Java Architecture: JDK, JVM, and JRE

The Complete Reference by Herbert Schildt

Lecture Overview

- Introduction to Java Architecture
- Understanding JVM
- JDK and JRE
- Java Compilation Process
- Platform Independence and Bytecode
- Practical Examples

Introduction to Java Architecture

- **Java's Key Components**:
  - **JDK (Java Development Kit)**: The toolkit for developers.
  - **JVM (Java Virtual Machine)**: The core that runs Java programs.
  - **JRE (Java Runtime Environment)**: The runtime environment for executing Java applications.
- **Why Learn About Java Architecture?**
  - Understanding these components will help in developing efficient and platform-independent Java applications.

JVM (Java Virtual Machine) Overview

- **What is the JVM?**
  - The Java Virtual Machine is an abstract machine that enables Java applications to run on any device or operating system.
  - It provides platform independence by converting bytecode into machine code.
- **Key Responsibilities of the JVM**:
  - **Loading**: Loads class files.
  - **Verification**: Ensures bytecode is safe to run.
  - **Execution**: Executes bytecode via interpretation or Just-In-Time (JIT) compilation.
  - **Memory Management**: Manages heap and stack memory.
  - **Garbage Collection**: Automatically deallocates unused objects.

JVM Architecture

Key Components:

- **Class Loader**: Loads class files.
- **Execution Engine**:
  - **Interpreter**: Converts bytecode into machine-specific instructions.
  - **JIT Compiler**: Converts bytecode into machine code for performance improvement.
- **Memory Areas**:
  - **Method Area**: Stores class structures.
  - **Heap**: Stores objects.
  - **Stack**: Stores method call information and local variables.
- **Garbage Collector**: Reclaims memory from unused objects.

Diagram: Display the architecture of the JVM (class loader, heap, stack, etc.).

JDK (Java Development Kit) Overview

What is the JDK?

- A software development kit (SDK) used to develop Java applications.
- It includes the **JRE**, development tools, and libraries.

Components of the JDK:

- **Compiler (`javac`)**: Converts Java source code into bytecode.
- **Java Runtime Environment (JRE)**: Runs Java programs.
- **Java Debugger (`jdb`)**: Helps debug Java programs.
- **Other tools**: `jar` (packager), `javap` (disassembler), etc.

JRE (Java Runtime Environment) Overview

- **What is the JRE?**
  - The runtime portion of Java software required to run Java applications.
  - It includes the **JVM** and **core libraries** but does not include development tools like a compiler.
- **JRE Components**:
  - **JVM**: Executes the Java bytecode.
  - **Class Libraries**: Precompiled libraries that provide reusable functions and APIs for Java applications.
  - **Runtime Libraries**: Necessary files and libraries that support Java runtime execution.

Java Compilation Process

- **Steps in Java Program Execution**:
  1. **Write Source Code**:
     - The source code is written in `.java` files.
  2. **Compilation**:
     - The `javac` compiler compiles the source code into **bytecode** (`.class` files).
     - Bytecode is platform-independent.
  3. **Execution**:
     - The **JVM** interprets or compiles the bytecode into native machine code specific to the platform.
- **Diagram**: Show the process from writing code to running it via the JVM.

Platform Independence and Bytecode

- **What is Bytecode?**
  - A low-level code that is platform-independent and executed by the JVM.
- **How Platform Independence Works**:
  - The same bytecode can be executed on any device that has a JVM, making Java applications "Write Once, Run Anywhere" (WORA).
- **Example**:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

This code compiles to bytecode, which can be run on any system with a JVM.

Just-In-Time (JIT) Compiler

What is the JIT Compiler?

- A part of the JVM that improves performance by compiling bytecode into machine code at runtime.

How It Works:

- The first time the bytecode is interpreted, but frequently used methods are compiled into machine code and stored for future use.

Performance Benefits:

- Increases efficiency by reducing the need for repeated interpretation.

Memory Management and Garbage Collection

- **JVM Memory Areas**:
  - **Heap**: Stores objects.
  - **Stack**: Stores local variables and method calls.
  - **Method Area**: Stores class data.
- **Garbage Collection**:
  - **Automatic Memory Management**: JVM automatically reclaims memory occupied by objects that are no longer in use.

Example:

```
String s = new String("Hello");
s = null;  // "Hello" is eligible for garbage collection
```

Practical Examples

Example 1: Compilation and Execution of a Java Program.`java`

```
javac HelloWorld.java
```
java HelloWorld
This demonstrates the process from source code to execution.

Example 2: Bytecode Inspection.Use the `javap` tool to inspect the generated bytecode.

```
javap -c HelloWorld.class
```
This displays the bytecode for the `HelloWorld` class.

Best Practices

Use the JDK for Development:

- Always install the latest JDK for updated tools and libraries.

Understand the JVM:

- Configure JVM settings (like heap size) for optimized performance.

Memory Management:

- Keep an eye on memory usage and avoid memory leaks by using proper garbage collection techniques.

Summary

- The **JDK** is the complete toolkit for Java development, including the JRE and development tools.
- The **JVM** allows Java to run on any platform by executing bytecode.
- The **JRE** is needed to run Java applications but doesn't include development tools.
- Java’s platform independence is achieved through the use of **bytecode** and the **JVM**.
- The **JIT compiler** helps improve performance by converting bytecode into native machine code.