JAVA NoteB06-1

1. **INTRODUCTION [Dt-8.jan.25]**

Early 1990s:

* Developed by James Gosling and his team member called “Green Team” at Sun Microsystems.
* Initially they wanted to create a language called “OAK”

1995:

* Renamed "Java" after the Indonesian island known for its java coffee.
* First version, JDK 1.0, was released on January 23, 1996.

Key Features:

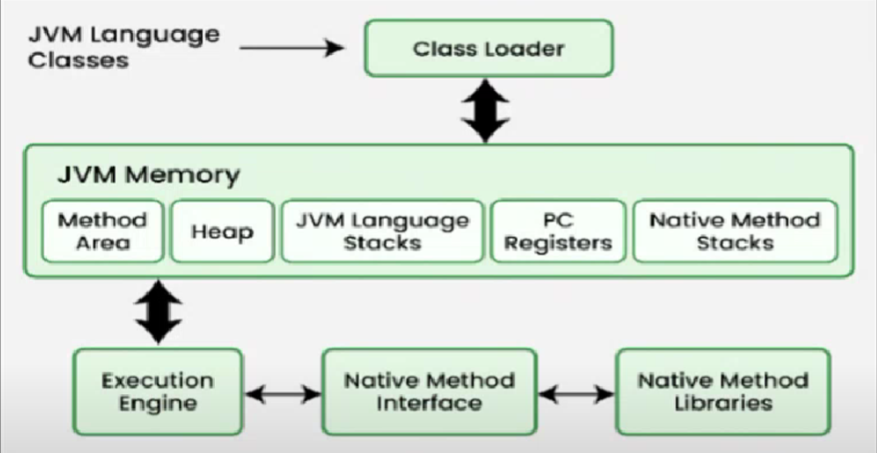
* Platform independence (write once, run anywhere).
* Object-oriented programming.
* Robust security.

Evolution:

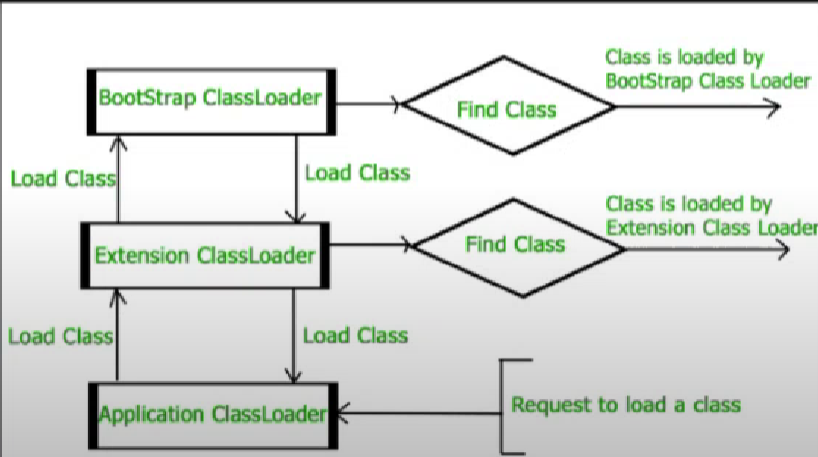
* Java has grown to include numerous new features and improvements.
* Widely used for web applications, mobile applications, enterprise solutions, and more.

Present:

* Maintained and developed by Oracle Corporation.
  1. **JVM Architechture**



* JVM(java virtual Machine) runs java application as a runtime engine.
* JVM calls the main method present in java code. It is a part of jre
* Java application are called WORA(Write Once Run Anywhere).
* Java code can run on any java enabled system. So it is platform Independent.
  1. **ClassLoaders:**



**What is a Class Loader?**

A class loader is a part of the Java Runtime Environment (JRE) that dynamically loads Java classes into the Java Virtual Machine (JVM). When your Java program is executed, it utilizes class loaders to load the necessary classes at runtime rather than at compile time.

**How Class Loaders Work**

1. **Loading**:
   * When a Java application requests a class that is not yet loaded, the class loader reads the class's bytecode from a .class file and transforms it into a java.lang.Class object.
2. **Linking**:
   * This involves three main steps:
     + **Verification**: Ensures the correctness of the bytecode.
     + **Preparation**: Allocates memory for class variables and initializes them to default values.
     + **Resolution**: Converts symbolic references to direct references.
3. **Initialization**:
   * This step involves executing the class's static initializers and static blocks.

**Types of Class Loaders**

1. **Bootstrap Class Loader**:
   * Loads the core Java classes (java.lang.\*, java.util.\*) from the bootstrap class path, typically the JRE’s rt.jar.
   * It is the parent class of all class loader.
2. **Extension Class Loader**:
   * Loads classes from the extension directories specified by the java.ext.dirs system property, usually the JRE’s lib/ext directory.
3. **System/Application Class Loader**:
   * Loads classes from the application's classpath, specified by the -classpath or -cp command-line options or the CLASSPATH environment variable.

**Custom Class Loaders**

Developers can create custom class loaders by extending the java.lang.ClassLoader class. This allows for defining unique class loading behaviors and is often used for applications requiring a high degree of modularity and security.

**How it Works (Example) :**

1. When java program executed- **Bootstrap Class Loader** first loads the essential classes like java.lang.Object.
2. When we create a class- It will be loaded by the **Application Class Loader.**
3. When we add external libraries(like jar file)- They might loaded by **Extension Class Loader** or custom class loader.(example: loading Spring,Hibernate these often use custom classloader).
4. We can write a custom Class Loader if we need to load classes from anon-standerd sourse(eg., netwirk,database).

