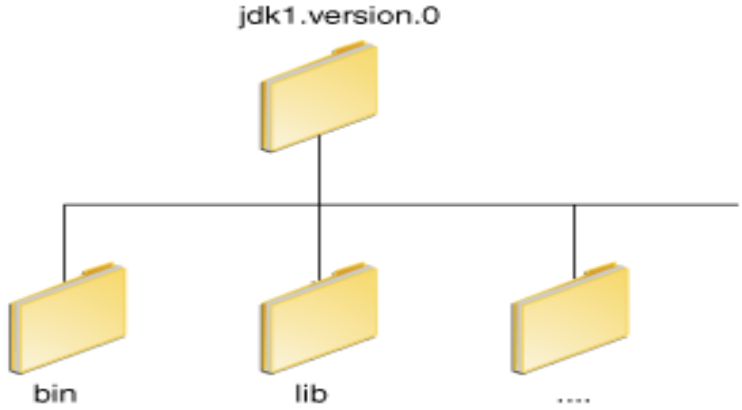
PATH and CLASSPATH

This section explains how to use the **PATH** and **CLASSPATH** environment variables on Microsoft Windows, Solaris, and Linux.

After installing the software, the JDK directory will have the structure shown below.



The bin directory contains both the compiler and the launcher.

Update the PATH Environment Variable (Microsoft Windows)

You can run Java applications just fine without setting the PATH environment variable. Or, you can optionally set it as a convenience.

Set the PATH environment variable if you want to be able to conveniently run the executables (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command. If you do not set the PATH variable, you need to specify the full path to the executable every time you run it, such as:
C:\Java\jdk1.7.0\bin\javac MyClass.java

The PATH environment variable is a series of directories separated by semicolons (;). Microsoft Windows looks for programs in the PATH directories in order, from left to right. You should have only one bin directory for the JDK in the path at a time (those following the first are ignored), so if one is already present, you can update that particular entry.

The following is an example of a PATH environment variable:
C:\Java\jdk1.7.0\bin;C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem

It is useful to set the PATH environment variable permanently so it will persist after rebooting. To make a permanent change to the PATH variable, use the **System** icon in the Control Panel. The precise procedure varies depending on the version of Windows:

Windows 7:

1. From the desktop, right click the **Computer** icon.
2. Choose **Properties** from the context menu.
3. Click the **Advanced system settings** link.
4. Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable and select it. Click **Edit**. If the PATH environment variable does not exist, click New.
5. In the **Edit System Variable** (or **New System Variable**) window, specify the value of the PATH environment variable. Click **OK**. Close all remaining windows by clicking **OK**.

Note: You may see a PATH environment variable similar to the following when editing it from the Control Panel:
%JAVA_HOME%\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem

Variables enclosed in percentage signs (%) are existing environment variables. If one of these variables is listed in the **Environment Variables** window from the Control Panel (such as JAVA_HOME), then you can edit its value. If it does not appear, then it is a special environment variable that the operating system has defined. For example, SystemRoot is the location of the Microsoft Windows system folder. To obtain the value of a environment variable, enter the following at a command prompt. (This example obtains the value of the SystemRoot environment variable):

```
echo %SystemRoot%
```

Checking the CLASSPATH variable (All platforms)

The CLASSPATH variable is one way to tell applications, including the JDK tools, where to look for user classes. (Classes that are part of the JRE, JDK platform, and extensions should be defined through other means, such as the bootstrap class path or the extensions directory.)

The preferred way to specify the class path is by using the -cp command line switch. This allows the CLASSPATH to be set individually for each application without affecting other applications. *Setting the CLASSPATH can be tricky and should be performed with care.*

The default value of the class path is ".", meaning that only the current directory is searched. Specifying either the CLASSPATH variable or the -cp command line switch overrides this value.

To check whether CLASSPATH is set on Microsoft Windows NT/2000/XP, execute the following:
C:> echo %CLASSPATH%

On Solaris or Linux, execute the following:
% echo $CLASSPATH

If CLASSPATH is not set you will get a **CLASSPATH: Undefined variable** error (Solaris or Linux) or simply **%CLASSPATH%** (Microsoft Windows NT/2000/XP).

