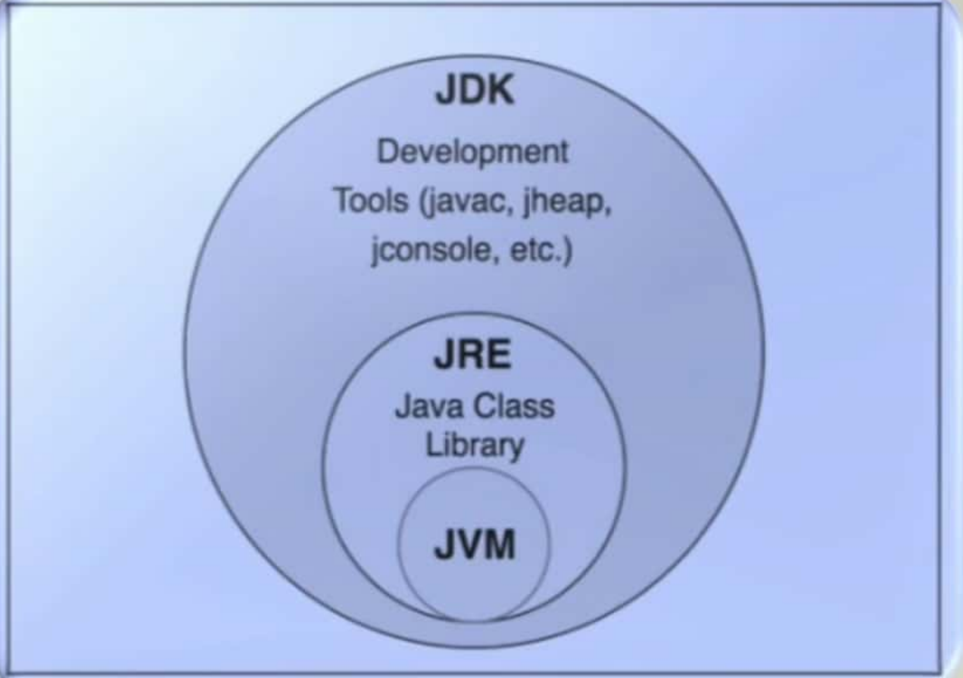
- Internal working of java virtual machine

WHAT IS JVM ?

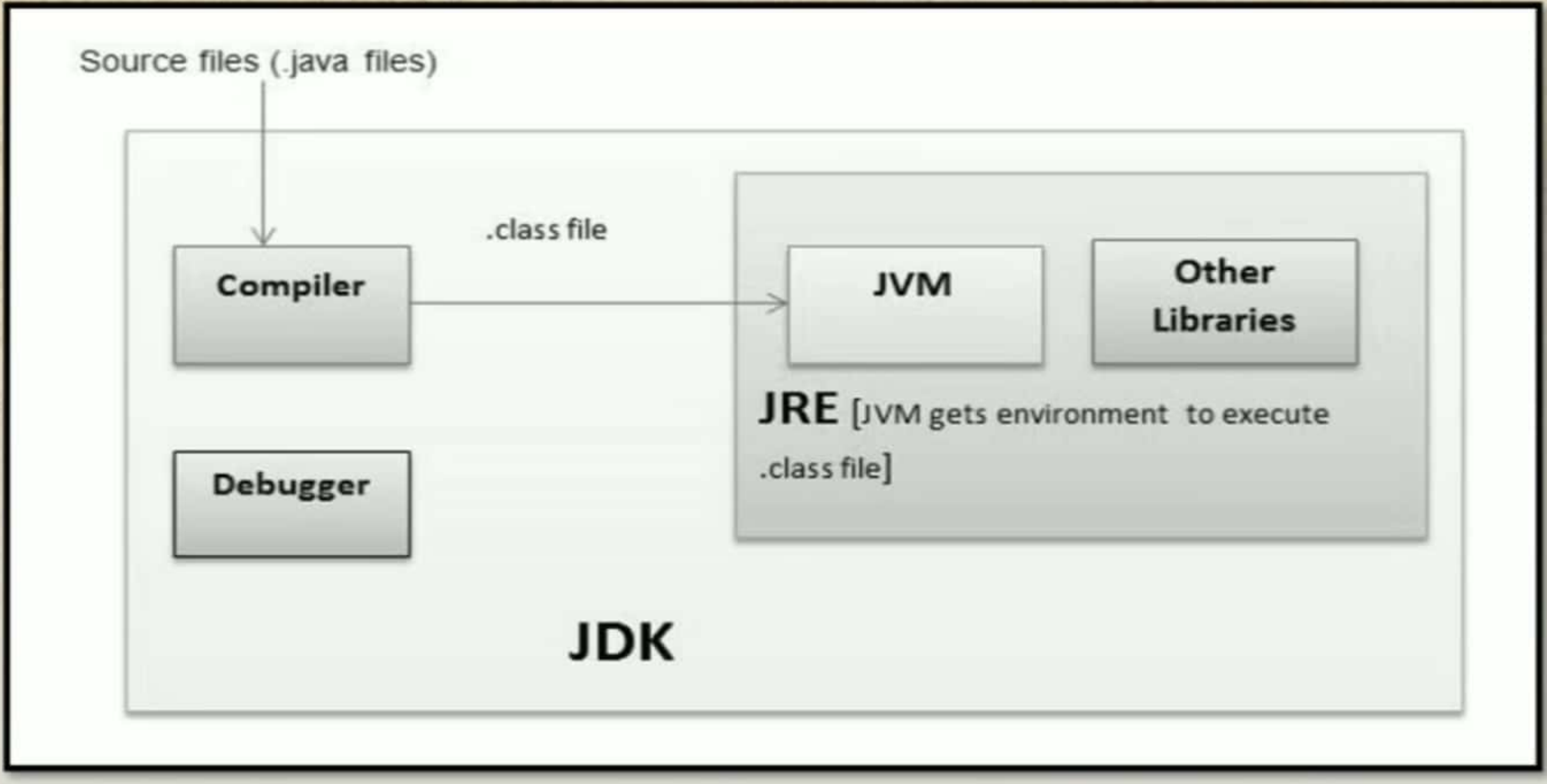
How Does the JVM Work?

