JDK, JRE, JVM

04 September 2020 04:16 PM

Topic Content

- JDK, JRE, JVM Overview.
- JVM Architecture.
- JIT Compiler
- Phases in JIT Compiler.

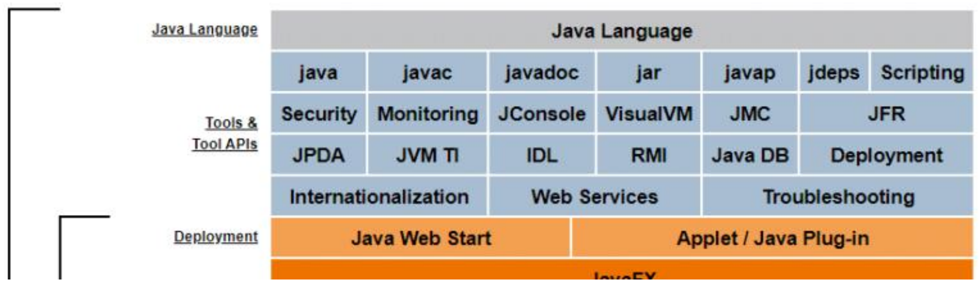
JDK, JRE, JVM overview

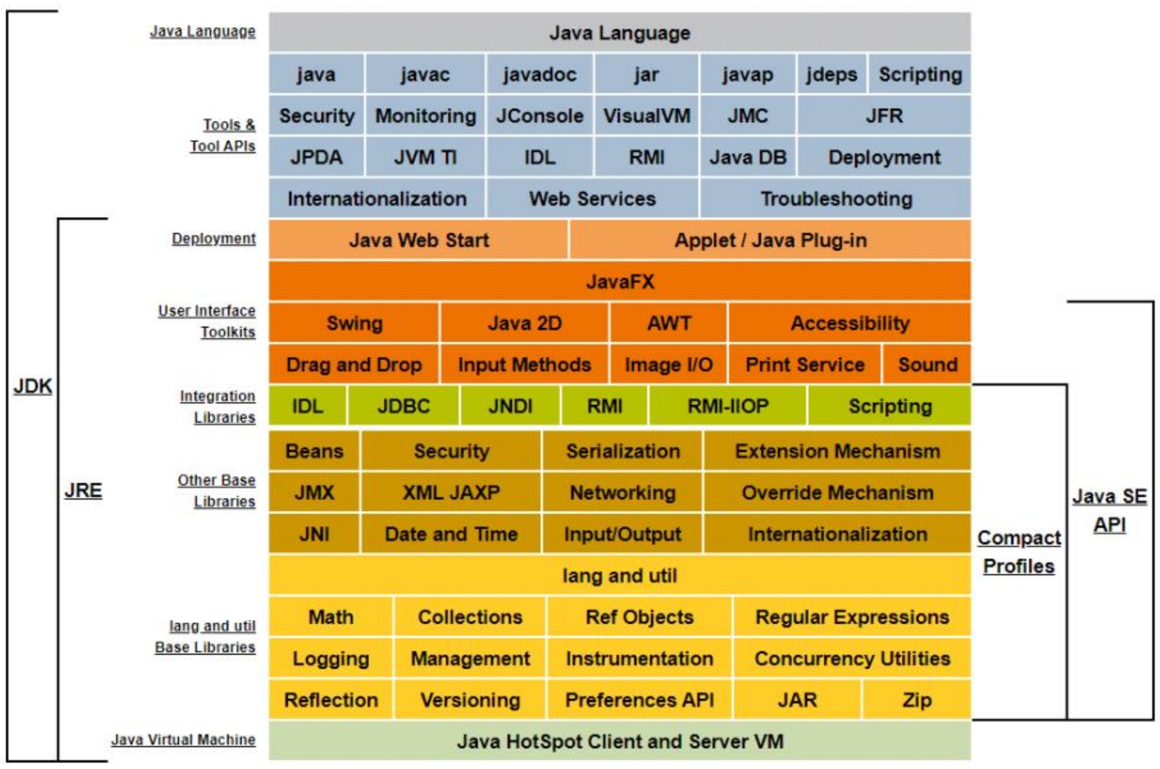
JDK = development, debugging, monitoring **tools** + **JRE**
JRE = base libraries (base binaries, deployment, User interface toolkits) + **JVM (Implementation)**
JVM = Specification of JVM

JDK is development kit which provides environment for the development of java applications, It has tools(**executable binaries**) like java (java interpreter), javac (compiler), jar, jconsole and many more to compile, execute and debug java program.