To modify the CLASSPATH, use the same procedure you used for the PATH variable.
Class path wildcards allow you to include an entire directory of .jar files in the class path without explicitly naming them individually. For more information, including an explanation of class path wildcards, and a detailed description on how to clean up the CLASSPATH environment variable, see the Setting the Class Path technical note.

Difference between path and classpath in Java

Path

Path variable is set for providing path for all Java tools like java, javac, javap, javah, jar, appletviewer. In Java to run any program we use **java** tool and for compile Java code use **javac** tool. All these tools are available in bin folder so we set path upto bin folder.

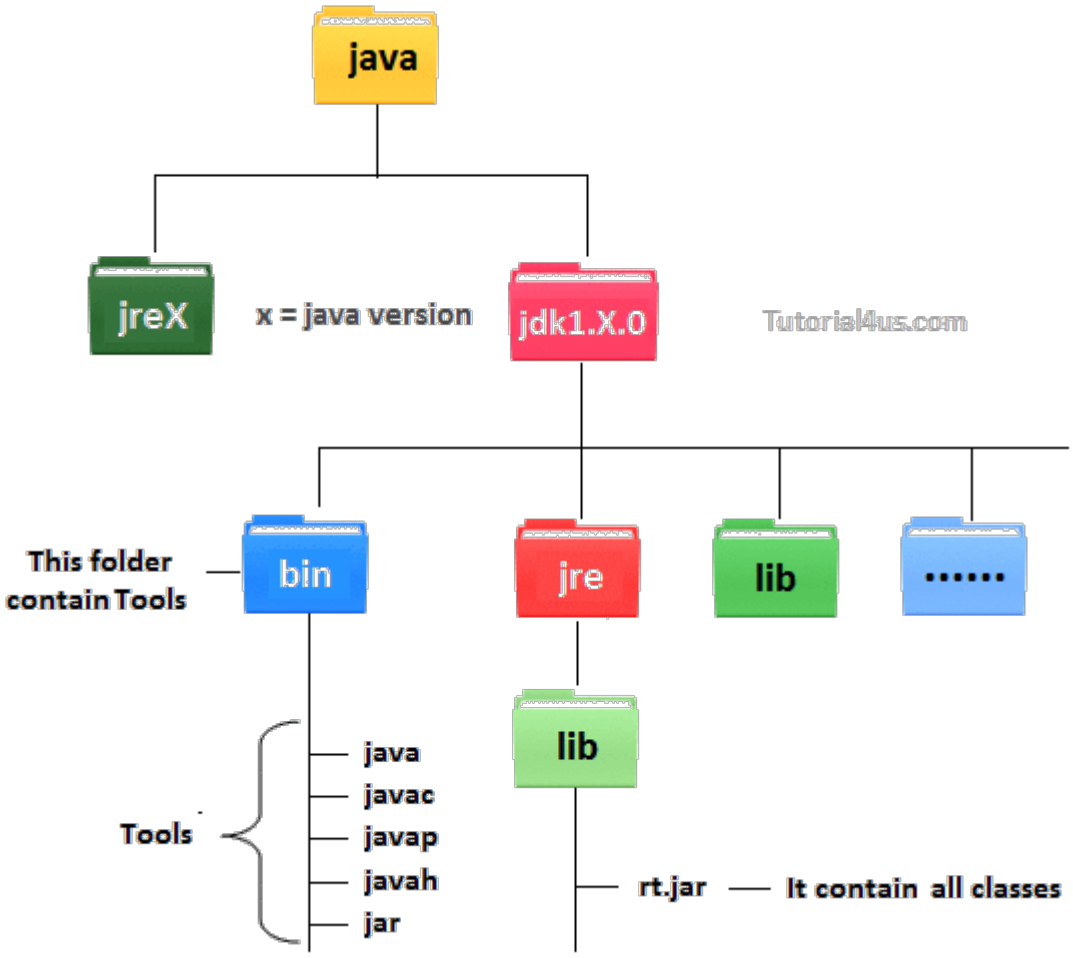
classpath

classpath variable is set for providing path of all Java classes which is used in our application. All classes are available in lib/rt.jar so we set classpath upto lib/rt.jar.

Difference between path and classPath

path	classpath
path variable is set for providing path for all java tools like java, javac, javap, javah, jar, applet-viewer	classpath variable is set for provide path of all java classes which is used in our application.