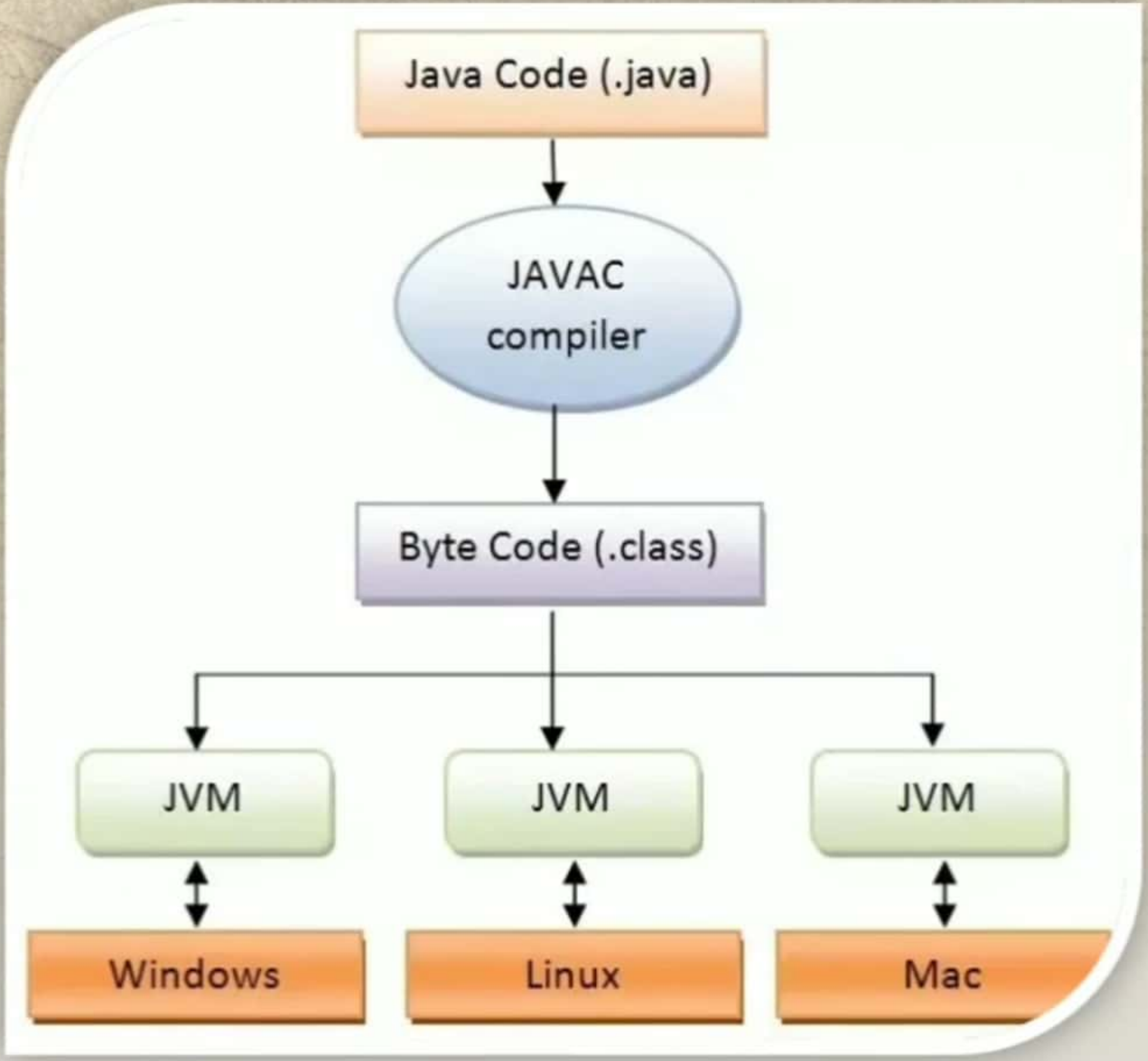
Components of JVM

WHAT EXACTLY IS JVM ?



JVM , JRE, JDK RELATION


Source files (.java files)
Compiler
.class file
JVM
Other Libraries
JRE [JVM gets environment to execute .class file]
Debugger
JDK

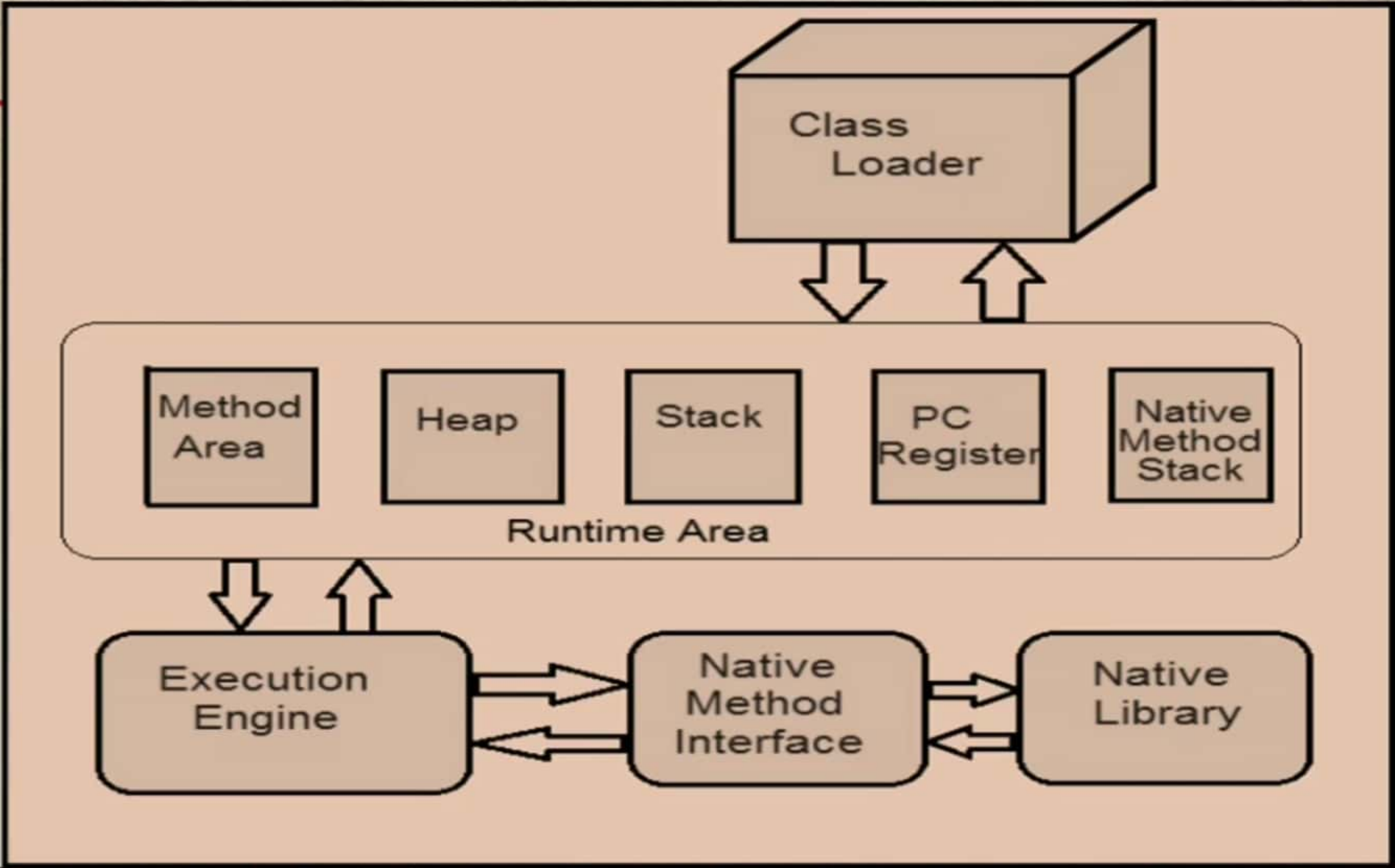

Java Code (.java)
JAVAC compiler
Byte Code (.class)
JVM
JVM
JVM
Windows
Linux
Mac

