Introduction Java Technology	The Java Programming Language	What Can Java Technology Do?	Introduction C++
Why can't computer languages like C or C++ have virtual machines like Java?	Why are Virtual Machines necessary?	JVM (Java Virtual Machine)	Internal Architecture of JVM (Not in syllabus)

Introduction Java Technology

- **Java** is a general-purpose strongly typed computer programming language that is
  - concurrent,
  - class-based,
  - object-oriented

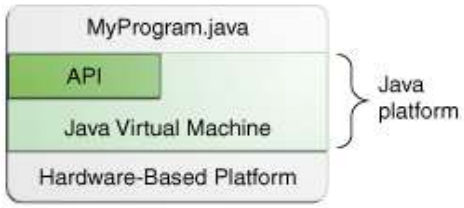
Concurrent means several computations are executed during overlapping time periods—*concurrently*—instead of *sequentially.*

- **Java technology** is both a programming language and a platform. Like any modern technology it enables the development of *secure, high performance,* and highly *robust* applications on *multiple platforms* in *heterogeneous*, *distributed networks*.

- The **Java platform** is a suite of programs that facilitate developing and running programs written in the Java. Most platforms are a combination of the operating system and underlying hardware. But Java platform is a software-only platform that runs on top of other hardware-based platforms. The Java platform has two components:

  1. The Java Virtual Machine 2. The Java Application Programming Interface (API)

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages.



The platform is not specific to any one processor or operating system, rather an execution engine (called a virtual machine) and a compiler with a set of libraries (API) are implemented for various hardware and operating systems so that Java programs can run identically on all of them. There are multiple platforms, each targeting a different class of devices:

- **Java Card**: For small Java app (applets) to be run securely on smart cards/ small-memory devices.
- **Java ME** (Micro Edition): For devices with limited storage, display, and power capacities. Often used to develop applications for mobile devices, PDAs, TV set-top boxes, and printers.
- **Java SE (Standard Edition)**: For general-purpose use on desktop PCs, servers and similar devices.
- **Java EE (Enterprise Edition)**: Java SE + various APIs for multi-tier client–server enterprise app.

The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

• **Simple**	• **Architecture neutral**	• **Robust**	• **Multithreaded**
• **Object oriented**	• **High performance**	• **Secure**	• **Portable**
• **Distributed**		• **Dynamic**	

- **Simple and Familiar**

  - Easy to grasp the fundamental concepts and Extensive programmer training is not required.
  - Programmers can be productive from the very beginning.
  - It retains the object-oriented features without unnecessary complexities of C++.
  - Rich libraries of tested objects exist to provide functionality ranging from basic data types through I/O and network interfaces to graphical user interface toolkits. These libraries can be extended to provide new behavior.

- **Robust and Secure**

  - It provides extensive compile-time checking, followed by a second level of run-time checking.
  - Language features guide programmers towards reliable programming habits.
  - The memory management model is extremely simple:
    - Objects are created with a new operator
    - No explicit programmer-defined pointer data types, no pointer arithmetic
    - Automatic garbage collection.

  - Java technology is designed to operate in distributed environments, so security is of paramount importance. Security features are provided into the language and run-time system.

- **Object Oriented –** We will discuss it late.
- **Architecture Neutral and Portable**

Java applications can be deployed into heterogeneous network environments i.e. a variety of operating systems over a variety of hardware architectures and interoperate with multiple programming language interfaces. The Java Compiler generates *bytecodes*--an *architecture neutral* intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java technology solves both the binary distribution problem and the version problem; the same Java programming language byte codes will run on any platform.

Architecture neutrality is just one part of a truly *portable* system. In addition it specifies the sizes of its basic data types and the behavior of its arithmetic operators. Your programs are the same on every platform--there are no data type incompatibilities across hardware and software architectures.

The architecture-neutral and portability is implemented using the *Java virtual machine* (which we will discuss later).