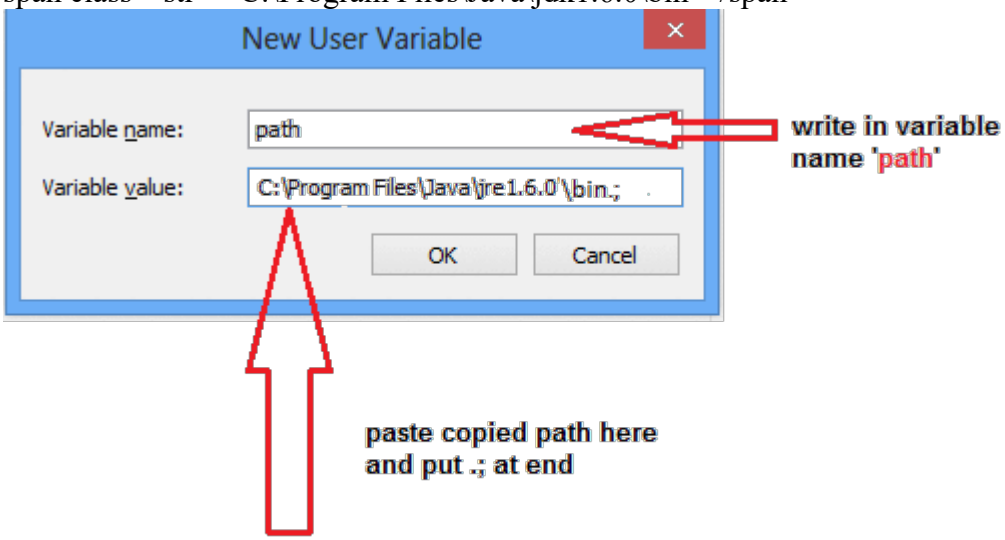
JDK Folder Hierarchy



Path variable is set for use all the tools like java, javac, javap, javah, jar, appletviewer etc.

Example

span class="str">"C:\Program Files\Java\jdk1.6.0\bin"



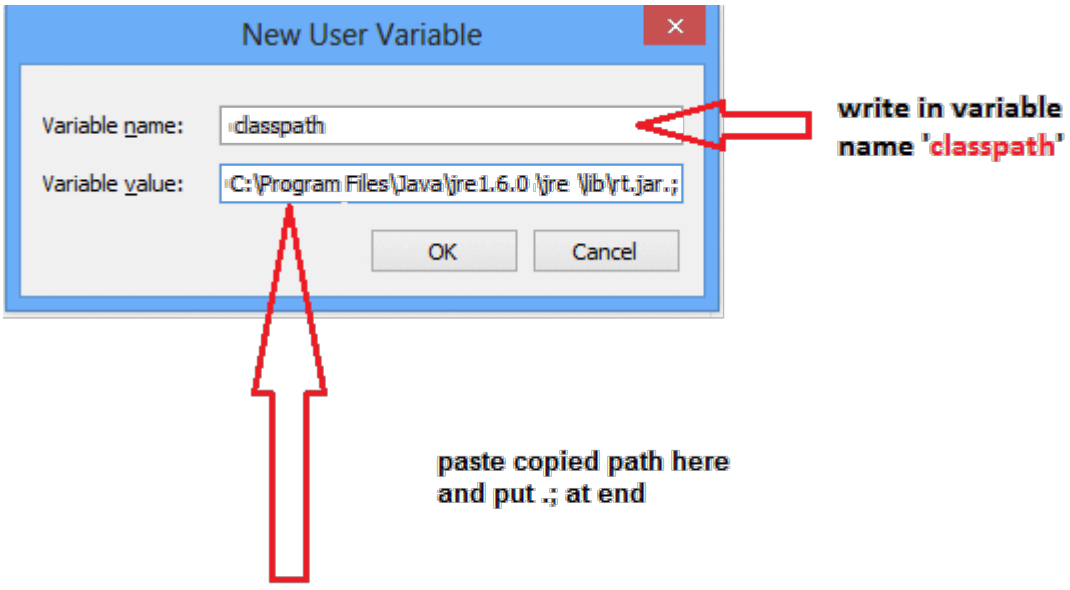
C\Program Files\java\jre\1.6.0\bin.;

All the tools are present in bin folder so we set path upto bin folder.