JVM Components

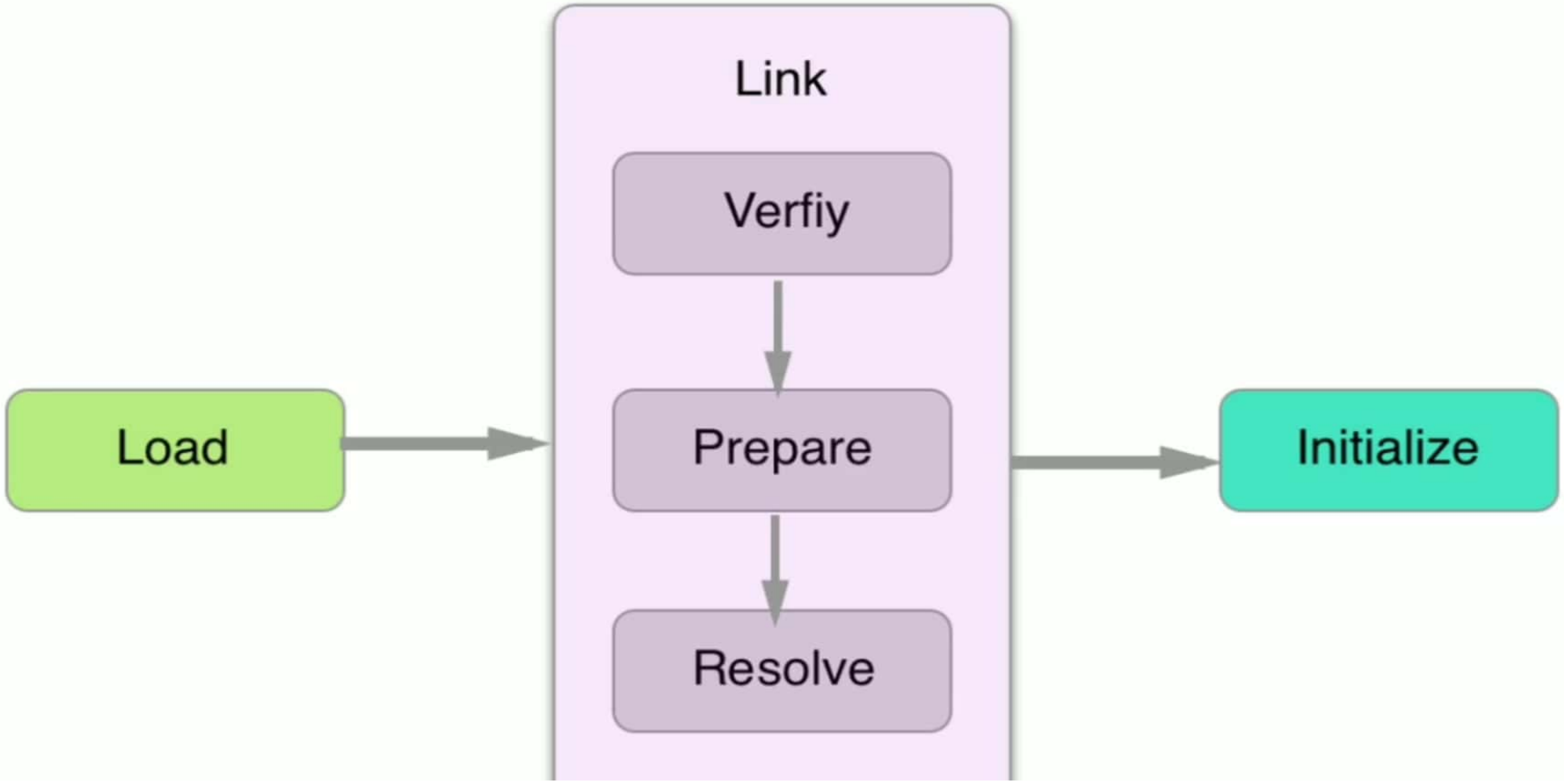
1.) Class Loader

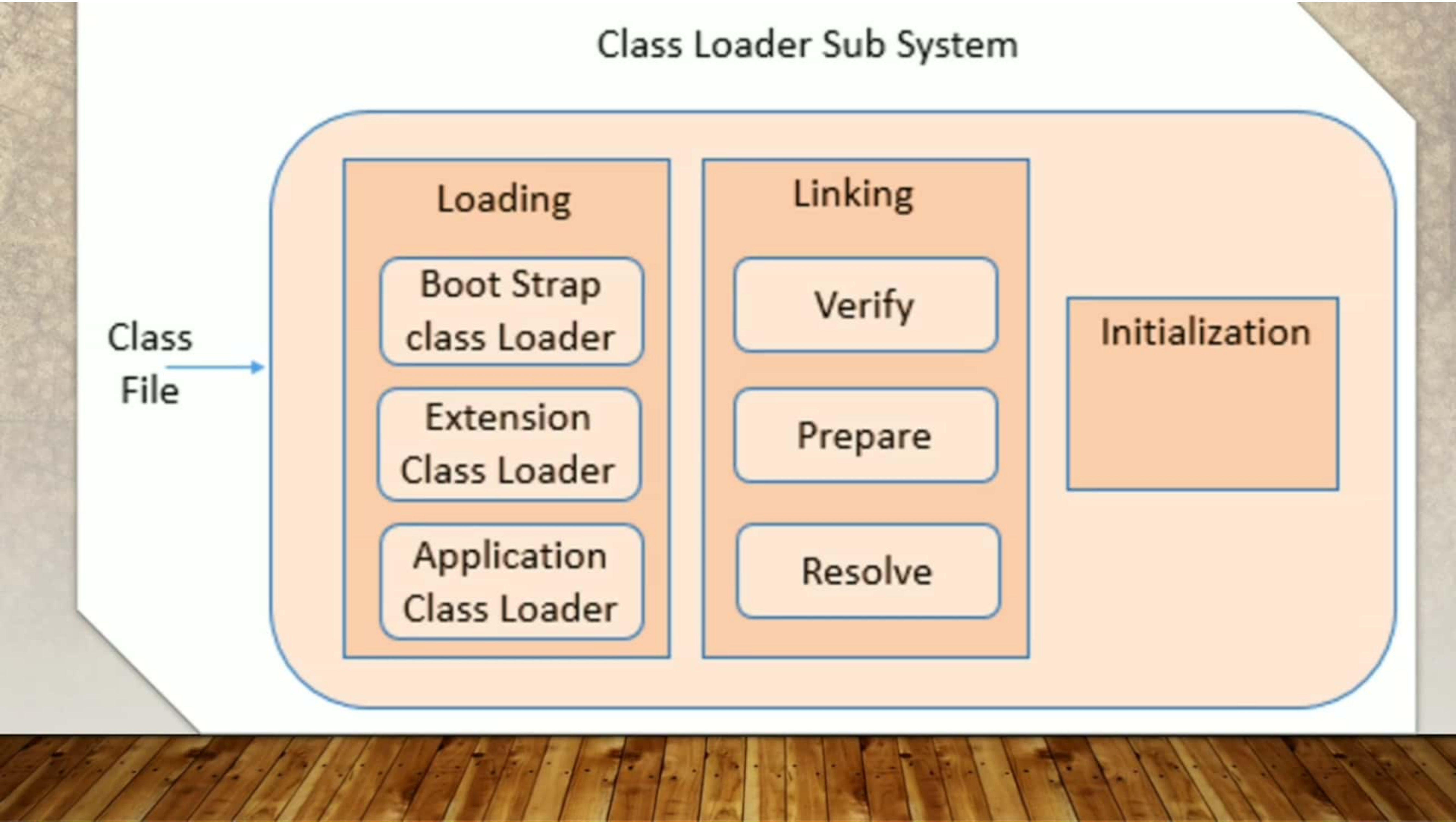
2.) Runtime Data Areas

3.) Execution Engine

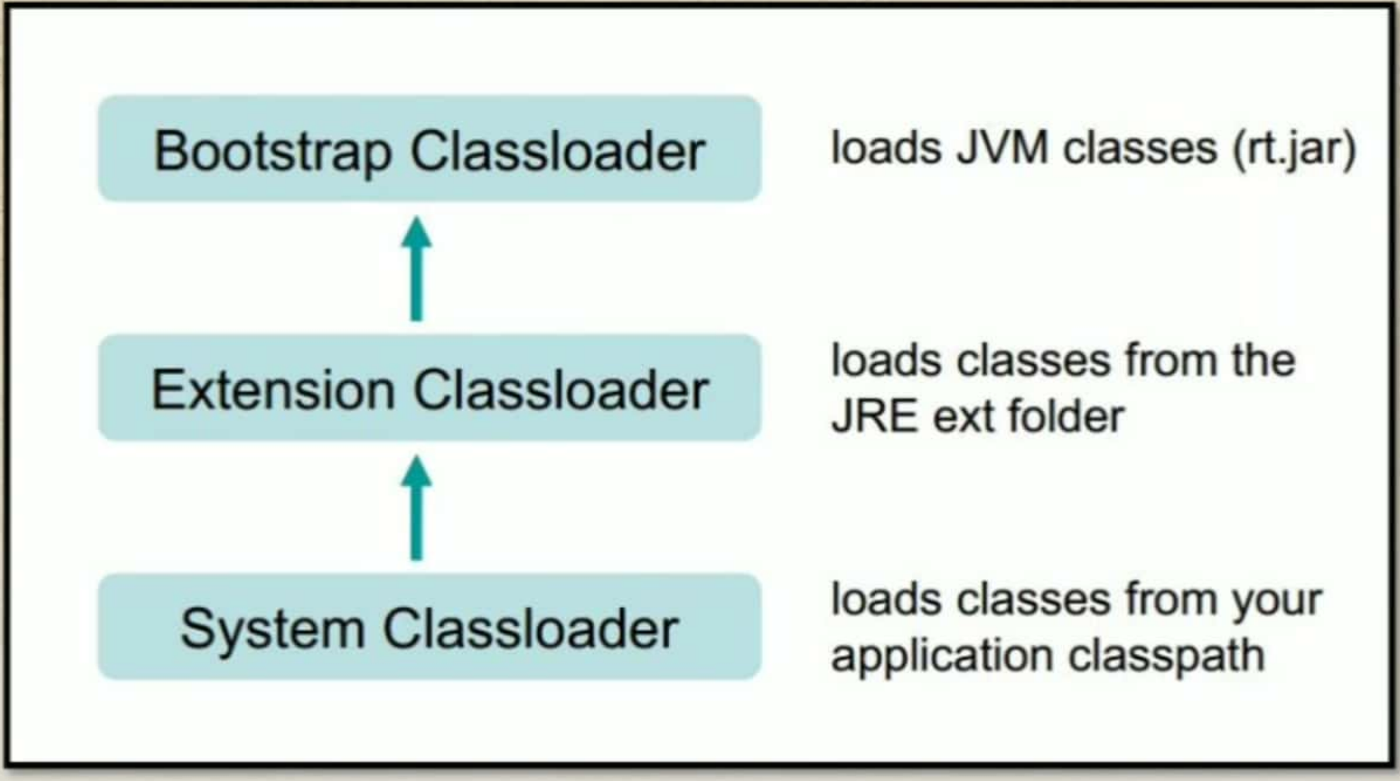


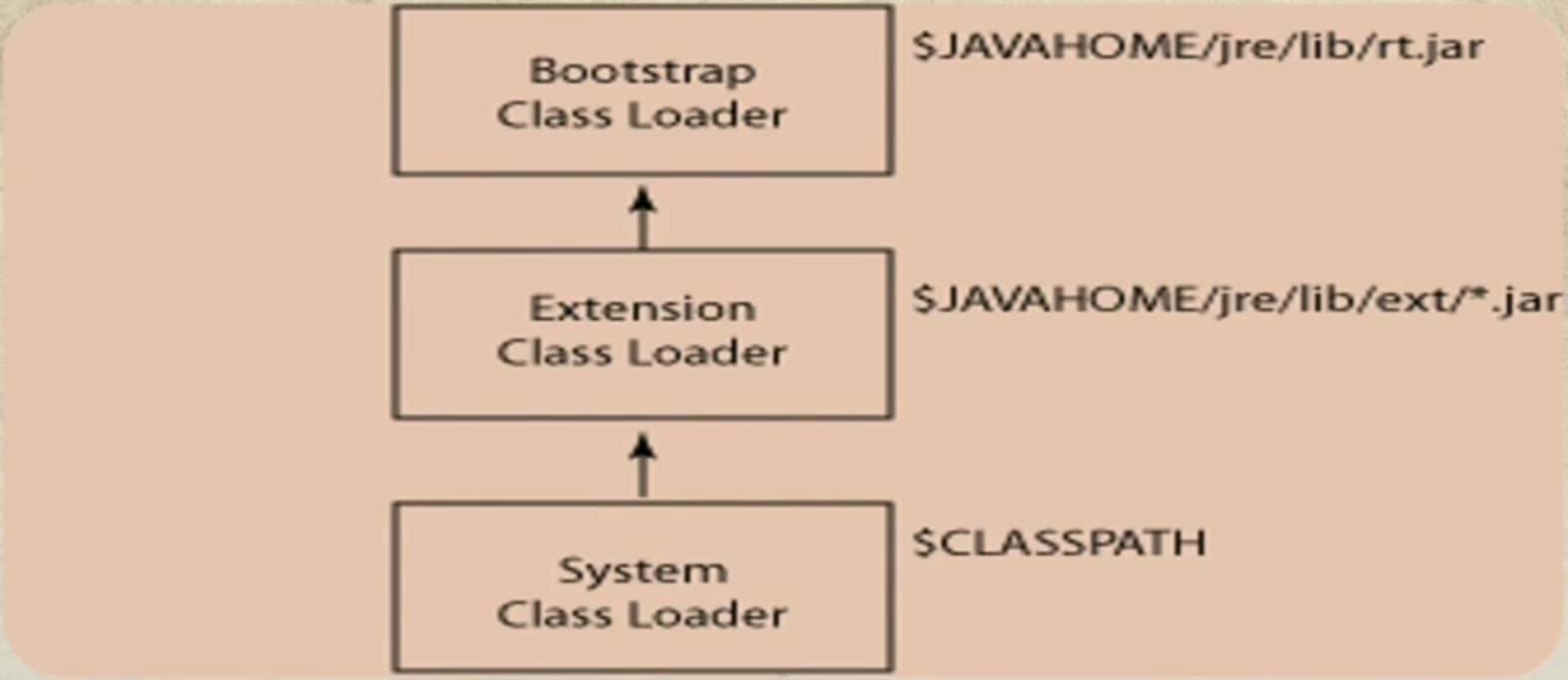
1.) Class Loader Subsystem



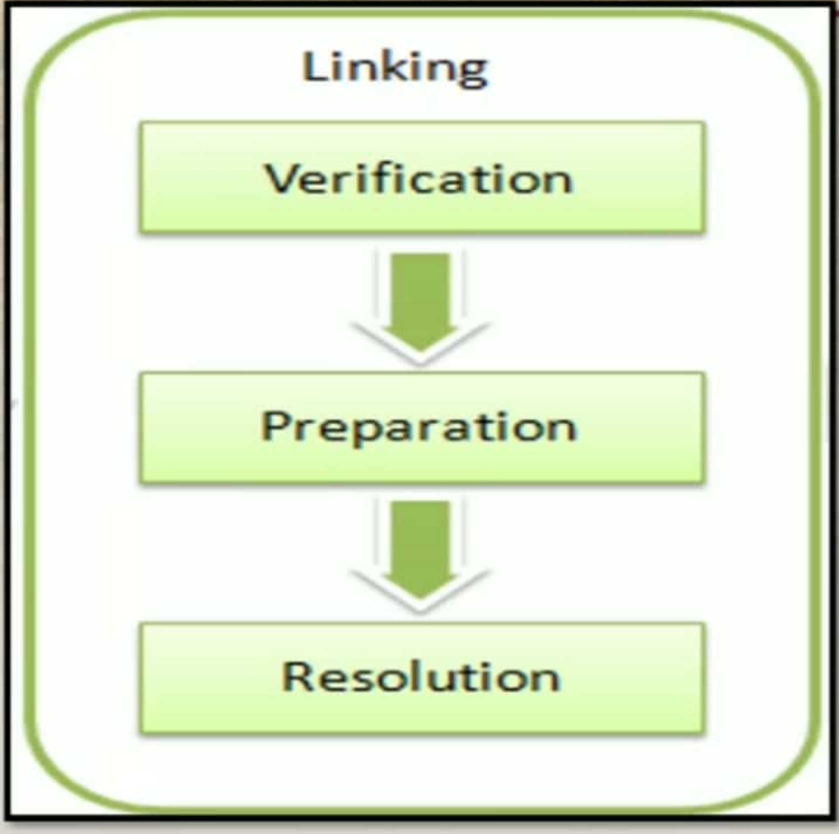

Class Loader Sub System
Class File
Loading
Boot Strap class Loader
Extension Class Loader
Application Class Loader
Linking
Verify
Prepare
Resolve
