Java

JVM Architecture

Java compilation process

- Hello.java --> Java Compiler --> Hello.class
  - javac Hello.java
- Java compiler converts Java code into the Byte code.

Byte code

- Byte code is machine level instructions that can be executed by Java Virtual Machine (JVM).
  - Instruction = Op code + Operands
    - e.g. iadd op1, op2
- Each Instruction in byte-code is of 1 byte.
  - .class --> JVM --> Windows (x86)
  - .class --> JVM --> Linux (ARM)
- JVM converts byte-code into target machine/native code (as per architecture).

.class format

- .class file contains header, byte-code, meta-data, constant-pool, etc.
- .class header contains
  - magic number -- 0xCAFEBABE (first 4 bytes of .class file)
  - information of other sections
- .class file can inspected using "javap" tool.
  - terminal> javap java.lang.Object
    - Shows public and protected members
  - terminal> javap -p java.lang.Object
    - Shows private members as well

  - terminal> javap -c java.lang.Object
    - Shows byte-code of all methods
  - terminal> javap -v java.lang.Object
    - Detailed (verbose) information in .class
      - Constant pool
      - Methods & their byte-code
      - ...
- "javap" tool is part of JDK.

Executing Java program (.class)

- terminal> java Hello
- "java" is a Java Application Launcher.
- java.exe (disk) --> Loader --> (Windows OS) Process
- When "java" process executes, JVM (jvm.dll) gets loaded in the process.
- JVM will now find (in CLASSPATH) and execute the .class.

JVM Architecture (Overview)

- JVM = Classloader + Memory Areas + Execution Engine

Classloader sub-system

- Load and initialize the class

Loading

- Three types of classloaders
  - Bootstrap classloader: Load Java builtin classes from jre/lib jars (e.g. rt.jar).
  - Extended classloader: Load extended classes from jre/lib/ext directory.
  - Application classloader: Load classes from the application classpath.
- Reads the class from the disk and loads into JVM method (memory) area.

Linking

- Three steps: Verification, Preparation, Resolution
- Verification: Byte code verifier does verification process. Ensure that class is compiled by a valid compiler and not tampered.
- Preparation: Memory is allocated for static members and initialized with their default values.
- Resolution: Symbolic references in constant pool are replaced by the direct references.

Initialization

- Static variables of the class are assigned with assigned values.
- Execute static blocks if present.

JVM memory areas

- While execution, memory is required for byte code, objects, variables, etc.
- There are five areas: Method area, Heap area, Stack area, PC Registers, Native Method Stack area.

Method area

- Created at while JVM startup.
- Shared by all threads (global).
- Class contents (for all classes) are loaded into Method area.
- Method area also holds constant pool for all loaded classes.

Heap area

- Created at while JVM startup.
- Shared by all threads (global).
- All allocated objects (with new keyword) are stored in heap.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.

Stack area

- Separate stack is created for each thread in JVM (while creating thread).
- When a method is called from the stack, a new FAR (stack frame) is created on its stack.
- This stack frame conatins local variable array, operand stack, and other frame data.
- When method returns, the stack frame is destroyed.

PC Registers

- Separate PC register is created for each thread. It maintains address of the next instruction executed by the thread.
- After an instruction is completed, the address in PC is auto-incremented.

Native method stack area

- Separate native method stack is created for each thread in JVM (while creating thread).
- When a native method is called from the stack, a stack frame is created on its stack.

Monitor memory areas

jconsole

- jconsole (JAVA_HOME/bin) can be used to monitor memory area.

Runtime class

- The Runtime class can be used to monitor JVM memory. The following code prints memory sizes in bytes.

```
class MemoryMontitor {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Max Memory: " + rt.maxMemory());
        System.out.println("Total Memory: " + rt.totalMemory());
        System.out.println("Free Memory: " + rt.freeMemory());
    }
}
```

Execution engine

- The main comonent of JVM.
- Execution engine executes for executing Java classes.

Interpreter

- Convert byte code into machine code and execute it (instruction by instruction).
- Each method is interpreted by the interpreter at least once.
- If method is called frequently, interpreting it each time slow down the execution of the program.
- This limitation is overcomed by JIT (added in Java 1.1).

JIT compiler

- JIT stands for Just In Time compiler.
- Primary purpose of the JIT compiler to improve the performance.
- If a method is getting invoked multile times, the JIT compiler convert it into native code and cache it.
- If the method is called next time, its cached native code is used to speedup execution process.

Profiler

- Tracks resource (memory, threads, ...) utilization for execution.
- Part of JIT that identifies hotspots. It counts number of times any method is executing. If the number is more than a threshold value, it is considered as hotspot.

Garbage collector

- When any object is unreferenced, the GC release its memory.

JNI

- JNI acts as a bridge between Java method calls and native method implementations.

Java build tool - Maven

- YouTube Tutorial: https://youtu.be/lMXBrlVFYA0
- Tool to compile/build Java projects along with the required dependencies.
- Dependency = Third party libraries
- Maven does
  - Automatically download dependencies from Maven central repository (mvnrepository.com).
  - Add them into current project/classpath.
  - Build the application.
  - Run the test cases.
  - Package the application for deployment (as jar or war).
- Maven project configuration must be written in pom.xml file.
  - POM - Project Object Model
  - Heart of maven project. Present at root directory of the project.
  - It includes Java version, dependencies & their versions, build steps (if any).
- Maven Repositories
  - Central repository
    - All popular jars are presrent on maven central repository.
    - Location: mvnrepository.com
  - Local repository
    - Repository on local (Developer's) machine.
    - Location: C:/Users/nilesh/.m2 directory
    - Dependencies are downloaded from central repo to local repo and then added into the current project.
- Maven project in STS/Eclipse

- File -> New -> Project -> Maven -> Maven Project -> Next
- Check mark "Create a simple project" -> Next
- Fill project details and then Next.
    - groupId -- company/dept developing this project (usually same as package name) e.g. com.sunbeam
    - name -- name of project e.g. Demo20_2 (Usually same as artifactId)
    - artifactId -- name of packaged file e.g. Demo20_2 will create file Demo20_2.jar
    - packaging -- Jar/War
- Maven by default use Java version 1.5
- To change the version
    - In pom.xml, after

      ```
      <properties>
          <maven.compiler.source>11</maven.compiler.source>
          <maven.compiler.target>11</maven.compiler.target>
      </properties>
      ```

    - Project -> Right click -> Maven -> Update project -> Ok
- Add third party libraries (dependencies) in pom.xml

  ```
  <dependencies>
      <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>8.0.30</version>
      </dependency>
  </dependencies>
  ```

- Add main class and code to execute.
- Run as Java application.
- To create the artifact (.jar)

        - project -> Right click -> Run as -> Maven build -> Goal=package -> Run
        - artifactId-version.jar will be created in target directory of the project.

Maven Life-cycles and Phases

- Maven Life-cycles
    - build -- default life cycle
    - clean -- clear/delete generated files
    - site
- Maven Build Phases (in Build life-cycle)
    - validate -- check syntax of pom.xml and download dependencies (if not presrent in local repo)
    - compile -- compile all .java files.
    - test -- Run unit test cases in the application
    - package -- Create the artifact (.jar or .war)
    - install -- Copy the artifact into local repo
    - deploy -- Copy the artifact into remote repo