Classpath variable is used to set the path for all classes which is used in our program so we set classpath upto rj.jar. in rt.jar file all the .class files are present. When we decompressed rt.jar file we get all .class files.

Example

span class="str">"C:\Program Files\Java\jre1.6.0\jre\lib\rt.jar"
In above rt.jar is a jar file where all the .class files are present so we set the classpath upto rt.jar.



C\Program Files\java\jre1.6.0\jre\lib\rt.jar.;

What is difference between JDK, JRE and JVM?

What is JDK, JRE and JVM?

JDK:- Java Development Kit (in short JDK) is Kit which provides the environment to Develop and execute(run) the Java program. For eg. You(as Java Developer) are developing an accounting application on your machine, so what do you going to need into your machine to develop and run this desktop app? You are going to need **J-D-K** for that purpose for this you just need to go to official website of sun or oracle to download the latest version of JDK into your machine.

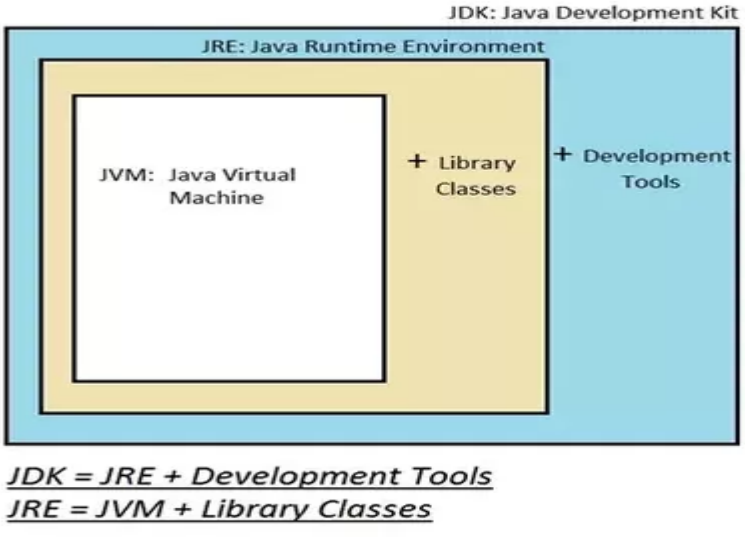
Hence, JDK is a kit(or package) which includes two things **i) Development Tools(to provide an environment to develop your java programs) and ii) JRE (to execute your java program)**. **JDK is only used by Java Developers.**

JRE :- Java Runtime Environment (to say JRE) is an installation package which provides environment to only run(not develop) the java program(or application)onto your machine. For eg(continuing with the same example) after developing your accounting application , you want to run this application into your client's machine . Now in this case your client only need to run your application into his/her machine so your client should install JRE in-order to run your application into his machine.

Hence, JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.

JVM :- Java Virtual machine(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever java program you run using JRE or JDK goes into JVM and **JVM is responsible to execute the java program line by line hence it is also known as interpreter**(we will discuss about interpreter later) . Hence you don't need to install JVM separately into your machine because it is inbuilt into your JDK or JRE installation package. We'll explore more about JVM soon.

Finally after learning about all the three main parts of java you can have a look at the above figure to have clear understanding of the architecture and interrelationship between all the main components of java.