**Code:**

**Public class MyClassLoader extends Classloader{**

**@Override**

**Public Class<?> loadClass(String name) throws ClassNotFoundException{**

**Sysout(“Loading class:”+name);**

**Return super.loadClass(name);**

**}**

**Public static void main(String[] args) throws Exception{**

**MyClassLoader myLoader = new MyClassLoader();**

**Class<?> clazz = myLoader.loadClass(“java.lang.String”);**

**Sysout(“Class loaded: ”+clazz.getName());**

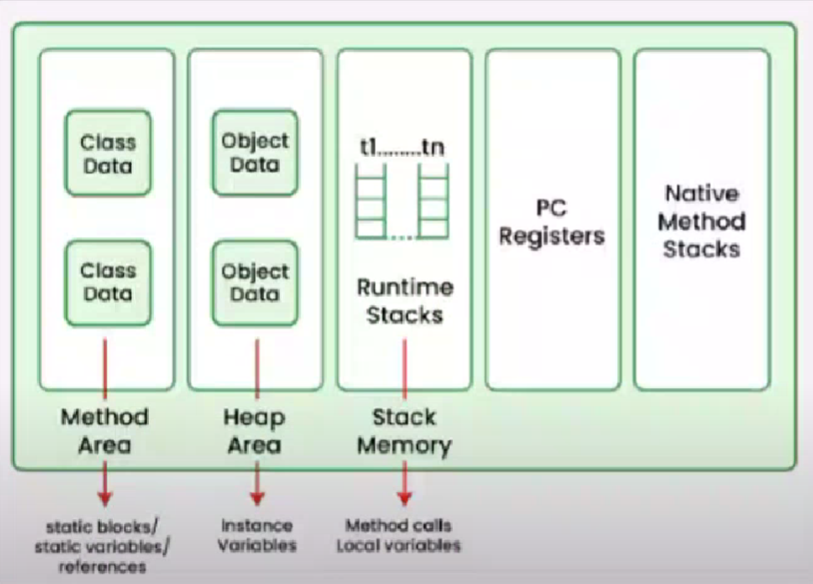
**}**

**}**

**Output:** Loading class: java.lang.String

Class loaded: java.lang.String

* 1. **JVM Memory Area:**



**JVM Memory Structure**

1. **Method Area**:
   * Stores class-level information, such as the structure of loaded classes (including metadata, method data, and constants).
   * Stores static variables and constants.
2. **Heap**:
   * All the objects (including arrays) and their instance variables are allocated on the heap.
   * This is where the runtime data area for new objects is created.
3. **Java Stacks**:
   * Each thread has its own private stack, created at the same time as the thread.
   * Stores frames, which hold local variables, operand stacks, and partial results.
   * Manages method invocations and returns.
4. **PC (Program Counter) Registers**:
   * Each thread has its own PC register to store the address of the JVM instruction currently being executed.
5. **Native Method Stacks**:
   * Used for native methods (methods written in a language other than Java, such as C or C++).

**Example:**

**public class JVMExample {**

**// Static variable (Method Area)**

**static int staticVar = 10;**

**// Instance variable (Heap)**

**int instanceVar;**

**// Constructor (Method Area, called during object creation)**

**public JVMExample (int value) {**

**this.instanceVar = value;**

**}**

**// Static method (Method Area)**

**public static void staticMethod() {**

**System.out.println("StaticVar: " + staticVar);**

**}**

**// Instance method (Method Area)**

**public void instanceMethod() {**

**System.out.println("InstanceVar: " + instanceVar);**

**}**

**// Main method (Method Area)**

**public static void main(String[] args) {**

**// Local variable (Stack)**

**int localVar = 5;**

**// Creating an instance of SimpleExample (Heap)**

**JVMExample obj = new JVMExample (20);**

**// Calling static method**

**staticMethod();**

**// Calling instance method**

**obj.instanceMethod();**

**// Printing local variable**

**System.out.println("LocalVar: " + localVar);**

**}**

**}**

* 1. The JVM loads the JVMexample class into the **method area**.
  2. When the main method starts, the stack is used to store local Variable and track the Static method call.
  3. The new JVMexample() locates memory in the heap of the object of obj.
  4. The PC Register ensures that each instruction is executed Sequentially.
  5. When the main method completes the stack frame for man is cleared but the object in the heap remains until garbage collection.

**Method Area**

* **Static Variable**: staticVar
* **Methods**:
  + staticMethod()
  + instanceMethod()
  + main(String[] args)
* **Constructor**:
  + JVMExample(int value)

**Heap**

* **Instance Variables**:
  + instanceVar
* **Object Instance**:
  + new JVMExample(20), referenced by obj

**Java Stack**

* **Local Variables**:
  + localVar in the main method
  + obj reference in the main method
* **Method Calls**:
  + Stack frames for staticMethod(), instanceMethod(), and main(String[] args)

**Summary**

* **Method Area**: Contains class-level information, including the static variable staticVar, the methods (staticMethod, instanceMethod, main), and the constructor.
* **Heap**: Stores the instance of JVMExample created by new JVMExample(20), and the instance variable instanceVar.
* **Java Stack**: Manages local variables (localVar and obj reference) and stack frames for method calls.