Object Oriented Programming

Java Virtual Machine

Sarsuna College
Affiliated to
Calcutta University

Krishna Daripa
Asst. Professor & Head
Computer Science Department
Sarsuna College

JDK- Java Development Kit

- Contains the basic tools and libraries necessary for creating and executing Java programs.
- JDK is a ***free software*** from Java Soft, a division of Sun Microsystems now part of Oracle.
- Contains a library of standard classes and a collection of utilities for building, testing, and documenting Java programs.
- The core Java Application Programming Interface (API) is the aforementioned library of prefabricated classes.

7 main programs in JDK

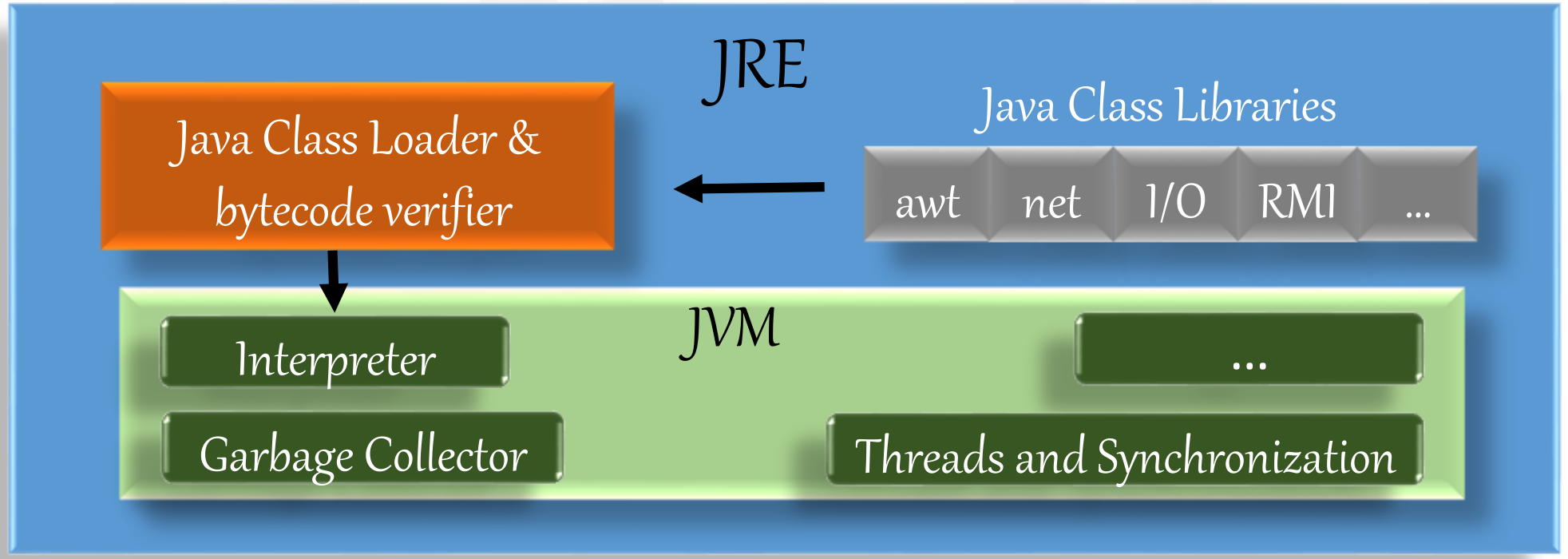
1. **javac**: The *Java compiler*. This program compiles Java source codes into byte codes.
2. **java**: The *Java interpreter*. This program runs Java byte codes.
3. **javadoc**: Generates *API documentation* in HTML format from Java source code.
4. **appletviewer**: A Java interpreter that executes *Java applet* (a special kind of Java Program) classes.
5. **jdb**: The *Java debugger*. Helps us find and fix bugs in Java programs.
6. **javap**: The *Java disassembler*. Displays the accessible functions and data in a compiled class file. It also displays the meaning of the byte codes.
7. **javah**: Creates *C header files* that can be used to make C routines, that can call Java routines,or make C routines that can be called by Java programs.