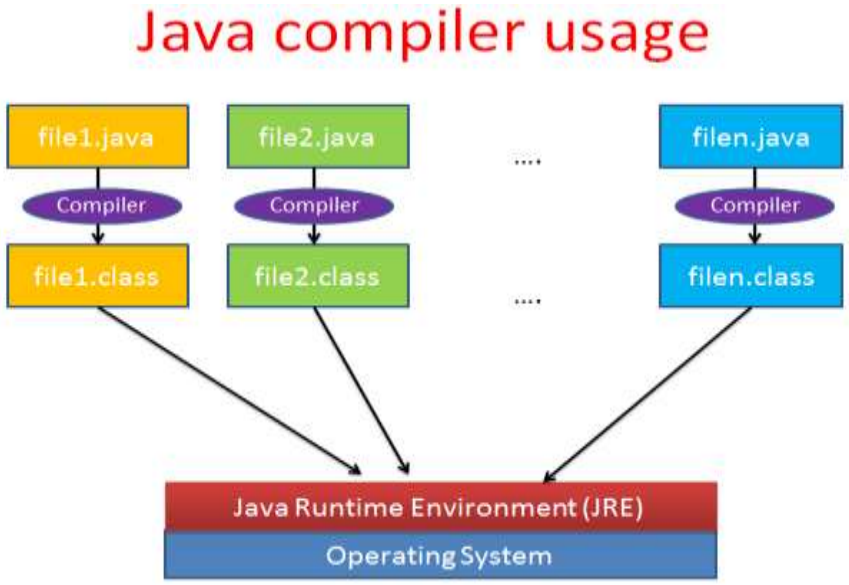
- **High Performance, Dynamic, Multithreaded**

JVM (the interpreter) can run at full speed without needing to check the run-time environment. The *automatic garbage collector* runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.

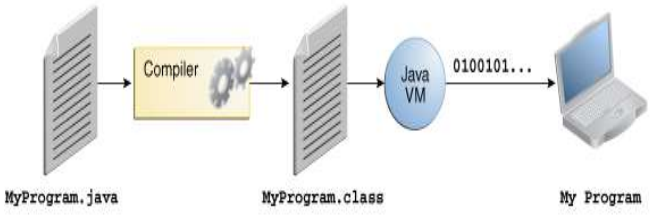
The link phase of a program is simple, incremental, and lightweight. While the Java Compiler is strict in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network.

The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives.

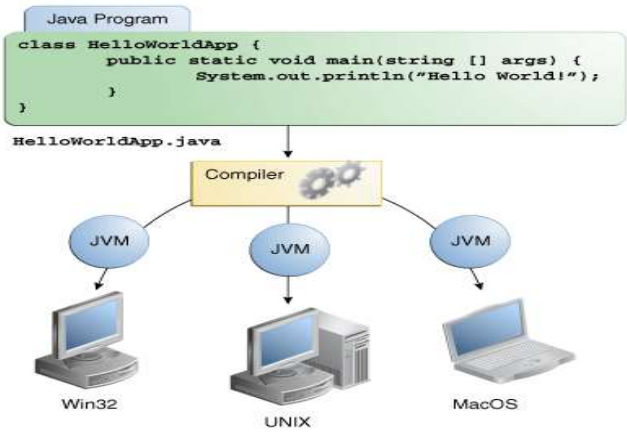
Development and Execution of Java Program



Write source code in plain text files ending with the .java extension. Compile it into .class files by the **javac** compiler. A .class file contains *bytecodes*. The java launcher tool then runs your application with an instance of the JVM.



An overview of the software development process.



Java VM is available on many different operating systems, Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.

What Can Java Technology Do?