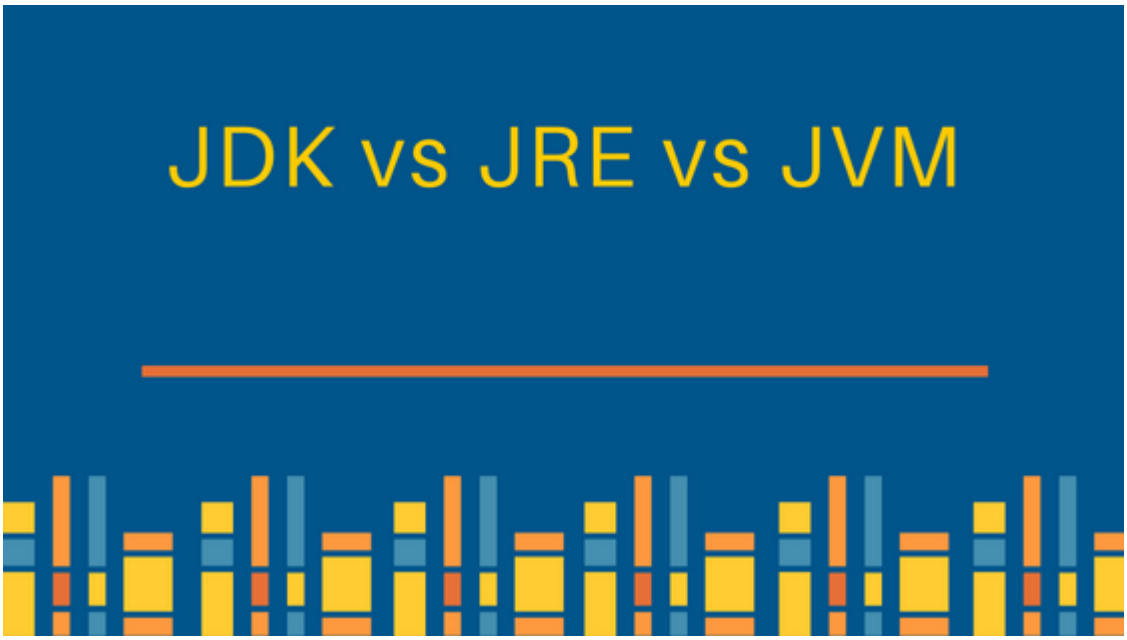
Difference between JDK, JRE and JVM

April 2, 2018 by Pankaj 15 Comments

Difference between JDK and JRE and JVM is one of the popular interview questions. You might also be asked to explain JDK vs JRE vs JVM.

Table of Contents [hide]

Difference between JDK and JRE and JVM



JDK, JRE and JVM are core concepts of Java programming language. Although they all look similar and as a programmer we don't care about these concepts a lot, but they are different and meant for specific purposes. It's one of the common java interview questions and this article will explain each one of these and what is the difference between them.

JDK

Java Development Kit is the core component of Java Environment and provides all the tools, executables and binaries required to compile, debug and execute a Java Program. JDK is a platform specific software and thats why we have separate installers for Windows, Mac and Unix systems. We can say that JDK is superset of JRE since it contains JRE with Java compiler, debugger and core classes. Current version of JDK is 1.7 also known as Java 7.

JVM

JVM is the heart of java programming language. When we run a program, JVM is responsible to converting Byte code to the machine specific code. JVM is also platform dependent and provides core java functions like memory management, garbage collection, security etc. JVM is customizable and we can

use java options to customize it, for example allocating minimum and maximum memory to JVM. JVM is called *virtual* because it provides a interface that does not depend on the underlying operating system and machine hardware. This independence from hardware and operating system is what makes java program write-once run-anywhere.

JRE

JRE is the implementation of JVM, it provides platform to execute java programs. JRE consists of JVM and java binaries and other classes to execute any program successfully. JRE doesn't contain any development tools like java compiler, debugger etc. If you want to execute any java program, you should have JRE installed but we don't need JDK for running any java program.

JDK vs JRE vs JVM

Let's look at some of the important difference between JDK, JRE and JVM.

1. JDK is for development purpose whereas JRE is for running the java programs.
2. JDK and JRE both contains JVM so that we can run our java program.
3. JVM is the heart of java programming language and provides platform independence.

Just-in-time Compiler (JIT)

Sometimes we heard this term and being it a part of JVM it confuses us. JIT is part of JVM that optimise byte code to machine specific language compilation by compiling similar byte codes at same time, hence reducing overall time taken for compilation of byte code to machine specific language.

The Just-In-Time (JIT) compiler

The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment. It improves the performance of Java applications by compiling bytecodes to native machine code at run time. This section summarizes the relationship between the JVM and the JIT compiler and gives a short description of how the compiler works.

- JIT compiler overview
  The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time.

- How the JIT compiler optimizes code
  When a method is chosen for compilation, the JVM feeds its bytecodes to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecodes before it can compile the method correctly.

- Frequently asked questions about the JIT compiler
  Examples of subjects that have answers in this section include disabling the JIT compiler, use of alternative JIT compilers, control of JIT compilation and dynamic control of the JIT compiler.

- JIT compiler overview

The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment that improves the performance of Java applications at run time.

Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly

than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.
JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.

After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached. The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler. The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations.