JVM – Java Virtual Machine

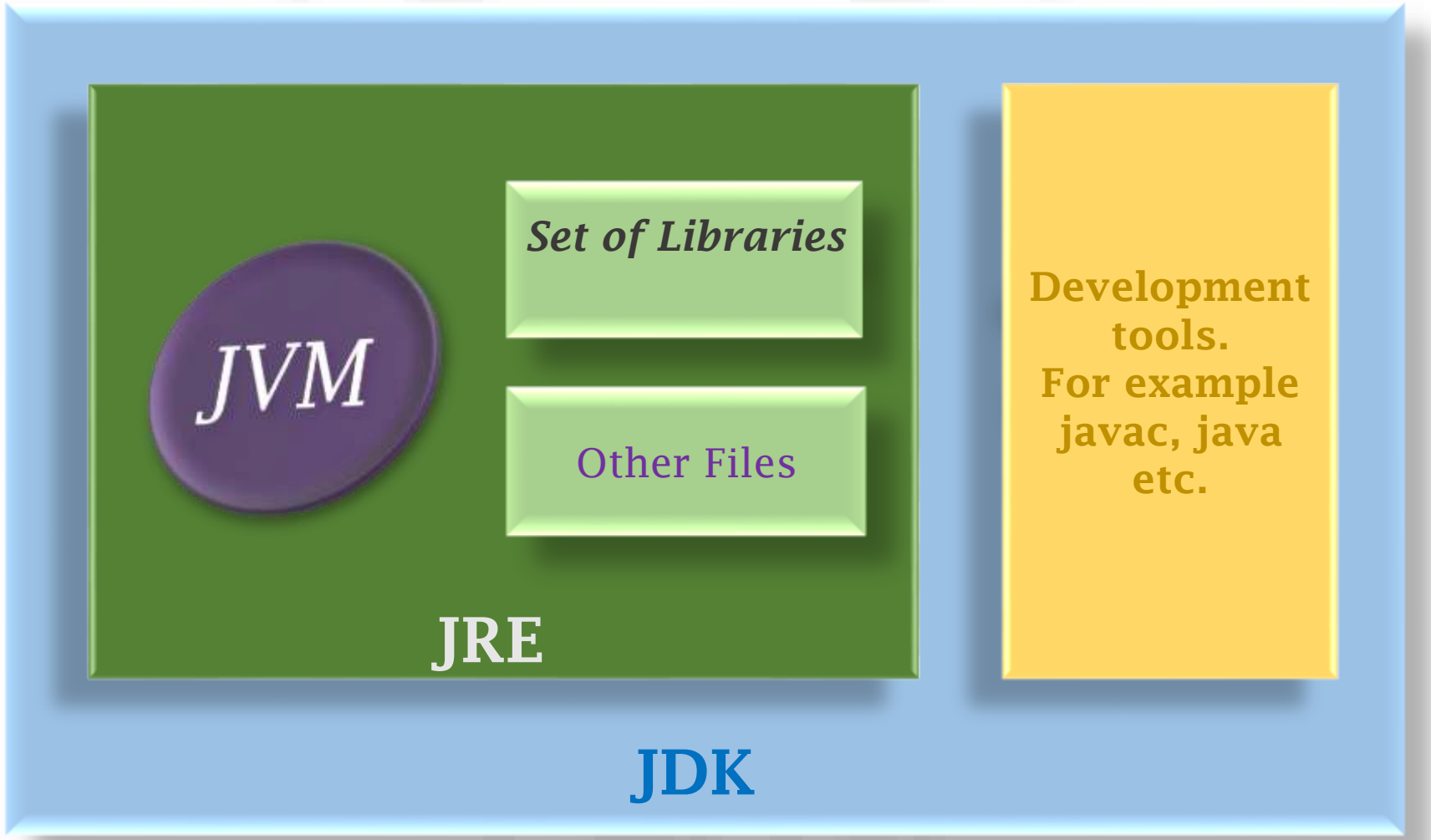
- Java run time system, an ***abstract*** *computing machine*.
- A specification that provides a runtime environment in which Java *bytecode* can be executed.
- JVM helps Java to solve both the security and the portability problems.
- Like a real computing machine, it has *an instruction set* and manipulates various *memory areas* at run time.
- Whenever you write *java* command on the command prompt to run the java class, an instance of JVM is created.
- Oracle's current implementations emulate the JVM on mobile, desktop and server devices, but the JVM does not assume any particular implementation technology, host hardware, or host operating system.