Every full implementation of the Java platform gives you the following features:

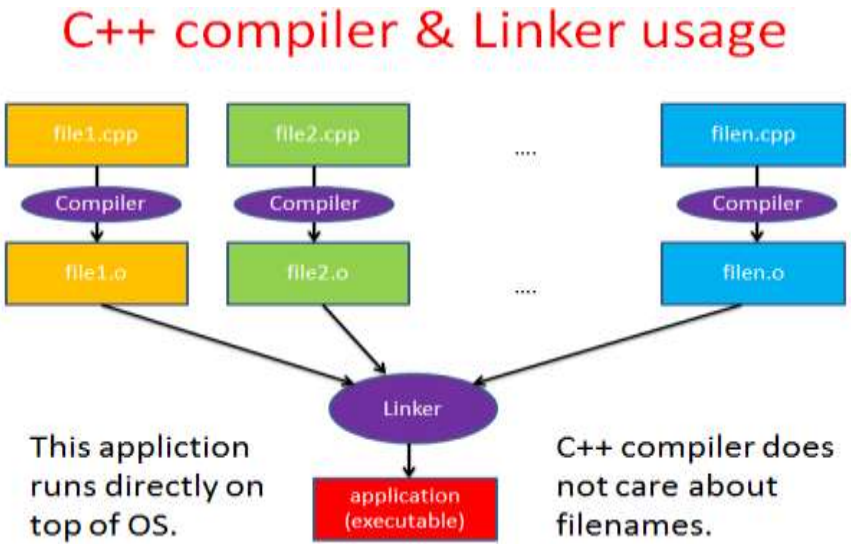
- **Development Tools**: The development tools provide everything you'll need for compiling, running, monitoring, debugging, and documenting your applications, example : the **javac** compiler, the **java launcher**, and the **javadoc** documentation tool.

- **Application Programming Interface (API): The API provides the core functionality of the Java** programming language. It offers a wide array of useful classes ready for use in your own applications. It spans everything from basic objects, to networking and security, to XML generation and database access, and more. The core API is very large; to get an overview of what it contains, consult the Java Platform Standard Edition 8 Documentation.
- **Deployment Technologies**: The JDK software provides standard mechanisms such as the Java Web Start software and Java Plug-In software for deploying your applications to end users.
- **User Interface Toolkits**: The JavaFX, Swing, and Java 2D toolkits make it possible to create sophisticated Graphical User Interfaces (GUIs).
- **Integration Libraries**: Integration libraries such as the Java IDL API, JDBC API, Java Naming and Directory Interface (JNDI) API, Java RMI, and Java Remote Method Invocation over Internet Inter-ORB Protocol Technology (Java RMI-IIOP Technology) enable database access and manipulation of remote objects.

Introduction C++

- C++ is termed a "**better C**" and can be learned just as easily as C.
- C++ is a small, block-structured, class based, object-oriented programming (OOP) language.
- It is a strongly typed language. C is known as a ***weakly typed*** language; variable data types do not necessarily have to hold the same type of data. For example, if you declare an integer variable and decide to put a character value in it, C enables you to do so. The data might not be in the format you expect, but C does its best.
- It has **fewer** than 46 keywords but **largest** variety of ***operators*** (almost equal to the number of keywords) such as +, -, and && (second only to APL).
- **No input or output statements** - the most important reasons for availability on so many different computers. The I/O (input/output) statements of most languages tie those languages to specific hardware. BASIC, for instance, has almost twenty I/O commands—some of which write to the screen, to the printer, to a modem, and so forth.
- C++'s input and output are performed through the abundant use of operators and function calls. With every C++ compiler comes a library of standard I/O functions. I/O functions are hardware independent.

- To master C++ completely, you have to be more aware of your computer's hardware than most other languages would require you to be. You certainly do not have to be a hardware expert, but understanding the internal data representation makes C++ much more usable and meaningful.
- In the world of PCs, both Borland and Microsoft, two of the leading names of PC software, offer full-featured C++ compilers.



Why can't computer languages like C or C++ have virtual machines like Java?

We may have a VM for each and every language you imagine. But **VM core** itself needs to be written and compiled into machine code. So there must be some high level language compilers which are capable of generating machine code directly (no intermediate code like VM).

Actually many VM for C/C++ exist. You can even compile Java and run it outside of a virtual machine (GCJ).