The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems.

- How the JIT compiler optimizes code

When a method is chosen for compilation, the JVM feeds its bytecodes to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecodes before it can compile the method correctly.

To help the JIT compiler analyze the method, its bytecodes are first reformulated in an internal representation called *trees*, which resembles machine code more closely than bytecodes. Analysis and optimizations are then performed on the trees of the method. At the end, the trees are translated into native code. The remainder of this section provides a brief overview of the phases of JIT compilation. For more information, see JIT and AOT problem determination.

The JIT compiler can use more than one compilation thread to perform JIT compilation tasks. Using multiple threads can potentially help Java™ applications to start faster. In practice, multiple JIT compilation threads show performance improvements only where there are unused processing cores in the system.

The default number of compilation threads is identified by the JVM, and is dependent on the system configuration. If the resulting number of threads is not optimum, you can override the JVM decision by

using the -XcompilationThreads option. For information on using this option, see JIT and AOT command-line options.
Note If your system does not have unused processing cores, increasing the number of compilation threads is unlikely to produce a performance improvement.
The compilation consists of the following phases:
1. Inlining
2. Local optimizations
3. Control flow optimizations
4. Global optimizations
5. Native code generation

All phases except native code generation are cross-platform code.

- Phase 1 - inlining
  Inlining is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.

- Phase 2 - local optimizations
  Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers.

- Phase 3 - control flow optimizations
  Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.

- Phase 4 - global optimizations
  Global optimizations work on the entire method at once. They are more "expensive", requiring larger amounts of compilation time, but can provide a great increase in performance.

- Phase 5 - native code generation
  Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture characteristics.

- Frequently asked questions about the JIT compiler

Examples of subjects that have answers in this section include disabling the JIT compiler, use of alternative JIT compilers, control of JIT compilation and dynamic control of the JIT compiler.

Can I disable the JIT compiler?
Yes. The JIT compiler is turned on by default, but you can turn it off with the appropriate command-line parameter. For more information, see Disabling the JIT or AOT compiler.

Can I use another vendor's JIT compiler?
No.

Can I use any version of the JIT compiler with the JVM?
No. The two are tightly coupled. You must use the version of the JIT compiler that comes with the JVM package that you use.

Can the JIT compiler "decompile" methods?

No. After a method is compiled by the JIT compiler, the native code is used instead for the remainder of the execution of the program. An exception to this rule is a method in a class that was loaded with a custom (user-written) class loader, which has since been unloaded (garbage-collected). In fact, when a class loader is garbage-collected, the compiled methods of all classes that are loaded by that class loader are discarded.

Can I control the JIT compilation?
Yes. For more information, see JIT and AOT problem determination. In addition, advanced diagnostic settings are available to IBM® engineers.

Can I dynamically control the JIT compiler?
No. You can pass options to the JIT compiler to modify the behavior, but only at JVM startup time, because the JIT compiler is started up at the same time as the JVM. However, a Java™ program can use the java.lang.Compiler API to enable and disable the JIT compiler at run time.

How much memory does the code cache consume?
The JIT compiler uses memory intelligently. When the code cache is initialized, it consumes relatively little memory. As more methods are compiled into native code, the code cache grows dynamically to accommodate the needs of the program. Space that is previously occupied by discarded or recompiled methods is reclaimed and reused. When the size of the code cache reaches a predefined maximum limit, it stops growing. At this point, the JIT compiler stops compiling methods to avoid exhausting the system memory and affecting the stability of the application or the operating system.

Stack and heap are two important concepts you should understand in relation to Java memory allocation. Let's take a look at the two concepts, why they matter, and when you should use each.

What Is Java Stack?

A Java stack is part of your computer's memory where temporary variables, which are created by all functions you do, are stored. It is used to execute a thread and may have certain short-lived values as well as references to other objects. It uses LIFO data structure, or last in first out.

What does this mean? When a method is invoked, it creates a new block in the stack for that particular method. The new block will have all the local values, as well as references to other objects that are being used by the method. When the method ends, the new block will be erased and will be available for use by the next method. The objects you find here are only accessible to that particular function and will not live beyond it.

This makes it very easy to keep track of the stack, where the latest reserved block is also the first to be freed. The variables created for the method are directly stored in the memory, allowing for fast access. The memory size of a Java stack is generally much less than in a Java heap space because when a method ends, all the variables created on the stack are erased forever.

