**Thread Safety in Java**

Thread safety in java is the process to make our program safe to use in multithreaded environment, there are different ways through which we can make our program thread safe.

* Synchronization is the easiest and most widely used tool for thread safety in java.
* Use of Atomic Wrapper classes from java.util.concurrent.atomic package. For example AtomicInteger
* Use of locks from java.util.concurrent.locks package.
* Using thread safe collection classes, check this post for usage of [ConcurrentHashMap](https://www.journaldev.com/122/concurrenthashmap-in-java) for thread safety.
* Using volatile keyword with variables to make every thread read the data from memory, not read from thread cache.

1. String pool is possible only because String is immutable in Java. This way Java Runtime saves a lot of [heap space](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory) because different String variables can refer to the same String variable in the pool.
2. If String is not immutable then it would cause a severe security threat to the application. For example, database username, password are passed as String to get database connection and in [socket programming](https://www.journaldev.com/741/java-socket-programming-server-client) host and port details passed as String. Since String is immutable, its value can’t be changed otherwise any hacker could change the referenced value to cause security issues in the application.
3. Since String is immutable, it is safe for [multithreading](https://www.journaldev.com/1079/multithreading-in-java). A single String instance can be shared across different threads. This avoids the use of synchronization for thread safety. Strings are implicitly thread-safe.
4. Since String is immutable, its **hashcode** is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for the key in a Map and its processing is faster than other HashMap key objects. This is why String is the most widely used as HashMap keys.

**Immutable Class in Java**An immutable class is good for caching purpose because you don’t need to worry about the value changes. Other benefit of immutable class is that it is inherently [**thread-safe**](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t need to worry about thread safety in case of multi-threaded environment.

<https://www.journaldev.com/129/how-to-create-immutable-class-in-java>

**Deep VS Shallow Copy:**

<https://www.baeldung.com/java-deep-copy>

**What is meant by the statement Java is platform independent?**

Java works on the principle of write once and run anywhere. Once a Java program is written, it gets compiled into what is known as the byte code, which can then be run on any Java Virtual Machine or JVM for short.

**What is the Java language specification?**

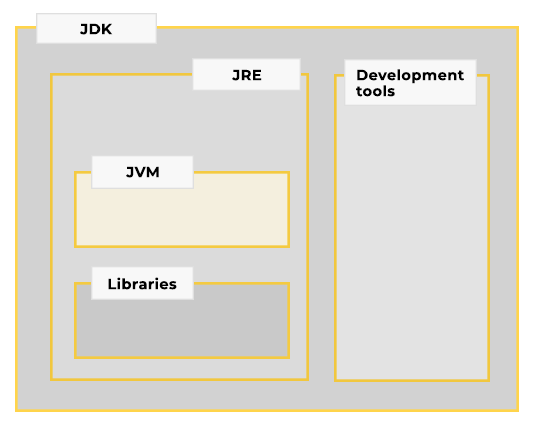
In general, language specification is an official document that gives detailed description of the syntax and semantics of a particular language. Java publishes Java language specifications and the virtual machine specifications for various editions. Note the following:

**What is the Java Runtime Environment?**

The Java Runtime Environment (JRE) includes the Java Virtual Machine and the standard Java APIs (core classes and supporting files.). The JRE contains just enough to execute or run a Java application, but not to compile it.

What is the Java Development Kit or JDK?

The Java Software Development Kit (Java SDK) is JRE plus the Java compiler, and a set of other tools.





For those who don't know Java is both compiler and interpreter language. When you compile a Java program it creates .class file which is collection of byte code, these byte code are not machine instruction instead they are instruction which Java virtual machine can understand. Since every Java program runs on Java virtual machine, same byte code can be run on any platform. key is byte code is not machine instruction they are platform independent instruction to JVM. On the other hand JVM or Java virtual machine is platform dependent because it converts byte code into machine level instruction which is platform specific and that's why you have different version of JDK and JRE for windows and Linux because both JDK and JRE comes with Java virtual machine.

**Is it possible to execute a Java programme without using JDK?**

Yes, you can execute Java program with out JDK. But to do that you need JVM. JDK is basically used to compile your code. Once you have .class file of your .java program you don’t need JDK. You simply run your code with JVM.