Initialization

Class Loader Subsystem -> 1.) a.) Loading Phase


Bootstrap Classloader
loads JVM classes (rt.jar)
Extension Classloader
loads classes from the JRE ext folder
System Classloader
loads classes from your application classpath


Bootstrap Class Loader
$JAVAHOME/jre/lib/rt.jar
Extension Class Loader
$JAVAHOME/jre/lib/ext/*.jar
System Class Loader
$CLASSPATH

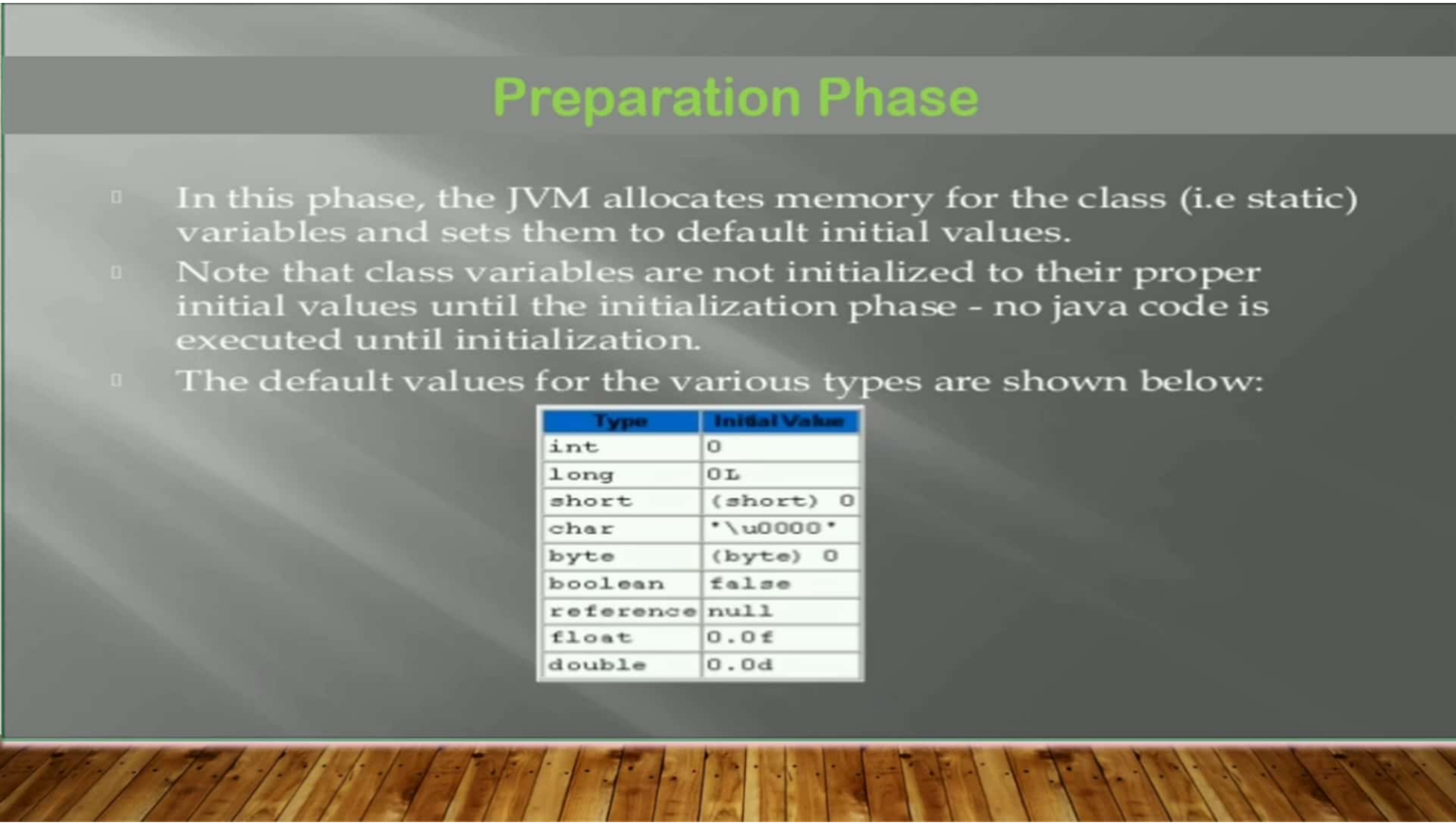
Class Loader Subsystem -> 1.) b.) Linking Phase