Here's an example of how to create an object in the stack:

```
void somefunction( ) {
   /* create an object "m" of class Member
   this will be put on the stack since the
   "new" keyword is not used, and we are
   creating the object inside a function
   */

   Member m;
} //the object "m" is destroyed once the function ends
```

What Is Java Heap?

Java objects are in an area, which is called the heap. It is created when the program is run, and its size may decrease or increase as your program runs. It can easily become full, and when it does, garbage collection is initiated. This is when objects that are no longer used are deleted to make way for new objects.

Unlike in a Java stack where memory allocation is done when your program is compiled, in a heap it is allocated as your program is run. Accessing variables placed here is a bit slower compared to a stack's direct and fast access.

Heap is likened to a global memory pool. A method or function will use the heap for memory allocation if you need the data or variables to live longer than the method or function in question. The objects you find here are accessible to all the functions.

Also, there is no specific order in reserving blocks in a heap. You can allocate blocks at any time, and then you can free it when you wish. As you can imagine, it is much more complex to keep track of the parts that are free and can be allocated, but it can also be divided into two generations or sub-areas.

These sub-areas are called the young space (or nursery) and the old space. The young space is typically earmarked for the memory allocation for new objects. When the young space becomes full, garbage collection happens. Short-lived or temporary objects typically use the young space. This help makes garbage collection faster when compared to a heap without any divisions.

Here's an example of how to create an object in the heap:

```
void somefunction( ) {
   /* create an object "m" of class Member
     this will be put on the heap since the
     "new" keyword is used, and we are
     creating the object inside a function
   */

   Member* m = new Member( );

   /* the object "m" must be deleted
     otherwise a memory leak occurs
   */

   delete m;
}
```

Similarities and Differences Between Stack and Heap

Both are ways that Java allocates memory and both are stored in the RAM. However, to make things easier to remember, heap is used for dynamic memory allocation, while stack is for static allocations.

Where is it stored? Variables that are allocated on the stack are accessible directly from memory, and as such, these can run very fast. Accessing objects on the heap, on the other hand, takes more time.

When does the allocation happen? On the stack, memory allocation happens when the program is compiled. Meanwhile, on the heap, it begins when the program is run.

And since this is the case, you would need to know just how much data and memory you are going to need before compiling if you want to use the stack. Another limitation that the stack has is that it cannot handle big chunks of variables that need a lot of memory. If you do not know how much data you are going to need at run time or if you need memory for a lot of data, then you need to use heap.

In a Nutshell…

Stack

- The size of the stack will vary as methods and functions create and delete local variables as needed.
- Memory is allocated and then subsequently freed without you needing to manage the memory allocation.
- Stack has size limits, which can vary according to the operating system you use.
- Variables that are stored on the stack exist for as long as the function that created them are running.

Heap

- Memory is not managed automatically nor is it as tightly managed by the central processing unit the

way stack is managed. You would need to free allocated memory yourself when these blocks are no longer needed.
- The heap is prone to memory leaks, where memory is allocated to unused objects and will not be available to processes other than that.
- There is no size limit in the heap.
- Compared to stack, objects in the heap are much slower to access. It is also slower to write to the memory on the heap.

Stack is easier and faster to use, but it comes with a lot of limitations that you can ignore if you use heap.

When do you use stack? Stack can only be used for local variables that use up small amounts of memory. The good news is that memory allocation and management is not going to be your problem and access to these objects is very fast. It does suffer from size limitations and the fact that you cannot resize variables on the stack.

When do you use heap? You use the heap to allocate memory if there are variables that you need to be accessed globally, as opposed to just being available only to the methods and functions that created it. Heap is also good when you have a need for a lot of memory since it has no limit on memory size. You can also resize the variables on the heap.

Additional Resources and Tutorials

To learn more about the differences between stack and heap, and the best use cases for each, visit the following resources and tutorials:
- Memory: Stack vs. Heap
- What is the difference between the stack and the heap?
- Stack vs. Heap Allocation
- C# Heap(ing) Vs Stack(ing) in .NET: Part I
-

Stack and heap are two ways Java allocates memory. Heap is better in instances in which you have variables requiring global access, while stack is your go-to for local variables requiring only small amounts of memory. Understanding when and how to use a stack and a heap is critical for developing better Java programs.