C or C++ code compiled into machine code is able to run on any computer as long as it is the compatible OS. If C code can run on any CPU then what is the purpose of the Virtual Machine?

Although all processors have differences there is a common standardization. The **AMD** and **intel** processors use the same instruction set and machine architecture x86 architecture (from the standpoint of execution of machine code). Primarily the X86 architecture provides a large enough set of common features so that the C code compiled on one X86 processor will work for the most part on another X86 processor.

However, if you want to use processor-specific properties, then you need a compiler or set of libraries which will do that for you. If you want to run c/c++ program on a different architecture you have to use a compiler for that architecture, the same binary executable won't run across different processor architectures.

If you want to run compiled program on X86 architecture on an ARM processor, or MIPS, or PowerPC, then you have to run a full machine instruction set **emulator** that interprets the binary machine code from X86 into whatever machine you're running it on.

Any C/C++ program can be compiled to an intermediate step and this intermediate step can be run on a virtual machine. The intermediate step, just to be clear, could be x86 opcodes, ARM opcodes, bytecodes, or any other representation that represents the original program. This type of virtual machine is typically called an **emulator**. Adding an emulator managed virtual memory space, and a few other additions and we have a relatively full fledged virtual machine in the classic sense. So, yes…you can run C/C++/whatever language on a virtual machine.

But the nature and use of the C/C++ language makes it perhaps not suitable to compile to an intermediate code (like Java bytecode) and then run by a VM, because

- C and C++ were created to make operating systems and applications based on them. The compilers for them are available for a huge variety of hardware. In fact if you design a new computer architecture, one of the first software tools you produce is a C compiler.

- It is very likely that you want to get direct access to the underlying hardware. Direct access to the hardware is exactly what a virtual machine prevents!

- C and C++, because they can compile directly to hardware machine code, will run faster than Java.
- You might also want to use system calls in C/C++ program which you'd need to somehow map out of the JVM and onto the OS proper.
- Even stuff like shared memory is going to require operating outside of the VM.

Why are Virtual Machines necessary?

Compiler takes the high level language and transforms it. This output, however if in the opcode format, can only run on the machine architecture for which it was compiled. This is the best option if **performance** and **efficiency** are wanted.

Virtual machines on the other hand (like JVM) take an intermediate step. This intermediate step (like in Java) is a bytecode format. This form is run on an interpreter/JIT within the virtual machine. The interpreter/JIT then sends the op codes to the processor in response to the bytecode it parses.

- The two main motivations for using a virtual machine are portability and security. The main motivation against is typically performance.
- VM is also helps to reduce **compiler construction costs**. Say for example there are many (N) programing languages and also many (M) hardware platforms exist in the world. If compilers worked without using an intermediate language (VM), the total number of "compilers" that would need to be written to support all languages on all hardware platforms would be **N*M**.

However, by defining an intermediate language and breaking a compiler up into 2 parts, a **front end** and a **back end**, with the front end compiling source code into IL and the back end compiling IL into machine code, you can get away with writing only N+M compilers. This ends up being a **huge cost savings**.

The big difference between CLR / JVM compilers and native code compilers is the way the front end and the back end compilers are linked to each other. In a native code compiler the two components are usually combined into the same executable, and both are run when the programmer hits "build" in the IDE.

With CLR / JVM compilers, the front end and the back end are run at different times. The front end is run at compile time, producing IL that is actually shipped to customers. The back end is then embodied in a separate component that is invoked at runtime.

Advantage and Drawback of VM

Advantage	Drawback
• Provides portability - It becomes possible to ship one set of binaries that can run on multiple hardware platforms • Easy to adopt improvements in the back end compilation technology without being redeployed • More secured – VM completely control the run time environment. The VM creates a secure sandbox that only allows commands with the right security access to perform potentially damaging code - like changing password, or updating an HD bootsector. • The most prominent service offered is garbage collection for free. They manage the memory for you. • Things like bounds checking, access violations (while still possible, they are extremely difficult) are also offered.	• As extensive compiler optimizations can be expensive, "JIT" back ends will often do less optimizations than upfront backends do. This can hurt performance. • Invoking the compiler at runtime increases the time necessary to load programs.