- linking
  - verifying -> verifies bytecode correctness
  - preparing -> allocates memory
  - resolving -> links with classes, interfaces, fields, methods

Linking : Verification

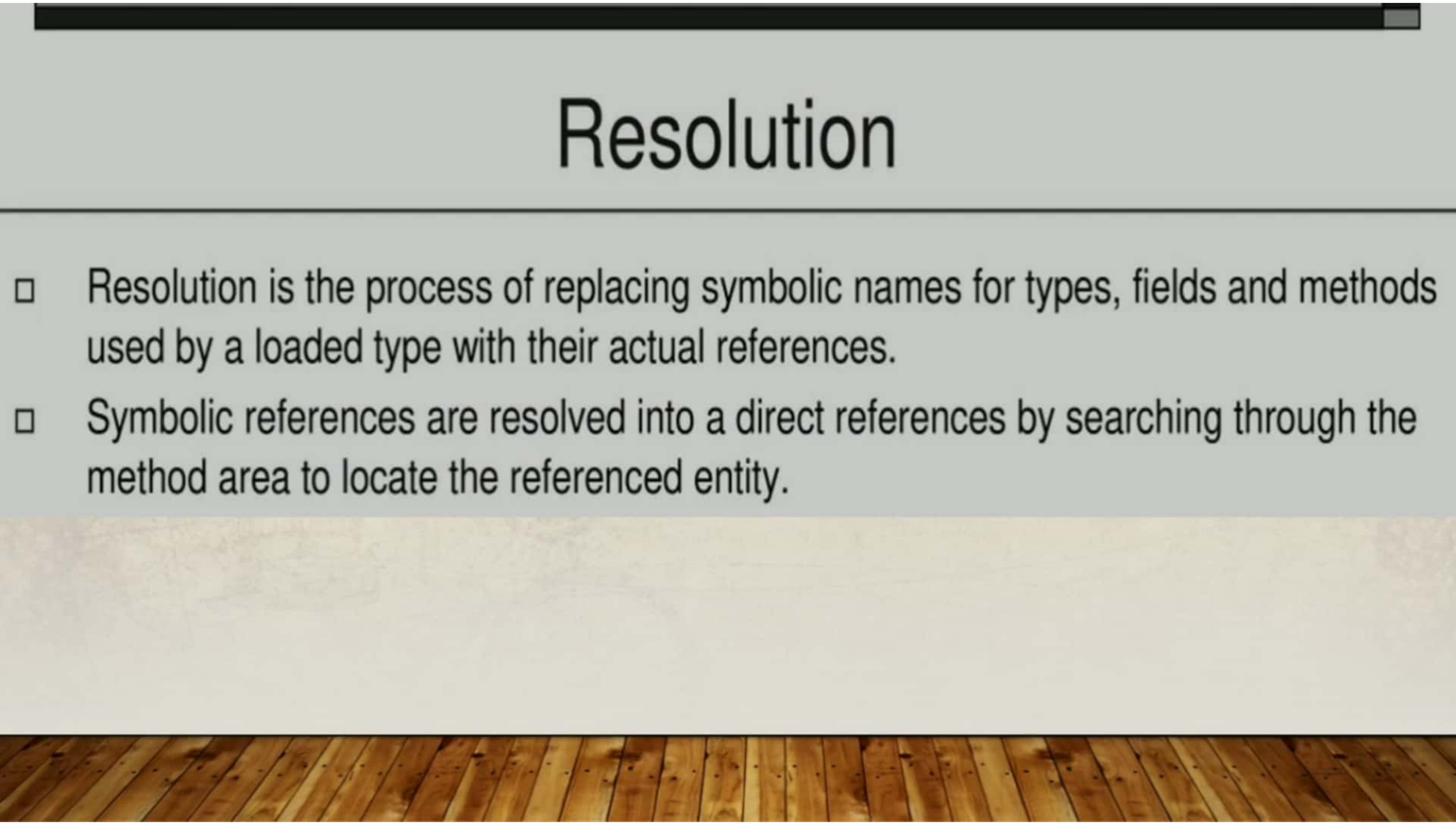
- The next process handled by the class loader is Linking. This involves three sub-processes: Verification, Preparation and Resolution.
- Verification is the process of ensuring that binary representation of a class is structurally correct.
- The JVM has to make sure that a file it is asked to load was generated by a valid compiler and it is well formed.
- Class B may be a valid sub-class of A at the time A and B were compiled, but class A may have been changed and re-compiled.
- Example of some of the things that are checked at verification are:
  - Every method is provided with a structurally correct signature.
  - Every instruction obeys the type discipline of the Java language
  - Every branch instruction branches to the start not middle of another instruction.



