# Difference Between JVM, JRE, and JDK

baeldung.com/jvm-vs-jre-vs-jdk

By
baeldung

7 June 2018

## 1. Overview

In this article, we'll discuss differences between JVM, JRE, and JDK by considering their components and uses.

## 2. JVM

**Java Virtual Machine (JVM) is an implementation of a virtual machine which executes a Java program.**

The JVM first interprets the bytecode. It then stores the class information in the memory area. Finally, it executes the bytecode generated by the java compiler.

It is an abstract computing machine with its own instruction set and manipulates various memory areas at runtime.

Components of the JVM are:

- Class Loaders
- Run-Time Data Areas
- Execution Engine

### 2.1. Class Loaders

Initial tasks of the JVM includes loading, verifying and linking the bytecode. Class loaders handle these tasks.

We have a detailed article specifically on class loaders.

### 2.2. Run-Time Data Areas

**The JVM defines various memory areas to execute a Java program.** These are used during runtime and are known as run-time data areas. Some of these areas are created on the JVM start-up and destroyed when the JVM exits while some are created when a thread is created and destroyed when a thread exits.

Let's consider these areas one by one:

**Method Area**

Basically, method area is analogous to the storage area for compiled code. It stores structures such as run-time constant pool, field and method data, the code for methods and constructors as well as fully qualified class names. The JVM stores these structure for each and every class.

The method area, also known as permanent generation space (PermGen), is created when the JVM starts up. The memory for this area does not need to be contiguous. All the JVM threads share this memory area.

**Heap Area**

The JVM allocates the memory for all the class instances and arrays from this area.

Garbage Collector (GC) reclaims the heap memory for objects. Basically, GC has three phases to reclaim memory from objects viz. two minor GC and one major GC.

The heap memory has three portions:

- Eden Space – it's a part of Young Generation space. When we create an object, the JVM allocates memory from this space
- Survivor Space – it's also a part of Young Generation space. Survivor space contains existing objects which have survived the minor GC phases of GC
- Tenured Space – this is also known as the Old Generation space. It holds long surviving objects. Basically, a threshold is set for Young Generation objects and when this threshold is met, these objects are moved to tenured space.

JVM creates heap area as soon as it starts up. All the threads of the JVM share this area. The memory for the heap area does not need to be contiguous.

**Stack area**

Stores data as frames and each frame stores local variables, partial results and nested method calls. JVM creates the stack area whenever it creates a new thread. This area is private for each thread.

Each entry in the stack is called Stack Frame or Activation record. Each frame contains three parts:

- Local Variable Array – contains all the local variables and parameters of the method
- Operand Stack – used as a workspace for storing intermediate calculation's result
- Frame Data – used to store partial results, return values for methods, and reference to the *Exception* table which provides corresponding catch block information in case of exceptions

The memory for the JVM stack does not need to be contiguous.

**PC Registers**

Each JVM thread has a separate PC Register which stores the address of the currently executing instruction. If the currently executing instruction is a part of the native method then this value is undefined.

**Native method stacks**

Native methods are those which are written in languages other than Java.

JVM provides capabilities to call these native methods. Native method stacks are also known as “C stacks”. They store the native method information. Whenever the native methods are compiled into machine codes, they usually use a native method stack to keep track of their state.

The JVM creates these stacks whenever it creates a new thread. And thus JVM threads don't share this area.

## 2.3. Execution Engine

Execution engine executes the instructions using information present in the memory areas. It has three parts:

**Interpreter**

Once classloaders load and verify bytecode, the interpreter executes the bytecode line by line. This execution is quite slow. The disadvantage of the interpreter is that when one method is called multiple times, every time new interpretation is required.

However, the JVM uses JIT Compiler to mitigate this disadvantage.

**Just-In-Time (JIT) Compiler**

JIT compiler compiles the bytecode of the often-called methods into native code at run-time. Hence it is responsible for the optimization of the Java programs.

JVM automatically monitors which methods are being executed. Once a method becomes eligible for JIT compilation, it is scheduled for compilation into machine code. This method is then known as a hot method. This compilation into machine code happens on a separate JVM thread.

As a result, it does not interrupt the execution of the current program. After compilation into machine code, it runs faster.