JRE - Java Runtime Environment

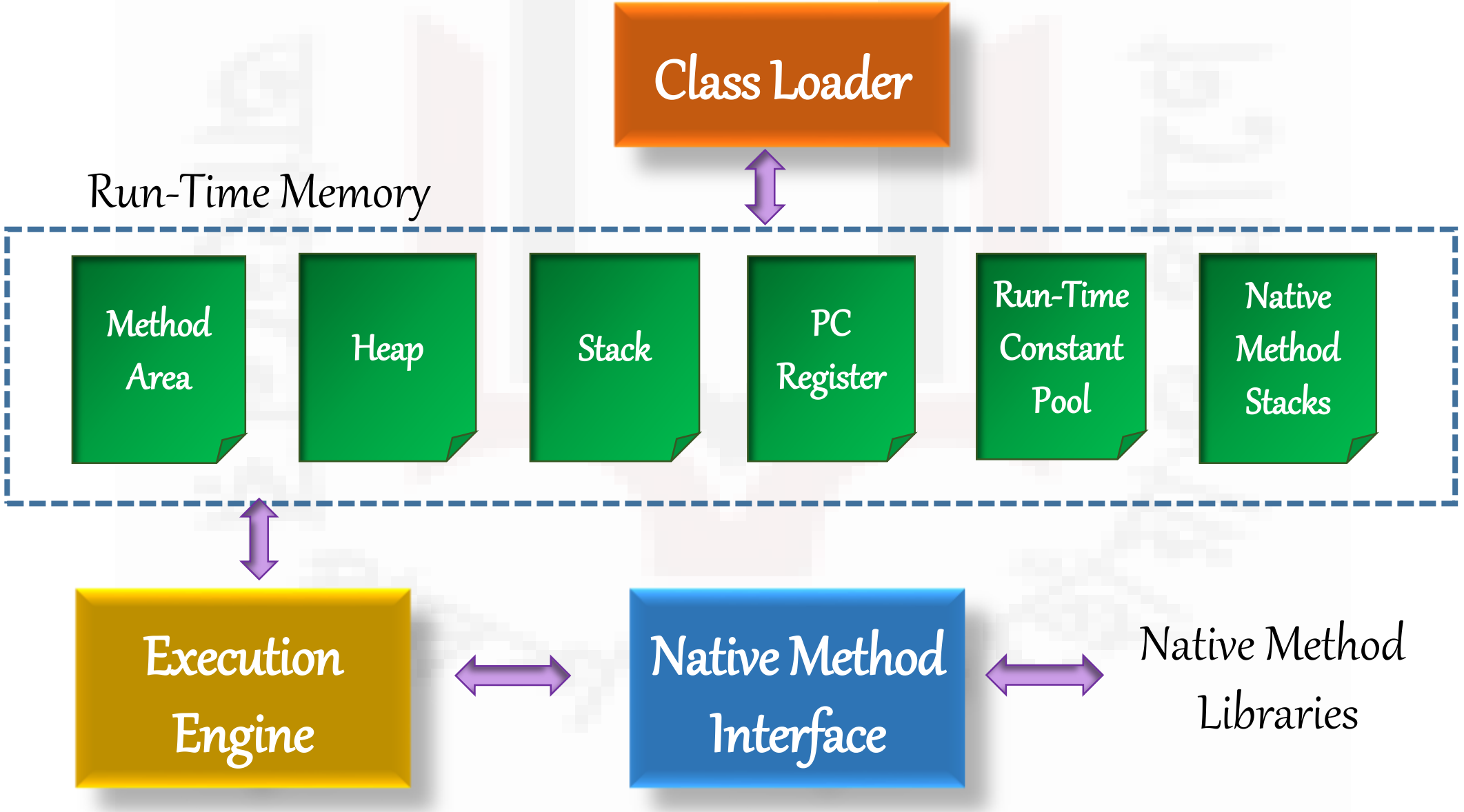
- Provides the runtime environment.
- Implementation of JVM
- It contains a set of libraries and other files that JVM uses at runtime.