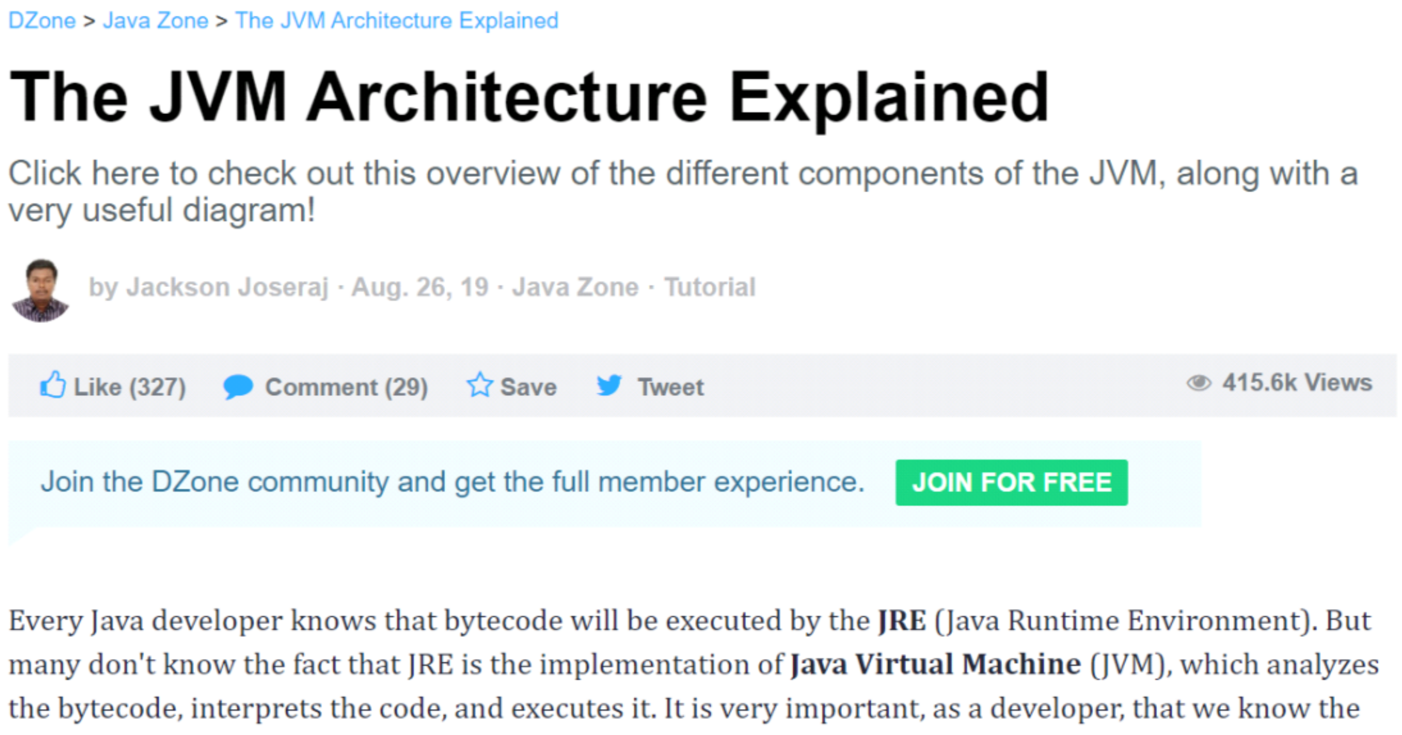
JRE stands for java runtime environment which has the implementation of JVM, java binaries and other classes which creates environment for the execution of java code. Additional to that it has java deployment and user interface toolkits

JVM is a specification of virtual machine whose implementation will be used to create environment for the execution of java code, JVM is the one responsible converting byte code generated by java compiler to platform specific machine code. JVM specification allows to implement different platform specific JVMs due to which we achieve platform independence and write once run anywhere (WORA) qualities.