All major browsers vendors are working on a virtual machine for C and C++. It is called **webassembly**. The idea is that you compile your C or C++ program into web assembly and you embed the compiled program in your web application. The browser would take that code, JIT compile it to native code and execute it.

JVM (Java Virtual Machine)

A virtual machine (VM) *is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine*. VM is needed for implementing **WORA** (*Write Once Run Anywhere*).

- JVM (Java Virtual Machine) specially designed for running Java program.
- JVM provides runtime environment in which java **bytecode** (a middle-language between Java (user language) and the machine language) can be executed.

- JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM?	What it does?
• **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies. • **An implementation :** Its implementation is known as **JRE** (Java Runtime Environment). • **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.	The JVM performs following operation: • Loads code • Verifies code • Executes code • Provides runtime environment JVM provides definitions for the: • Memory area • Class file format • Register set • Garbage-collected heap • **Fatal error reporting etc.**

Every developer who uses Java knows that Java bytecode runs in a JRE (Java Runtime Environment). The most important element of the JRE is **Java Virtual Machine** (JVM), which analyzes and executes Java byte code. Java developers do not need to know how JVM works. However, if you understand JVM, you will understand Java more, and will be able to solve the problems which seem to be so simple but unsolvable.

Now we will briefly explain how JVM works, its structure, how it executes Java bytecode, the order of execution etc.

Internal Architecture of JVM

JVM contains **classloader**, **runtime data area** (memory area), **execution engine** etc. as shown in figure. The JVM specification defines the tasks. However, it allows flexible application of the execution time.

Class loader loads the compiled Java Bytecode to the **Runtime Data Areas**, and the execution engine executes the Java Bytecode. Java provides a dynamic load feature; it loads and links the class when it refers to a class for the first time at runtime, not compile time.

An incorrect class file is created because of a Java compiler error. Or a class file can be broken due to errors in network transfer or file copy process.
Java class loader verifies those through a very strict and tight process.