Preparation Phase

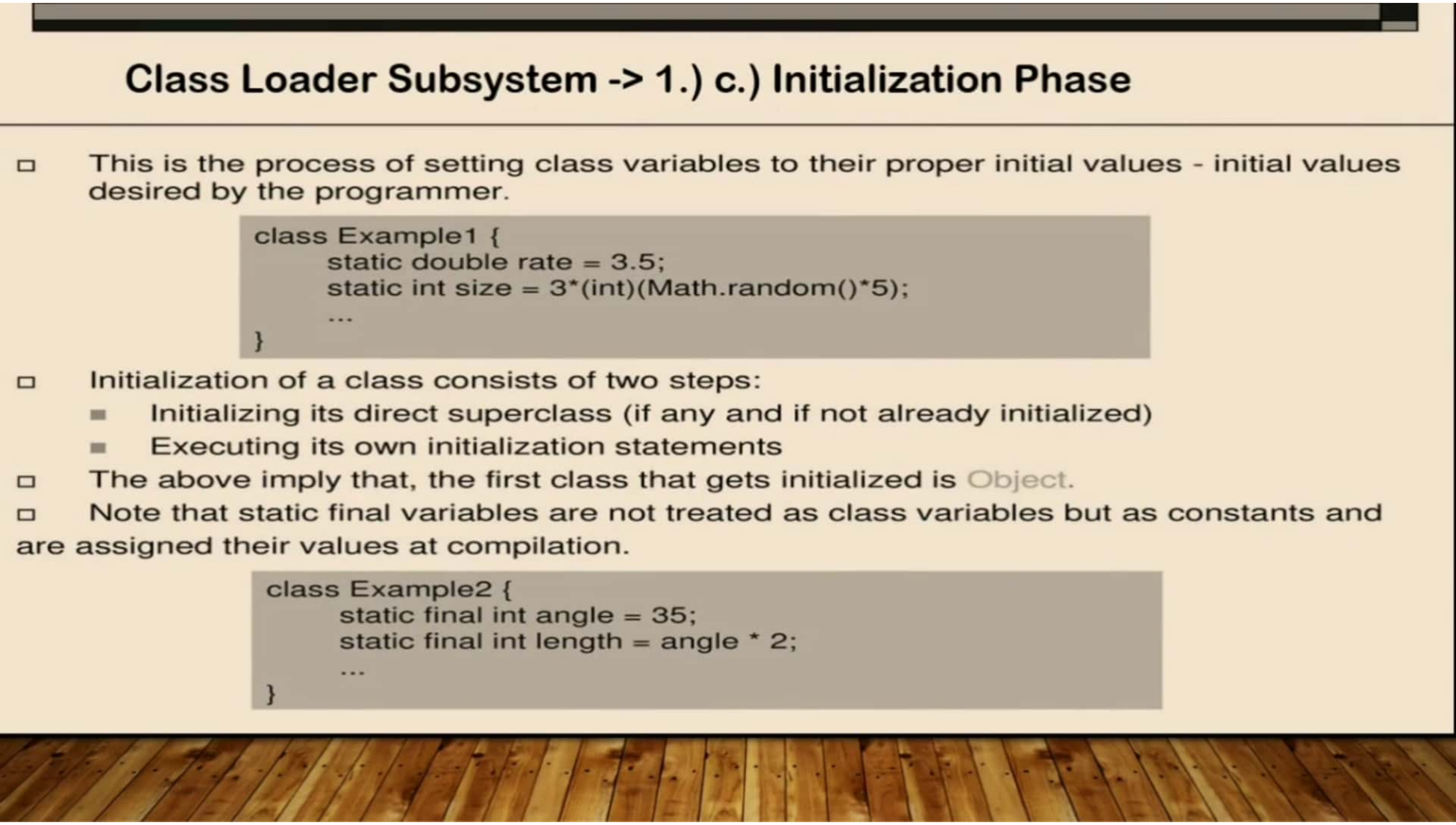
- In this phase, the JVM allocates memory for the class (i.e static) variables and sets them to default initial values.
- Note that class variables are not initialized to their proper initial values until the initialization phase - no java code is executed until initialization.
- The default values for the various types are shown below:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d



Resolution

- Resolution is the process of replacing symbolic names for types, fields and methods used by a loaded type with their actual references.
- Symbolic references are resolved into a direct references by searching through the method area to locate the referenced entity.



Class Loader Subsystem -> 1.) c.) Initialization Phase

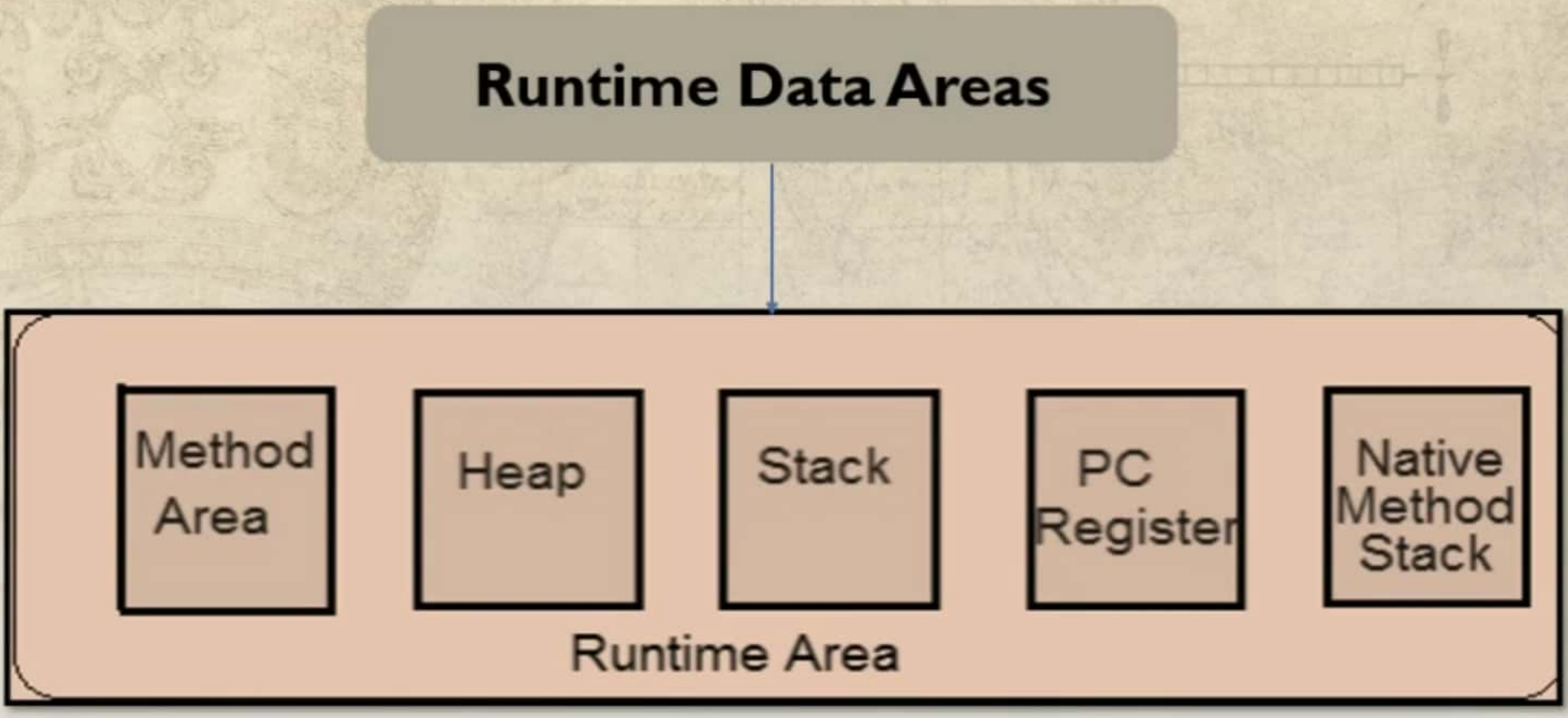
- This is the process of setting class variables to their proper initial values - initial values desired by the programmer.

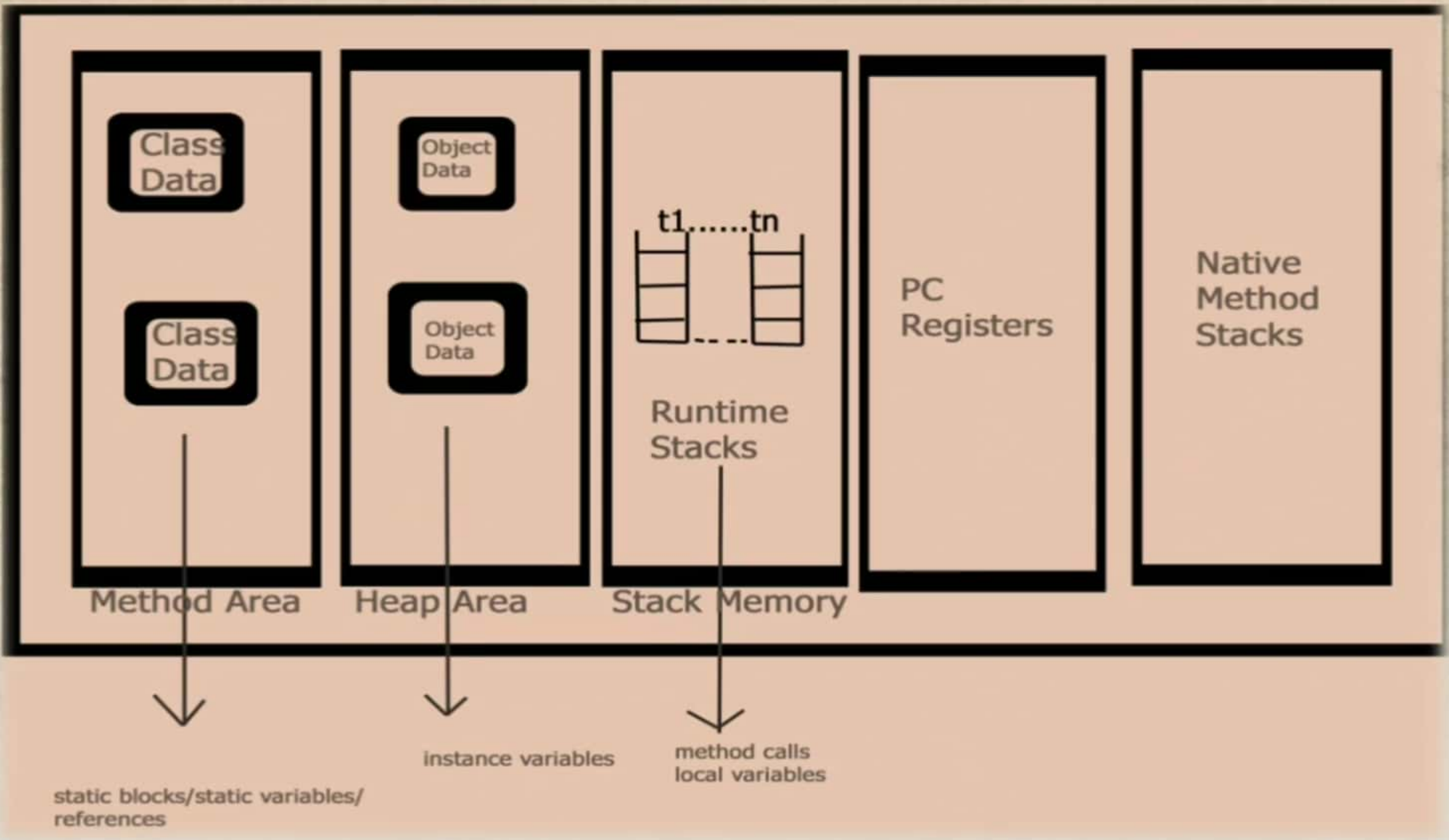
```
class Example1 {
        static double rate = 3.5;
        static int size = 3*(int)(Math.random()*5);
        ...
}
```