Article about JVM Architecture



DZone > Java Zone > The JVM Architecture Explained

The JVM Architecture Explained

Click here to check out this overview of the different components of the JVM, along with a very useful diagram!

by Jackson Joseraj · Aug. 26, 19 · Java Zone · Tutorial

Like (327) Comment (29) Save Tweet 415.6k Views

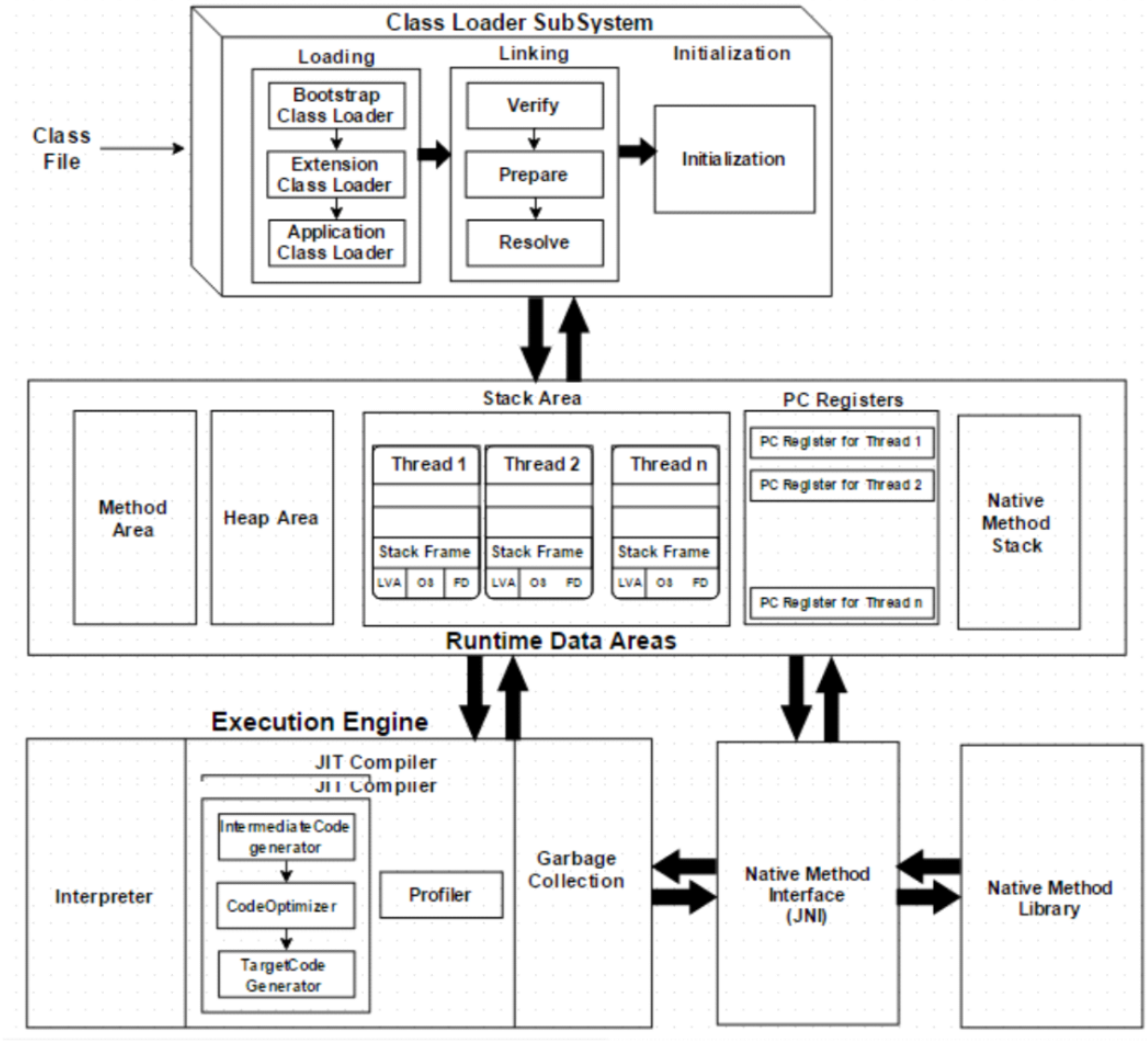
Join the DZone community and get the full member experience. JOIN FOR FREE

