# Java Virtual Machine Specification

## Introduction

JVM specification is an abstract description. It doesn’t describe any particular implementation of the Java Virtual Machine.

To implement the java Virtual machine correctly, you need only be able to read the class file format and correctly perform the operations speciffied in that class file.

Implementation is not part of the specification which unnecessarly constraint the creativity of implementors. Which means implementor can have their logic to achive the final result.

1. Memory Layout of the runtime data areas was chosen by the implementor it is not part of specification.
2. Implementor can be choosen their own Garbage collector algorithm and it is not part of the Specification.

## Data types

JVM operates on two kind of types: primitive and reference types. Checking of this data types can be done prior to run-time, typically by a compiler, and does not to be done by the Java Virtual Machine itself.

The instruction set of the JVM distingushes its operand types using instructions intended to operate on values of specified types.

Ex: iadd, ladd, fadd, dadd are all JVM instructions that add two numeric values and produce numeric results. But each is specialized for its operand type: int, long, float and double, respectively.

1. Byte: whose values are 8-bit signed two’s-complement integers, and whose default value is zero.
2. Short: whose values are 16-bit signed two’s-complement integers, and whose default value is zero
3. Int: whose valuea are 32-but signed
4. Long: whose values are 64-bit signed
5. Char: 16-bit unsigned

Integral Types and values

The values of the integral types of the java Virtual Machine are:

1. For byte, from -128 to 127, inclusive
2. For short, from -2^15 to 2^15-1
3. For int, from -2^35 to 2\_15-1
4. For long, from -2^63 to 2^63-1
5. For char, from 0 to 65535 inclusive.

## Runtime Data Areas

JVM defines various run-time data areas that are used duing execution of a program. Some of these data areas are created on JVM stat-up and are destroyed only when JVM exists. Other areas are per thread. Per-thread areas are created when a thread is created and destroyed when the thread exists.

1. **PC Register:** Each JVM thread has its own PC Register. At any point, each JVM thread is executing the code of a single method, namely the current method for that thread. If that method is not native, **the pc register contains the address of the JVM instruction currently being executed.** If it is native pc register values is undefined. JVM’s pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.
2. **JVM Stacks**: Each JVM thread has a private JMV Stack. Created at the same time as the thread. A JVM stack stores frames. **It holds local variables a**nd **partial results**, plays a part in method invocation and return. JVM stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a JVM stack does not need to be contiguous (side by side). This specification permits JVM stacks either to be of a fixed size or to dynamically expand.

**Exceptional conditions are associated with JVM stacks:**

* If the computation in a thread requires a larger JVM stack than is permitted, the JVM throws a **StackOverflowError**.
* If the JVM stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to affect the expansion, the JVM throws an **OutOfMemoryError**.

1. **Heap**: JVM has heap memory that is shared among all the JVM threads. **The heap is the run-time data area from which memory for all class instances and arrays is allocated.**

The heap is created on Virtual Machine start-up. Heap storage for objects is reclaimed is by an automatic storage management system. Objects are never explicitly deallocated.

NOTE: JVM specification doesn’t tell particular type of automatic storage management system, and the storage management technique may be chosen acoording to the implementor’s system requirements. Heap may be fixed or expanded as required. The memory for the heap does not need to be contiguous.

If a computation requires more space than can be made available by automatic storage system, the JVM throws an **OutOfMemoryError.**

1. **Method Area**: It is also shared among all JVM threads. The method area is analogous to the storage area for compiled code of a conventional language or anlogous to the “text” segment in an OS process. It stores per-class structures such as the **run-time constant pool**, **field** and **method data,** and the code for **methods** and **constructors**.

**NOTE:** method area is logically part of heap. If the memory is not sufficient then it may throw OutOfMemoryError

1. **Runtime Constant Pool:** it is per-class or per-interface run-time representation of the constant\_pool table in a class file. It contains several kinds of constants, rangin from numeric literals to method and fields references that must be resolved at run-time. It is similar to the symbol table for a conventional programming language.

Run-time constant pool is allocated from the Method area (Permgen space of the heap/ metaspace on Java8).

1. **Native method stacks:** it may store native method instructon sets. (Need to elaborate)

Frames: Frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked and destoyed when its method invocation completes, whether that completion is normal or abrupt. Frames are allocated from the JVM stack of the thread.

Each frame has following things

1. Array of local variables
2. Operand stack
3. Reference to the run-time constant pool of the class of the current method.

<https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html>