**Garbage Collector**

Java takes care of memory management using Garbage Collection. It's a process of looking at heap memory, identifying which objects are in use and which are not, and finally deleting unused objects.

GC is a daemon thread. It can be called explicitly using *System*.gc*()* method, however, it won't be executed immediately and the JVM decides when to invoke GC.

### 2.4. Java Native Interface

It acts as an interface between the Java code and the native (C/C++) libraries.

There are situations in which Java alone doesn't meet the needs for your application, for example, implementing a platform-dependent feature.

In those cases, we can use JNI to enable the code running in the JVM to call. Conversely, it enables native methods to call the code running in the JVM.

### 2.5. Native Libraries

These are platform specific libraries and contains the implementation of native methods.

## 3. JRE

**Java Runtime Environment (JRE) is a bundle of software components used to run Java applications.**

Core components of the JRE include:

- An implementation of a Java Virtual Machine (JVM)
- Classes required to run the Java programs
- Property Files

We discussed the JVM in the above section. Here we will focus on the core classes and support files.

### 3.1. Bootstrap Classes

We'll find bootstrap classes under *jre/lib/*. **This path is also known as the bootstrap classpath.** It includes:

- Runtime classes in *rt.jar*
- Internationalization classes in *i18n.jar*
- Character conversion classes in *charsets.jar*
- Others

Bootstrap ClassLoader loads these classes when the JVM starts up.

### 3.2. Extension Classes

We can find extension classes in *jre/lib/extn/* which acts as a directory for extensions to the Java platform. **This path is also known as extension classpath.**

It contains JavaFX runtime libraries in *jfxrt.jar* and locale data for *java.text* and *java.util* packages in *localedata.jar*. Users can also add custom jars into this directory.

### 3.3. Property Settings

Java platform uses these property settings to maintain its configuration. Depending on their usage they are located in different folders inside */jre/lib/*. These include:

- Calendar configurations in the *calendar.properties*
- Logging configurations in *logging.properties*
- Networking configurations in *net.properties*
- Deployment properties in */jre/lib/deploy/*
- Management properties in */jre/lib/management/*

### 3.4. Other Files

Apart from the above-mentioned files and classes, JRE also contains files for other matters:

- Security management at *jre/lib/security*
- The directory for placing support classes for applets at *jre/lib/applet*
- Font related files at *jre/lib/fonts* and others

## 4. JDK

**Java Development Kit (JDK) provides environment and tools for developing, compiling, debugging, and executing a Java program.**

Core components of JDK include:

- JRE
- Development Tools

We discussed the JRE in the above section.

Now, we'll focus on various development tools. Let's categorize these tools based on their usage:

## 4.1. Basic Tools

These tools lay the foundation of the JDK and are used to create and build Java applications. **Among these tools, we can find utilities for compiling, debugging, archiving, generating Javadocs, etc.**

They include:

- *javac* – reads class and interface definitions and compiles them into class files
- *java* – launches the Java application
- *javadoc* – generates HTML pages of API documentation from Java source files
- *apt* – finds and executes annotation processors based on the annotations present in the set of specified source files
- *appletviewer* – enables us to run Java applets without a web browser
- *jar* – packages Java applets or applications into a single archive
- *jdb* – a command-line debugging tool used to find and fix bugs in Java applications
- *javah* – produces C header and source files from a Java class
- *javap* – disassembles the class files and displays information about fields, constructors, and methods present in a class file
- *extcheck* – detects version conflicts between target Java Archive (JAR) file and currently installed extension JAR files

## 4.2. Security Tools

**These include key and certificate management tools that are used to manipulate Java Keystores.**

A Java Keystore is a container for authorization certificates or public key certificates. Consequently, it is often used by Java-based applications for encryption, authentication, and serving over HTTPS.

Also, they help to set the security policies on our system and create applications which can work within the scope of these policies in the production environment. These include:

- *keytool* – helps in managing keystore entries, namely, cryptographic keys and certificates
- *jarsigner* – generates digitally signed JAR files by using keystore information
- *policytool* – enables us to manage the external policy configuration files that define installation's security policy

**Some security tools also help in managing Kerberos tickets.**

Kerberos is a network authentication protocol.