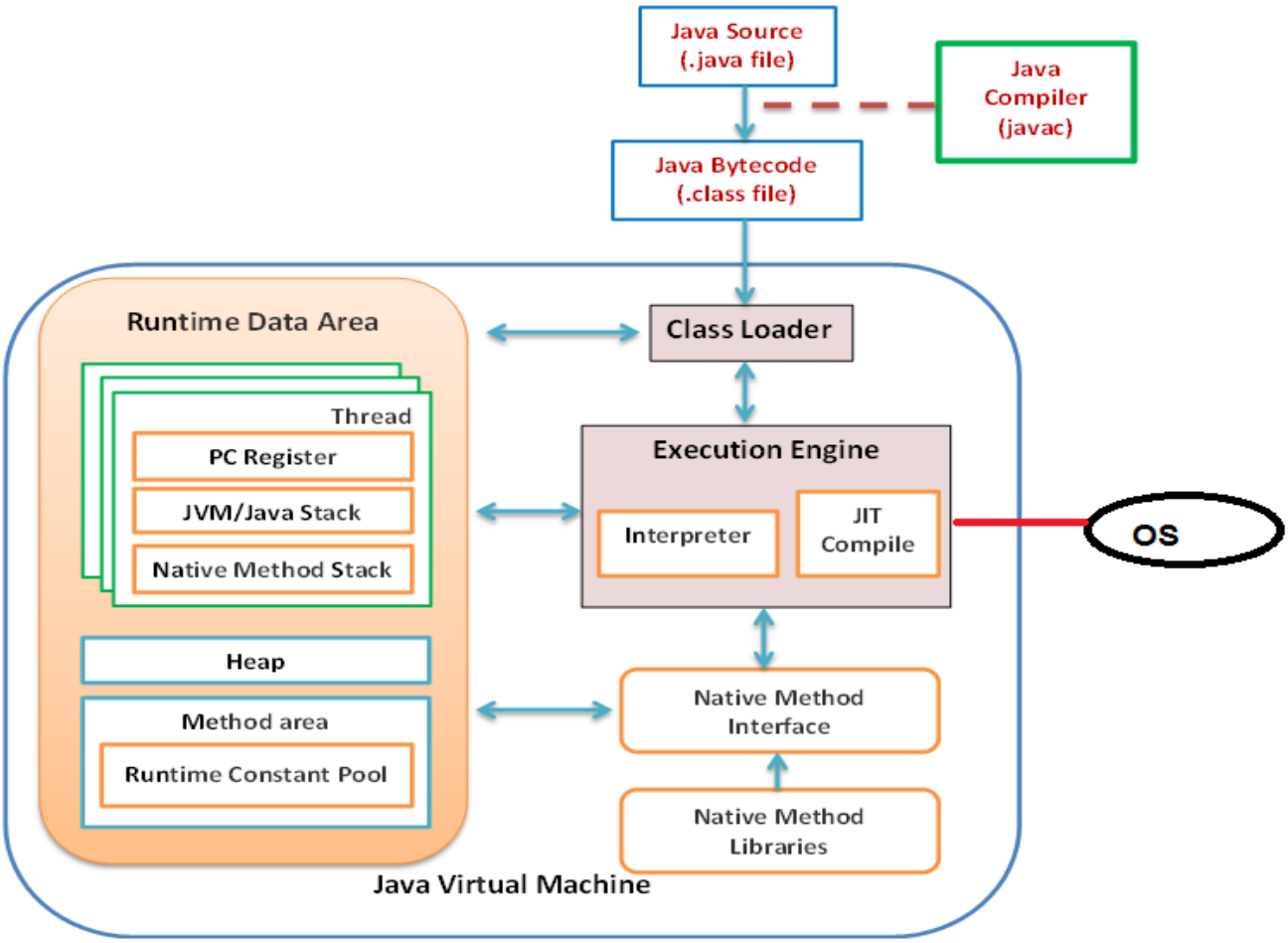


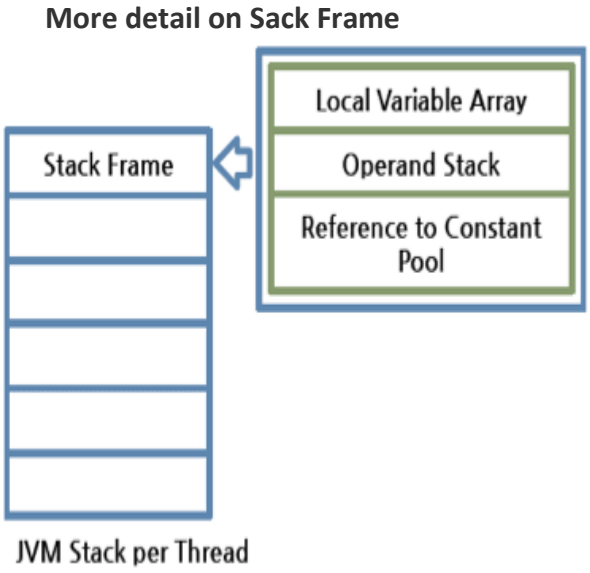
Fig. - Java Code Execution Process

Runtime Data Areas are the memory areas assigned when the JVM runs the program on the OS. The runtime data areas can be divided into **6 areas** and store the data and results while running the program. Of the six, one PC Register, JVM Stack, and Native Method Stack are created for each thread. Heap, Method Area, and Runtime Constant Pool are shared by all threads.

- **PC register**: One PC (Program Counter) register exists for one thread, and is created when the thread starts. PC register has the address of a JVM instruction being executed currently. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.