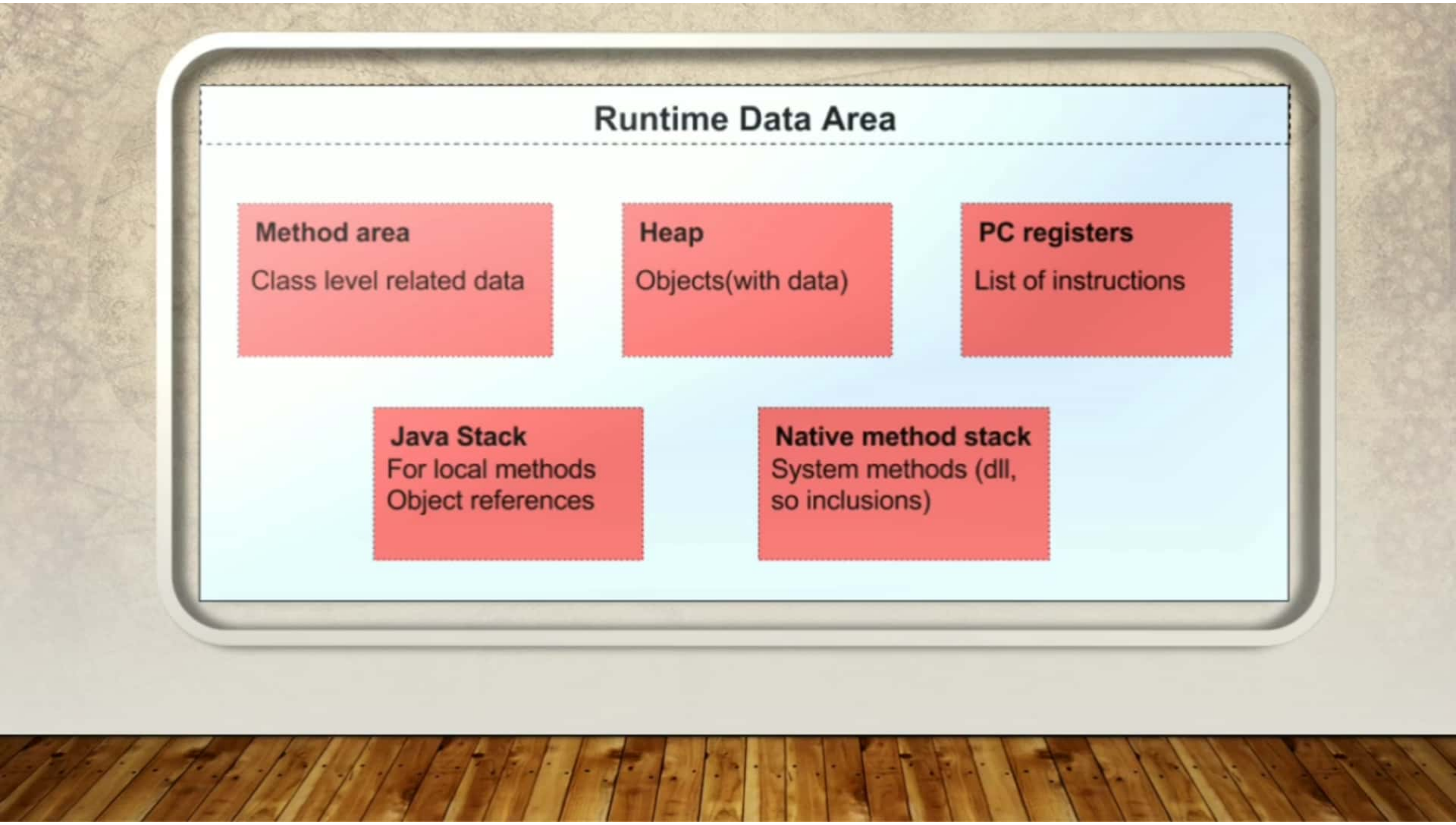
- Initialization of a class consists of two steps:
  - Initializing its direct superclass (if any and if not already initialized)
  - Executing its own initialization statements
- The above imply that, the first class that gets initialized is Object.
- Note that static final variables are not treated as class variables but as constants and are assigned their values at compilation.

```
class Example2 {
        static final int angle = 35;
        static final int length = angle * 2;
        ...
}
```

2.) Runtime Data Areas




Class Data
Class Data
Object Data
Object Data
t1......tn
Runtime Stacks
PC Registers
Native Method Stacks
Method Area
Heap Area
Stack Memory
instance variables
method calls
local variables
static blocks/static variables/
references



Runtime Data Area

Method area
Class level related data

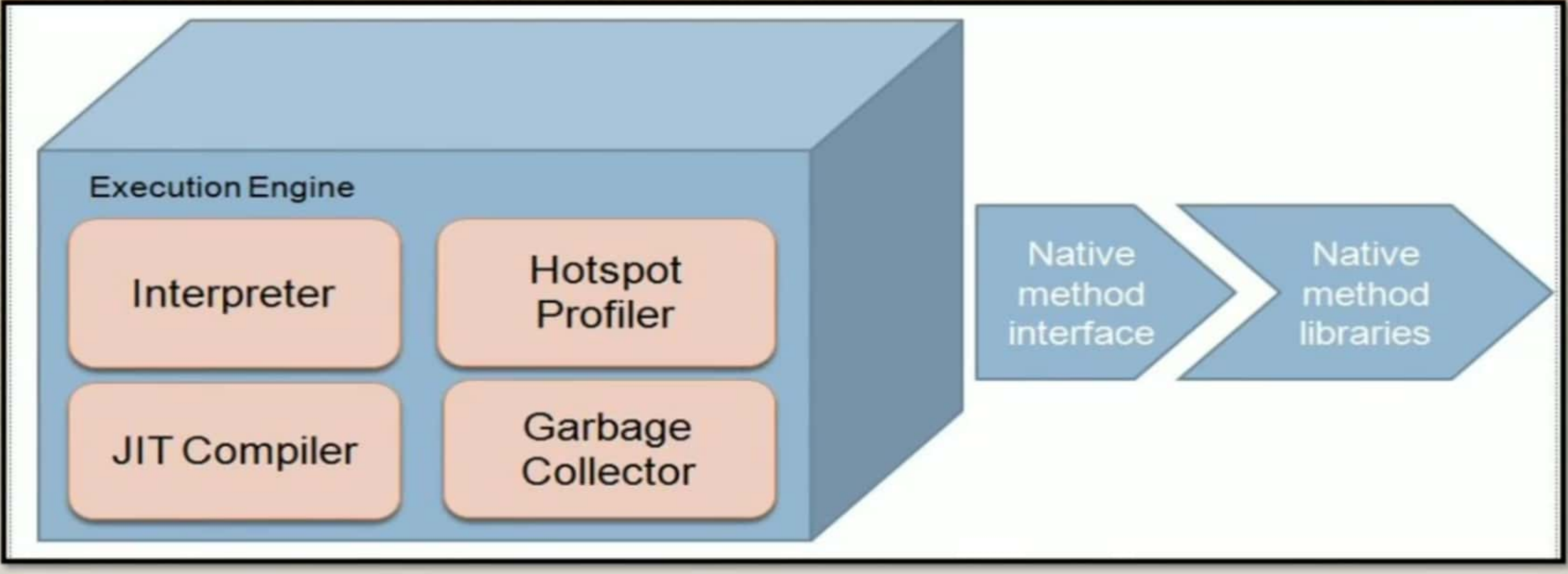
Heap
Objects(with data)