It works on the basis of tickets to allow nodes communicating over a non-secure network to

prove their identity to one another in a secure manner:

- *kinit* – used to obtain and cache Kerberos ticket-granting tickets
- *ktab* – manages principle names and key pairs in the key table
- *klist* – displays entries in the local credentials cache and key table

## 4.3. Internationalization Tool

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes.

**For this purpose, the JDK brings *native2ascii*. This tool converts a file with characters supported by JRE to files encoded in ASCII or Unicode escapes.**

## 4.4. Remote Method Invocation (RMI) Tools

**RMI tools enable remote communication between Java applications thus providing scope for development of distributed applications.**

RMI enables an object running in one JVM to invoke methods on an object running in another JVM. These tools include:

- *rmic* – generates stub, skeleton, and tie classes for remote objects using the Java Remote Method Protocol (JRMP) or Internet Inter-Orb Protocol (IIOP)
- *rmiregistry* – creates and starts remote object registry
- *rmid* – starts the activation system daemon. This allows objects to be registered and activated in a Java Virtual Machine
- *serialver* – returns serial version UID for specified classes

## 4.5. Java IDL and RMI-IIOP Tools

**Java Interface Definition Language (IDL) adds Common Object-Based Request Broker Architecture (CORBA) capability to the Java platform.**

These tools enable distributed Java web applications to invoke operations on remote network services using industry standard Object Management Group (OMG) – IDL.

Likewise, we could use Internet InterORB Protocol (IIOP).

RMI-IIOP, i.e. RMI over IIOP enables programming of CORBA servers and applications via the RMI API. Thus enabling connection between two applications written in any CORBA-compliant language via Internet InterORB Protocol (IIOP).

These tools include:

- *tnameserv* – transient Naming Service which provides a tree-structured directory for object references
- *idlj* – the IDL-to-Java Compiler for generating the Java bindings for a specified IDL file
- *orbd* – enable clients to transparently locate and invoke persistent objects on the server in CORBA environment
- *servertool* – provides command-line interface to register or unregister a persistent server with ORB Daemon (*orbd*), start and shut down a persistent server registered with ORB Daemon, etcetera

## 4.6. Java Deployment Tools

**These tools help in deploying Java applications and applets on the web.** They include:

- *pack200* – transforms a JAR file into a *pack200* file using the Java *gzip* compressor
- *unpack200* – transforms *pack200* file into a JAR file

## 4.7. Java Plug-in Tool

**JDK provides us with *htmlconverter*. Furthermore, it's used in conjunction with the Java Plug-in.**

On the one hand, Java Plug-in establishes a connection between popular browsers and the Java platform. As a result of this connection, applets on the website can run within a browser.

On the other hand, *htmlconverter* is a utility for converting an HTML page containing applets to a format for Java Plug-in.

## 4.8. Java Web Start Tool

JDK brings *javaws*. We can use it in conjunction with the Java Web Start.

**This tool allows us to download and launch Java applications with a single click from the browser.** Hence, there is no need to run any installation process.

## 4.9. Monitoring and Management Tools

**These are great tools that we can use to monitor JVM performance and resource consumption.** Here are a few of these: :

- *jconsole* – provides a graphical console that lets you monitor and manage Java applications
- *jps* – lists the instrumented JVMs on the target system

- *jstat* – monitors JVM statistics
- *jstatd* – monitors creation and termination of instrumented JVMs

### 4.10. Troubleshooting Tools

**These are experimental tools that we can leverage for troubleshooting tasks:**

- *info* – generates configuration information for a specified Java process
- *jmap* – prints shared object memory maps or heap memory details of a specified process
- *jsadebugd* – attaches to a Java process and acts as a debug server
- *jstack* – prints Java stack traces of Java threads for a given Java process

## 5. Conclusion

In this article, we identified that the basic difference between JVM, JRE, and JDK lies in their usage.

First, we described how the JVM is an abstract computing machine that actually executes the Java bytecode.

Then, we explained how to just run Java applications, we use the JRE.

And finally, we understood how to develop Java applications, we use the JDK.

We also took some time to dig into tools and fundamental concepts of this components.

### I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

**>> CHECK OUT THE COURSE**

Learning to "Build your API with Spring"?