- **JVM stack (Java Stack)** : Each thread has a private JVM stack, created at the same time as thread starts. While running a method, it needs some memory to store data and results. This memory is allocated on java stack. So Java Stack is the memory area where java methods are executed. While executing methods a separate frame (Stack Frame) will be created in Java Stack, where the method is executed. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

The JVM just pushes or pops the stack frame to the JVM stack. If any exception occurs, a method as printStackTrace() expresses each line of one stack frame.



Each stack frame has the reference for local variable array, Operand stack, and runtime constant pool of a class where the method being executed belongs. The size of local variable array and Operand stack is determined while compiling. Therefore, the size of stack frame is fixed according to the method.

- Local variable array: It has an index starting from 0. 0 is the reference of a class instance where the method belongs. From 1, the parameters sent to the method are saved. After the method parameters, the local variables of the method are saved.
- Operand stack: An actual workspace of a method. Each method exchanges data between the Operand stack and the local variable array, and pushes or pops other method invoke results.

- **Native method stack**: Java methods are executed on Java Stacks. Similarly this stack is used for execution of native code written in a language (for example C/C++ functions) other than Java. To execute the native methods, generally native method library are required (C/C++ codes are invoked through JNI (Java Native Interface)).
- **Heap**: A space that stores instances or objects, and is a target of garbage collection. Whenever JVM loads a class, a method and a heap are immediately created in it.
- **Method area**: The method area is shared by all threads, created when the JVM starts. It stores runtime constant pool, field and method information, static variable, and method bytecode for each of the classes and interfaces read by the JVM.
- **Runtime constant pool**: An area that corresponds to the constant_pool table in the class file format. As well as the constant of each class and interface, it contains all references for methods and fields. In short, when a method or field is referred to, the JVM searches the actual address of the method or field on the memory by using the runtime constant pool.

Execution Engine:

The bytecode that is assigned to the runtime data areas in the JVM via class loader is executed by the execution engine. The execution engine reads the Java Bytecode in the unit of instruction. It is like a CPU executing the machine command one by one. Each command of the bytecode consists of a 1-byte OpCode and additional Operand. The execution engine gets one OpCode and execute task with the Operand, and then executes the next OpCode.

Most of the JVM implementations use both **Interpreter and Just-In-Time(JIT) compiler simultaneously** to convert and execute byte code**.** This technique is also called adaptive optimizer.

- **Interpreter**: Reads, interprets and executes the bytecode instructions one by one. As it interprets and executes instructions one by one, it can quickly interpret one bytecode, but slowly executes the interpreted result. This is the disadvantage of the interpret language.
- **JIT (Just-In-Time) compiler**: The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The execution engine runs as an interpreter first, and at the appropriate time, the JIT compiler compiles the entire bytecode to change it to native code. After that, the execution engine no longer interprets the method, but directly executes using native code. Execution in native code is much faster than interpreting instructions one by one. The compiled code can be executed quickly since the native code is stored in the cache.

Why both are needed and how both work simultaneously? To understand this, let consider a sample code as shown below (assume it is byte code).

```
print a;
print b;
Repeat the following 100 times b changing the value of p
print p;
```

Interpreter will convert the first Instruction to machine code and gives It to cpu for execution.
Say conversion time = 2 ns.
For 2nd Instruction it will also take 2 ns. For next instruction conversion and execution will occur for 100 times, so it will take **200 ns**. This procedure is not efficient in terms of time. That is the reason why JVM does not allocate this code to **Interpreter.** It allocates this code to the **compiler**.

JIT compiler will read the looping instruction and convert it to machine code (say it takes **2 ns**), then it allocates a memory block and push the machine code to that block (say it takes another **2 ns**). So total time needed is **4 ns**. Now the processor will fetch this instruction and execute it. So **JIT compiler** will take **4 ns** when **Interpreter** will take **200** ns.

After loading the .class code into memory, JVM first of all identifies which code is to be left to interpreter and which one to JIT compiler so that he performance is better. The blocks of code allocated to JIT compiler are also called **hotspots**.

How the execution engine runs is not defined in the JVM specifications. Therefore, JVM vendors improve their execution engines using various techniques, and introduce various types of JIT compilers.