JDK, JRE & JVM

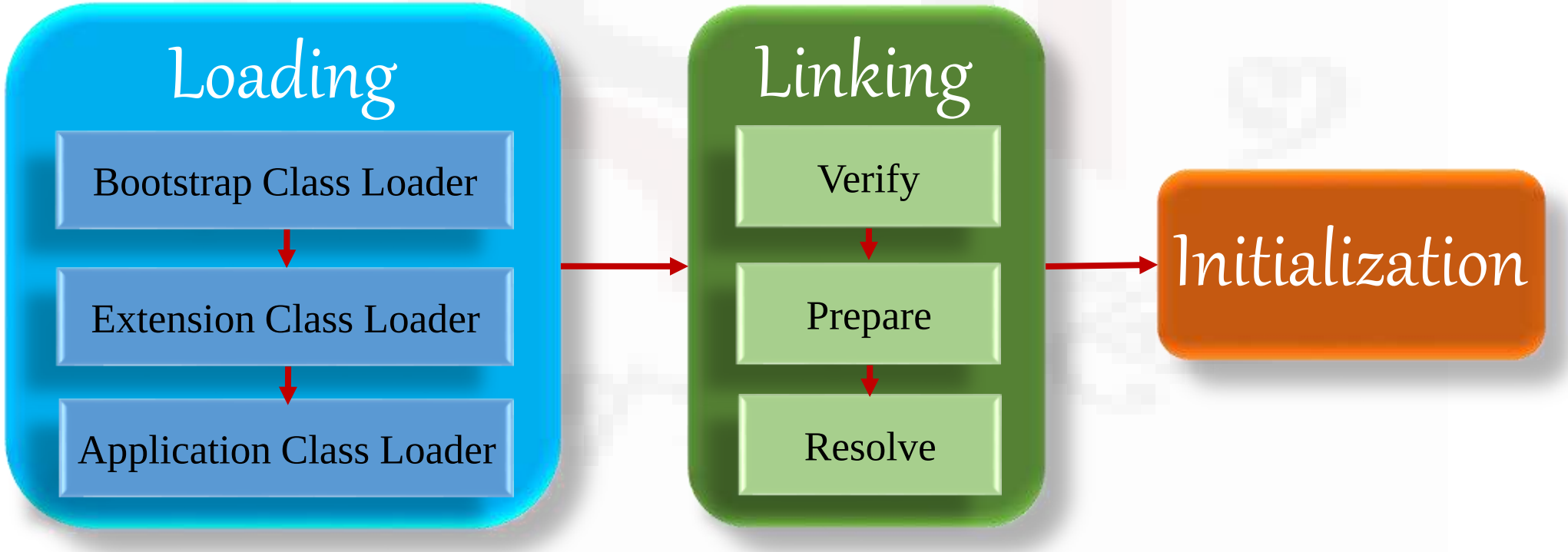

JVM
Set of Libraries
Other Files
JRE
Development tools.
For example javac, java etc.
JDK

JVM Architecture



Java Class loader Subsystem

- Used to load class files.
- Linking and Initialization.
- Whenever we run the java program, it is loaded first by the class loader.



Class loader

- Loading is the process of finding the binary representation of a class or interface type with a particular name and *creating* a class or interface from that binary representation.