Every Java developer knows that bytecode will be executed by the **JRE** (Java Runtime Environment). But many don't know the fact that JRE is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it. It is very important, as a developer, that we know the many don't know the fact that JRE is the implementation of **Java Virtual Machine** (JVM), which analyzes the bytecode, interprets the code, and executes it. It is very important, as a developer, that we know the architecture of the JVM, as it enables us to write code more efficiently. In this article, we will learn more deeply about the JVM architecture in Java and different components of the JVM.

What Is the JVM?

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**. The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.
into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.
Below is a diagram of the Architecture of the JVM.

JVM Architecture Diagram



How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

1. ClassLoader Subsystem

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

1.1 Loading

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

1. **BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar.** Highest priority will be given to this loader.
2. **Extension ClassLoader** – Responsible for loading classes which are inside the ext folder **(jre\lib).**
3. **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.

1.2 Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

1.3 Initialization

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

2. Runtime Data Area

The Runtime Data Area is divided into five major components:

1. **Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored

2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored

here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.
3. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:
    1. **Local Variable Array** – Related to the method how many local variables are involved and the corresponding values will be stored here.
    2. **Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
    3. **Frame data** – All symbols corresponding to the method is stored here. In the case of any **exception**, the catch block information will be maintained in the frame data.
4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
5. **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    1. **Intermediate Code Generator** – Produces intermediate code
    2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
    3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
    4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling `System.gc()`, but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries, which is required for the Execution Engine.

Originally published September 2016

Further Reading

Java Memory Management

A Detailed Breakdown of the JVM

The Evolution of the Java Memory Architecture

Article about JIT Compiler

JIT Compiler

The JIT Compiler compiles bytecode to machine code at runtime and improves the performance of Java applications.

Of course, JIT compilation does require processor time and memory usage. When the JVM first starts up, lots of methods are called. Compiling all of these methods might affect startup time significantly, though a program ultimately might achieve good performance.

Methods are not compiled when they are called the first time. For each and every method, the JVM maintains a call count, which is incremented every time the method is called. The methods are interpreted by the JVM until the call count exceeds the JIT compilation threshold (the JIT compilation threshold improves performance and helps the JVM to start quickly. The threshold has been selected carefully by Java developers for optimal performance. The balance between startup times and long-term performance is maintained).

Therefore, very frequently used methods are compiled as soon as the JVM has started, and less frequently used methods are compiled later.

After a method is compiled, its call count is reset to zero, and subsequent calls to the method increment its call count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles method a second time, applying more optimizations as compared to optimizations applied in the previous compilation. This process is repeated until the maximum optimization level is reached. The most frequently used methods are always optimized to maximize the performance benefits of using the JIT compiler the previous compilation. This process is repeated until the maximum optimization level is reached. The most frequently used methods are always optimized to maximize the performance benefits of using the JIT compiler.

Let's say the JIT recompilation threshold = 2.

After a method is compiled, its call count is reset to zero and subsequent calls to the method increment its call count. When the call count of a method reaches 2 (i.e. JIT recompilation threshold), the JIT compiler compiles the method a second time, applying more optimizations.

Phases of just-in-time compilation

As mentioned, a JIT compiler will compile suitable bytecode sequences into machine code, and then the code is sent to a processor where the code instructions are carried out.

JIT compilation can also be separated by different levels of optimization: cold, warm, hot, very hot and scorching. Each optimization level is set to provide a certain level of performance. The initial, or default, level of optimization is called warm, while code that looks like it can be further re-optimized is called hot. This level increases upwards until scorching, with each level being of higher importance pertaining to performance that can be re-optimized. Through sampling, a JIT compiler can determine which methods appear more often at the top of a stack. The optimization level can also be downgraded to cold, to further improve startup time.