Lets see cmd commands that we use with java

javac MyProgram.java (Used to compile code and need JDK which generated .class file you can delete .java file if you don’t need to edit the code)

java MyProgram (Used to run your program that is complied in .class file)

**What is byteCode?**

Bytecode is [program](https://techterms.com/definition/program) code that has been [compiled](https://techterms.com/definition/compile) from [source code](https://techterms.com/definition/sourcecode) into low-level code designed for a software [interpreter](https://techterms.com/definition/interpreter). It may be executed by a virtual machine (such as a [JVM](https://techterms.com/definition/jvm)) or further compiled into machine code, which is recognized by the [processor](https://techterms.com/definition/processor).

Different types of bytecode use different [syntax](https://techterms.com/definition/syntax), which can be read and executed by the corresponding virtual machine. A popular example is Java bytecode, which is compiled from [Java](https://techterms.com/definition/java) source code and can be run on a Java Virtual Machine (JVM). Below are examples of Java bytecode instructions.

new (create new object)

aload\_0 (load reference)

istore (store [integer](https://techterms.com/definition/integer) value)

ladd (add [long](https://techterms.com/definition/long) value)

swap (swap two values)

areturn (return value from a function)

While it is possible to write bytecode directly, it is much more difficult than writing code in a high-level language, like Java. Therefore, bytecode files, such as Java [.CLASS](https://fileinfo.com/extension/class) files, are most often generated from source code using a compiler, like javac.

**Does Java's platform independence affect its performance?**

As a platform-independent environment, the Java platform can be a bit slower than native code. Converting byte code to machine code incurs a performance penalty. Additionally, Java is a memory managed language which requires cleaning up of the memory space periodically by the platform, resulting in reduced application throughput. However, advancements in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

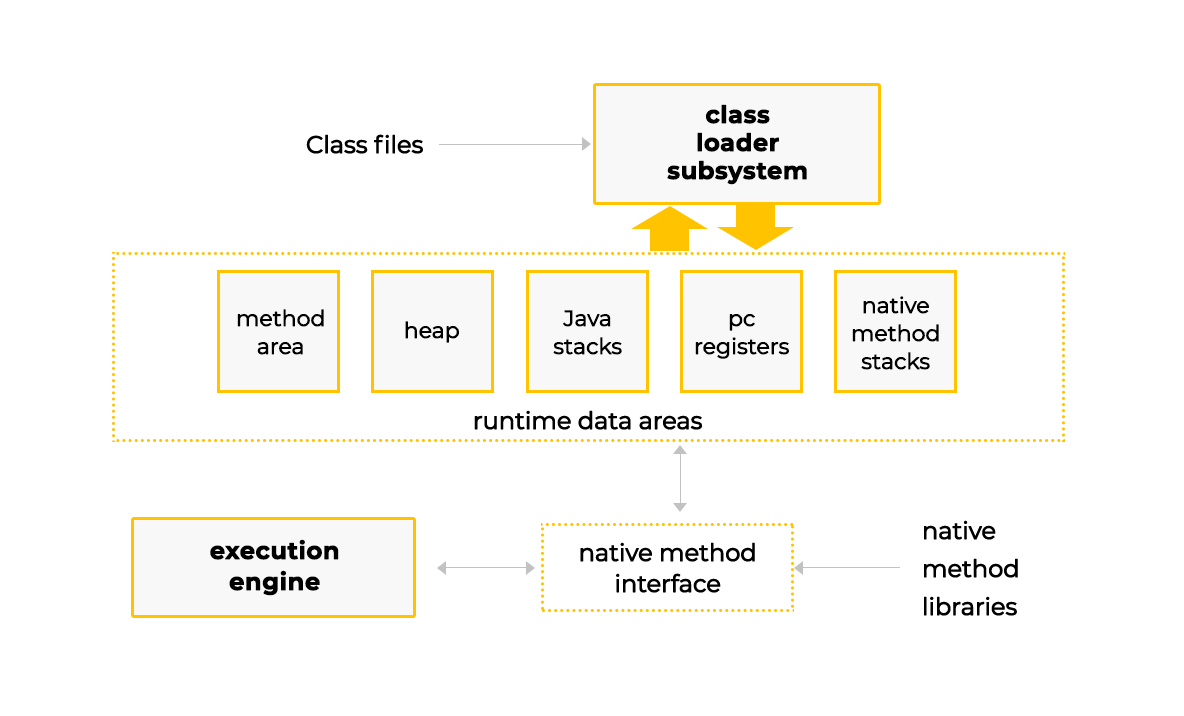
**Describe JVM architecture?**

The Java Virtual Machine consists of three components:

**Class Loader Subsystem:** The part of a Java virtual machine implementation that takes care of finding and loading types is called the class loader subsystem. The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references.

**Runtime Data Area:** The memory areas allocated by the JVM are called Runtime Data Area. These consist of method area, heap area, stack, pc registers and native stack.

**Execution Engine:** The execution is responsible for the actual execution of the bytecode. It consists of three components: interpreter, just-in-time compiler and the garbage collector.



If we launch two Java programs on the same machine, how many instances of JVM would be created?

A runtime instance of the Java virtual machine runs a single Java application. **When a Java application starts, a runtime instance is born. When the application completes, the instance dies.** If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

**What are JVM languages?**

JVM runs bytecode. The Java compiler converts code written in the Java language to bytecode. Similarly, other languages can take advantage of the Java platform if they have a compiler to convert code written in their respective languages to bytecode. For instance, [**Jython**](http://www.jython.org/) is an implementation of the Python language for the Java platform and a program written in Jython can run on any Java platform. [**JRuby**](https://www.jruby.org/) is another example, which is an implementation of the Ruby programming language atop the JVM.

What is Java Hotspot?

HotSpot is one of the most popular implementations of the JVM concept. **There are two editions available:**

**Oracle Hotspot**

**OpenJDK Hotspot**

Sun open sourced and donated the Hotspot JVM source code which became the OpenJDK project. Implementations by OpenJDK serve as reference implementation.

There's not too many differences between the two as Oracle's Hotspot is based on OpenJDK's Hotspot project and comes with additional bells and whistles for paying customers. The JVM implementation is called HotSpot because it continually analyzes the program's performance for hot spots of code (code paths which are executed repeatedly). The repeating code paths are compiled into very highly optimized native machine code for faster execution.

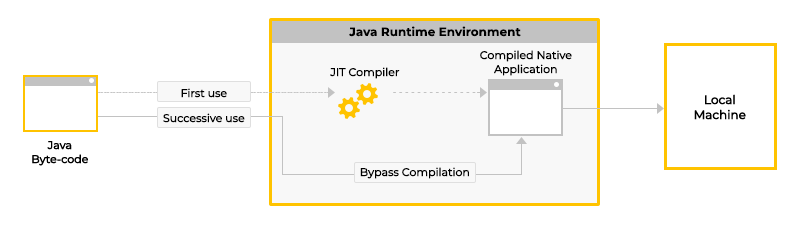
**What is the Java interpreter?**

The Java Interpreter can be thought of as a translator that converts Java bytecode into native machine code. The translation of bytecodes to native machine code is done line by line.

For each hardware architecture, a different Java bytecode interpreter is needed. When Java bytecode is executed by an interpreter, the execution will always be slower than the execution of the same program compiled into native machine language. This problem is mitigated by just-in-time (JIT) compilers for executing Java bytecode.

**Explain the working of the JIT compiler?**

A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. A JIT compiler has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently. This is in contrast to a traditional compiler that compiles all the code to machine language before the program is run for the first time.



Usually JIT compilers employ a sophisticated, low-cost, sampling-based technique to identify which functions merit optimization. A "sampler thread" wakes up at periodic intervals and checks the status of several application threads. It identifies what each thread is executing and notes some of the execution history. This information is tracked for all the methods and when it is perceived that a method is experiencing heavy use or in other words, becomes *hot* — that method is slated for optimization. Usually, a flurry of such optimization opportunities occurs in the application’s early run stages, with the rate slowing down as execution continues.

**What is the execution engine?**

The execution engine is responsible for executing bytecode. The execution engine is one part of the virtual machines that can vary in different JVM implementations. The most used JVMs have three components of the execution engine:

* Interpreter
* Just in Time Compiler
* Garbage Collector

The simplest kind of execution engine just interprets the bytecodes one at a time.