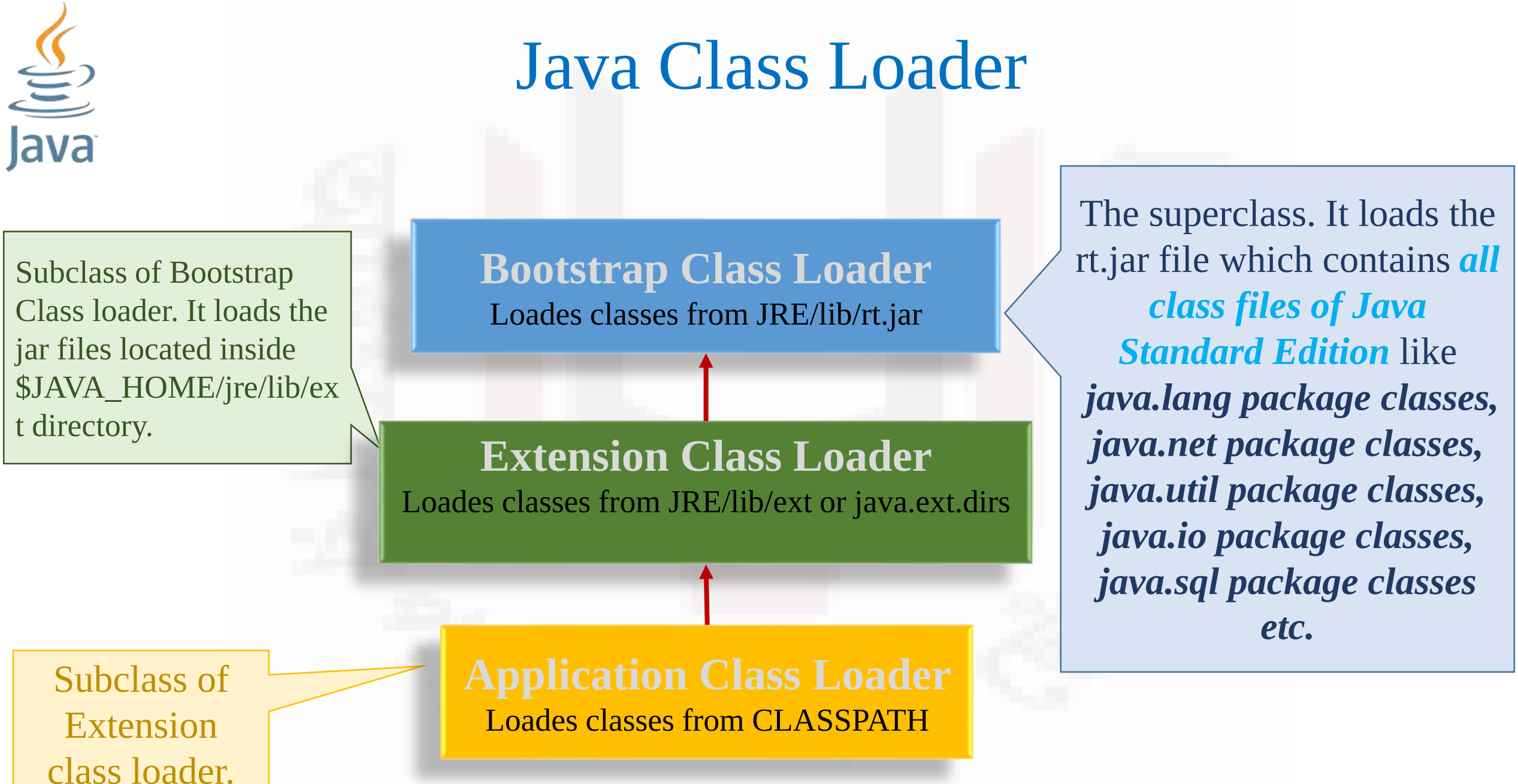
- Creation of a class or interface consists of the construction in the *method area of the JVM* of an implementation-specific internal representation.

- There are two kinds of class loaders:
  1. The **bootstrap class loader** supplied by the JVM.
  2. User-defined class loaders.

User Defined Class loader

- Every user-defined class loader is an instance of a subclass of the abstract class *ClassLoader*.
- Applications employ user-defined class loaders in order to extend the manner in which the JVM dynamically loads and thereby creates classes.
- User-defined class loaders can be used to create classes that originate from user-defined sources.
- For example, a class could be downloaded across a network, generated on the fly, or extracted from an encrypted file.

Java Class Loader



Linking

- Linking is the process of taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed.
- Linking a class or interface involves verifying and preparing that class/interface, its direct superclass, its direct super-interfaces, etc.
- Resolution of symbolic references in the class or interface is an optional part of linking.

Linking

Verify

- Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

Prepare

- For all class variables (static variables) memory will be allocated and assigned with default values.

Resolve

- All symbolic memory references are replaced with the original references from Method Area.

Linking: Verification

- Verification ensures that the binary representation of a class or interface is structurally correct or not.
- Verification may cause additional classes and interfaces to be loaded but need not cause them to be verified or prepared.
- If the binary representation of a class or interface does not satisfy the static or structural constraints, then a *VerifyError* must be thrown at that point.