PC registers
List of instructions

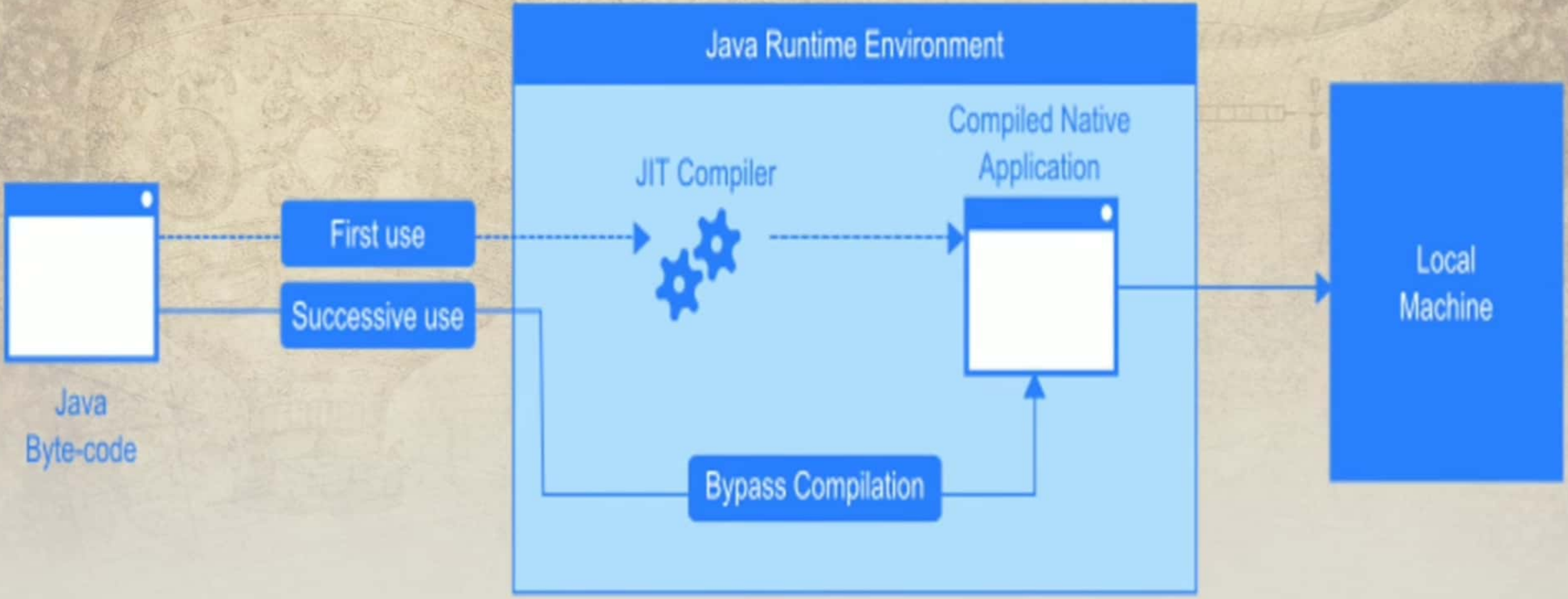
Java Stack
For local methods
Object references

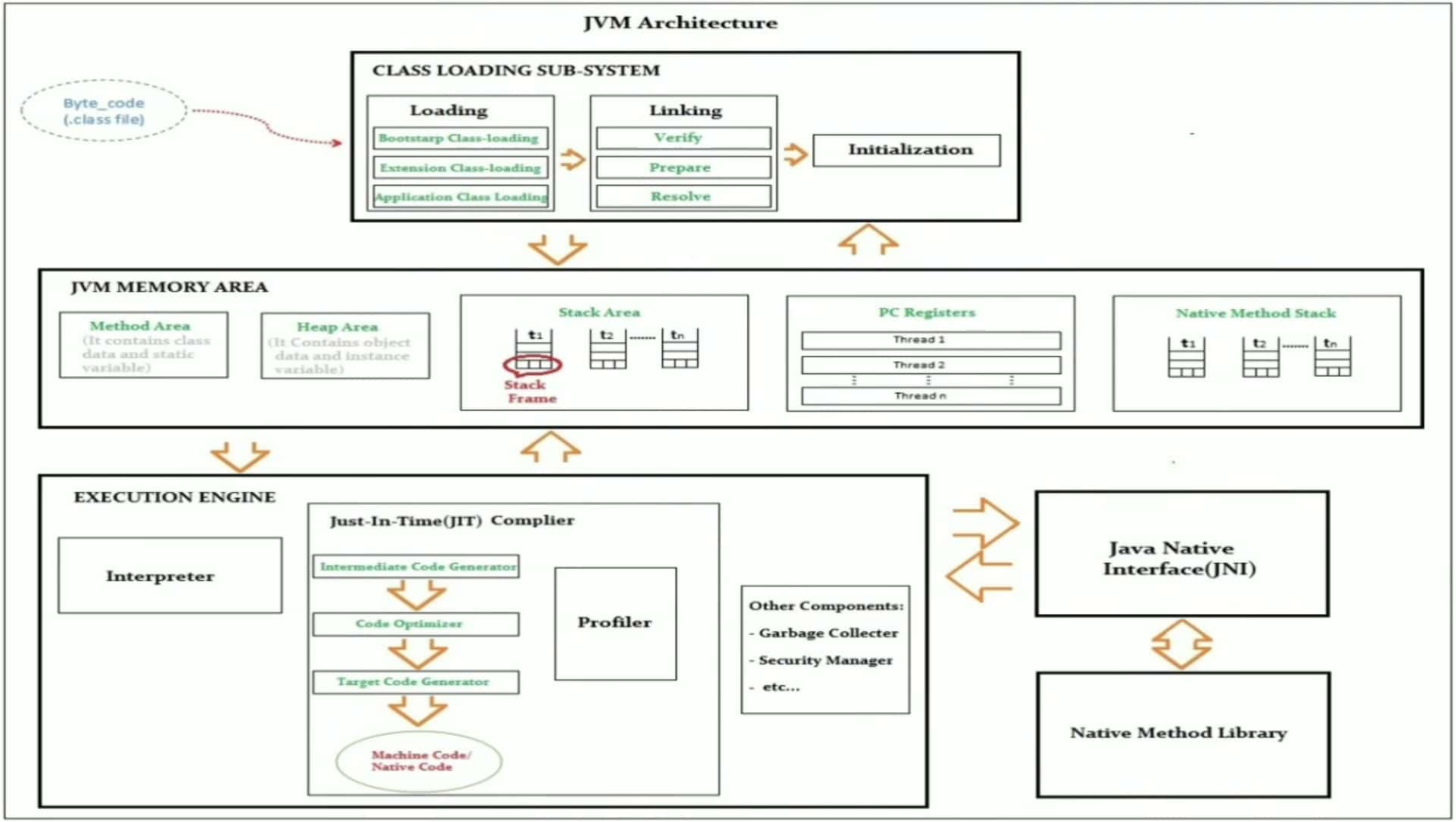
Native method stack
System methods (dll, so inclusions)

3.) EXECUTION ENGINE



Use of Just-in time Compiler [JIT]




JVM Architecture
Byte_code (.class file)
CLASS LOADING SUB-SYSTEM
Loading
Bootstarp Class-loading
Extension Class-loading
Application Class Loading
Linking
Verify
Prepare
Resolve
Initialization
JVM MEMORY AREA
Method Area (It contains class data and static variable)
Heap Area (It Contains object data and instance variable)
Stack Area
t1
t2
tn
Stack Frame
PC Registers
Thread 1
Thread 2
Thread n
Native Method Stack
t1
t2
tn
EXECUTION ENGINE
Interpreter
Just-In-Time(JIT) Complier
Intermediate Code Generator
Code Optimizer
Target Code Generator
Machine Code/ Native Code
Profiler
Other Components:
- Garbage Collecter
- Security Manager
- etc...
Java Native Interface(JNI)
Native Method Library