Another kind of execution engine, one that is faster but requires more memory, comes with a just-in-time compiler. In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked.

The third type of execution engine is an adaptive optimizer. In this approach, the virtual machine starts by interpreting bytecodes, but monitors the activity of the running program and identifies the most heavily used areas of code. As the program runs, the virtual machine compiles to native and optimizes just these heavily used areas. The rest of the of code, which is not heavily used, remains as bytecodes which the virtual machine continues to interpret.

**What are .jar files?**

A JAR (Java Archive) is a package file format used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. A jar file is built on the ZIP format and typically has a .jar file extension. The jar tool can be used to create .jar file.***Fat jar,* or *uber jar,* is a jar which contains all project class files and classes from all the libraries, on which the project depends.**

How can we pass multiple or variable number of arguments to a method on each invocation call?

We can pass variable number of arguments to a method using varargs feature. Below is an example of passing multiple arguments of the same type to a method.

**public void childrenNames(String... names) {  
        for (int i = 0; i < names.length; i++)  
            System.out.println(names[i]);  
    }**

The type name is followed by three dots, a space, and then the variable name.

The varargs variable is treated like an array.

The varargs variable must appear at the last in the method signature.

As a consequence of the above, there can only be a single varargs in a method signature.

**childrenNames();  
        childrenNames("jane");          
        childrenNames("jane", "tom", "peter");**

**Is Java both pass by reference and pass by value?**

**Java is pass by value only. Even reference data types are passed by value. Remember Java's method passing always work as pass by value!**

Passing by reference doesn't apply to Java! Reference data type parameters, such as objects, are also passed into methods by value. This means that when the method returns, *the passed-in reference still references the same object as before*.

**public class SuperList {  
    // Constructor  
    public SuperList(int n) {  
1.       List<Integer> superList;  
2.       allocate(superList, n);  
    }  
    // Method that does initialization  
    void allocate(List<Integer> list, int n) {  
3.       list = new ArrayList<>(n);  
    }  
}**

What should happen when we initialize an object of class SuperList? It will be null, which may seem counterintuitive.

* Consider superList to be a holder that will hold a value of null on line 1.
* On line 2, we are passing a value of null and not the variable superList itself. This is a very important distinction to realize.
* When program control, reaches line 3, the list variable is not the variable superList. In fact, it's a brand-new variable (holder) which receives a value of null.
* Line 3 also initializes the list variable to an object of ArrayList and the list variable will hold the reference or the address of the ArrayList object in the memory(heap).
* When the program control returns to line 2, superList is still null because it was never passed in and assigned the ArrayList object.

class IntegerSwap {

    public static void main( String args[] ) {

        (new IntegerSwap()).run();

    }

   public void run() {

        Integer x = 5;

        Integer y = 9;

        System.out.println("Before Swap x: " + x + " y: " + y);

        swap(x, y);

        System.out.println("After Swap x: " + x + " y: " + y);

    }

    private void swap(Integer a, Integer b) {

        Integer temp = a;

        a = b;

        b = temp;

    }

}

* Look at this image : 🡪 C:\Users\do936e\Pictures\index.svg

# Method Overloading

Methods of a class can be overloaded in Java by:

* Changing the number of parameters
* Changing the type of the parameters passed into the methods

Note that methods can't be overloaded by changing the return types of the methods, as it may cause ambiguity. While **overloading has nothing to do with polymorphism**, Java programmers also refer to method overloading as **Compile Time Polymorphism** because the method that is going to get called will be decided at compile time.

**Can the main method be overloaded?**

Yes, the static main method can be overloaded. But only public static void main(String[] args) will be used when your class is launched by the JVM even if you specify one or two command-line arguments. However, programmatically one can invoke the overloaded versions of the main method.

class Demonstration {

    public static void main( String args[] ) {

        System.out.println( "Traditional main method" );

        }

    public static void main( String singleArg) {

        System.out.println( "Method with single arg" );

    }

    public static void main( ) {

        System.out.println( "Method with no args" );

    }

}

###### Output

###### Traditional main method