Linking: Preparation

- Preparation involves creating the static fields for a class or interface and initializing such fields to their default values.
- This does not require the execution of any JVM code; explicit initializers for static fields are executed as part of initialization, not preparation.
- Preparation may occur at any time following creation but must be completed prior to initialization.

Linking: Resolution

- All symbolic memory references are replaced with the original references from Method Area.
- Resolution is the process of dynamically determining concrete values from symbolic references in the run-time constant pool.
- If an error occurs during resolution of a symbolic reference, then an instance of *IncompatibleClassChangeError* (or a subclass) must be thrown at the point.

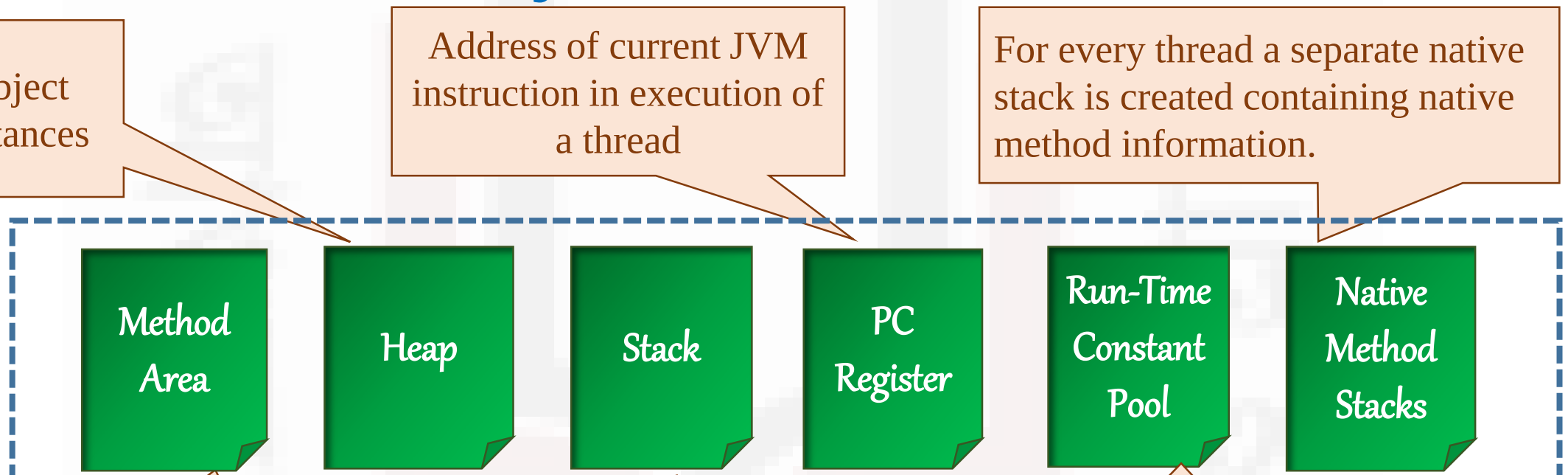
Initialization

- Initialization of a class or interface consists of executing its class or interface initialization method.
- Prior to initialization, a class or interface must be linked, that is, verified, prepared, and optionally resolved.
- The implementation of the JVM is responsible for taking care of synchronization and recursive initialization.
- All static variables will be assigned with the original values, and the static block will be executed.

JVM Memory: Run-Time Data Areas

- Like a real computing machine, JVM has an instruction set and manipulates various memory areas at run time.
- The Java Virtual Machine defines various run-time data areas that are used during execution of a program.
- Some of these data areas are created on Java Virtual Machine start-up and are destroyed only when the Java Virtual Machine exits.
- Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

JVM Memory or Runtime Data Areas



JVM Runtime Data Areas

Method Area:

- It is shared among all Java Virtual Machine threads. There is only one method area per JVM
- Stores per-class structures or meta data such as the runtime constant pool, data field and method data, the code for methods.
- The method area is created on virtual machine start-up.

Heap Area:

- All the Objects, their related instance variables are stored in the heap.
- This memory is common and shared across multiple threads.
- The heap is created on virtual machine start-up.
- Heap storage for objects is reclaimed by an automatic storage management system, known as a garbage collector. Objects are never explicitly deallocated.

JVM Runtime Data Areas

Stack:

- For every thread, JVM creates one private run-time stack at the same time when the thread is created.
- Every block of this stack is called frame which store methods calls.
- A new frame is created each time a method is invoked. It holds local variables and partial results, and plays a part in method invocation and return.
- A frame is destroyed when its method invocation completes.

PC Registers:

- JVM can support many threads of execution at once.
- Each JVM thread has its own PC register.
- Store address of current JVM instruction in execution of a thread.

JVM Runtime Data Areas

Run Time Constant Pool:

- A per-class or per-interface run-time representation of the constant_pool table in a class file.
- Contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at run-time.
- The run-time constant pool serves a function similar to that of a symbol table with a wider range.

Native Method Stack:

- For every thread, separate native stack is created.
- It stores native method information.

Execution Engine

- Execution engine executes the bytecode.
- It is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks.
- Initially, the JVM was designed as an interpreter for bytecode.
- Generally, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code.
- However, with Java, the difference between the two is not so great.
- Because bytecode has been highly optimized.
- To boost the performance Sun provides *Just-In-Time (JIT) Compiler* for Bytecode.

Parts of Execution Engine

1. **Interpreter** : It interprets the bytecode line by line and then executes it.
2. **Just-In-Time Compiler(JIT)** : It is used to increase efficiency of interpreter. Whenever the interpreter sees repeated method calls, it compiles the entire bytecode and changes it to native code. JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
3. **Garbage Collector** : It destroys un-referenced objects.

Java Native Interface & Native Method Library

- **Java Native Interface (JNI) :**

  It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

- **Native Method Libraries :**

  It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Garbage Collector

- A program that manages memory by automatically deleting unreferenced objects.
- Garbage Collection is the process of reclaiming the runtime unused memory automatically.
- In Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
- Garbage collector is best example of Daemon thread as it is always running in background.

Automatic Garbage Collection

- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An object in use, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused object, or unreferenced object, is no longer referenced by any part of your program.
- So the memory used by an unreferenced object can be reclaimed.

Garbage Collector

How can a object be unreferenced?

Some techniques:

1. By nulling the reference

```
Employee e=new Employee();
e=null;
```

2. By assigning a reference to another

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;
```

3. By anonymous object

```
new Employee();
```

Garbage Collection

- finalize() method :
  - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
  - The finalize method of class *Object* performs no special action; it simply returns.
  - Invoked each time before the object is garbage collected.
  - Subclasses of *Object* may override this method.
  - A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.
  - This method is defined in *java.lang.**Object***

    protected void finalize(){}

Garbage Collection

➢Explicit Garbage Collection by gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in ***System*** and ***Runtime*** classes within ***java.lang*** Package.

1. **Using System.gc() method** :

   System class contain static method *gc()* for requesting JVM to run Garbage Collector. This is more convenient.

2. **Using Runtime.getRuntime().gc() method** :

   In *Runtime* class *gc()* is not a static method rather it is an instance method, so it can not be called directly. To invoke it we need the help of the static method *getRuntime()*.

Garbage Collection: Explicit Example

```
public class TestGarbage1{
    public void finalize(){
        System.out.println("object is garbage collected");
    }
    public static void main(String args[]){
        TestGarbage1 s1=new TestGarbage1();
        TestGarbage1 s2=new TestGarbage1();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

Calling the gc() suggests that the JVM expend effort toward recycling unused objects. When control returns, the JVM has made a best effort to reclaim space from all unused objects. There is no guarantee that this effort will recycle any particular number of unused objects, reclaim any particular amount of space, or complete at any particular time, if at all, before the method returns or ever.

Few References

- ***Java The Complete Reference***. Herbert Schildt, McGrawHill Education
- ***Object-Oriented Software Development Using Java***. Xiaoping Jia. Addison Wesley, ISBN 0-201-73733-7.
- ***Head First Object-Oriented Analysis and Design.*** Brett D. McLaughlin, Gary Pollice, and Dave West. O'Reilly.
- ***Head First Design Patterns.*** Eric Freeman and Elizabeth Freeman. O'Reilly.
- ***Java How to Program***. Paul Deitel, Harvey Deitel, Pearson