It's also helpful to understand how memory allocation works when dealing with memory leaks. For a comprehensive guide to all the tools, websites, blogs, and other resources you need to level-up your Java game, download our Comprehensive Java Developer's Guide.

14.1.8 Stack vs Heap Allocation

We conclude our discussion of storage class and scope by breifly describing how the memory of the computer is organized for a running program. When a program is loaded into memory, it is organized into three areas of memory, called ***segments*****:** the ***text segment*****,** ***stack segment***, and ***heap segment***. The text segment (sometimes also called the code segment) is where the compiled code of the program itself resides. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system.

The remaining two areas of system memory is where storage may be allocated by the compiler for data storage. The stack is where memory is allocated for automatic variables within functions. A stack is a *Last In First Out* (LIFO) storage device where new storage is allocated and deallocated at only one

``end", called the Top of the stack. This can be seen in Figure 14.13.

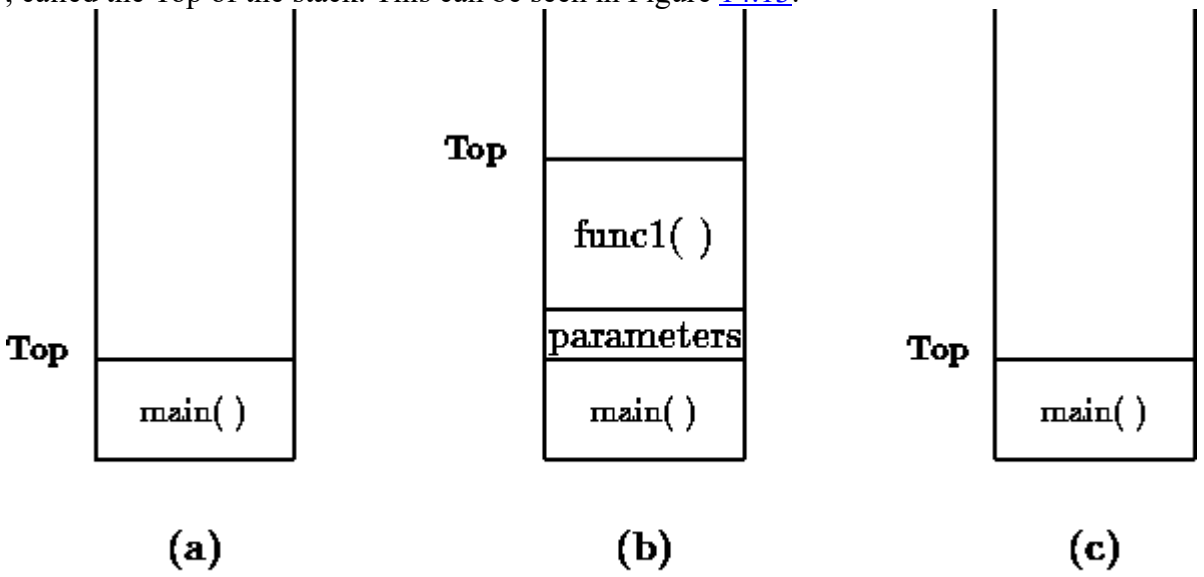


Figure 14.13: Organization of the Stack

When a program begins executing in the function main(), space is allocated on the stack for all variables declared within main(), as seen in Figure 14.13(a). If main() calls a function, func1(), additional storage is allocated for the variables in func1() at the top of the stack as shown in Figure 14.13(b). Notice that the parameters passed by main() to func1() are also stored on the stack. If func1() were to call any additional functions, storage would be allocated at the new Top of stack as seen in the figure. When func1() returns, storage for its local variables is deallocated, and the Top of the stack returns to to position shown in Figure 14.13(c). If main() were to call another function, storage would be allocated for that function at the Top shown in the figure. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program. Therefore, global variables (storage class external), and static variables are allocated on the heap. The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it. Thus, the heap area need not contain garbage.

Stack and heap differ in the following ways:

1. **Memory Allocation:** Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM .

2. **Variables: Variables allocated on the *stack* are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.** When a function or a method calls another function which in turns calls another function etc., the execution of all those functions remains suspended until the very last function returns its value. **Variables allocated on the *heap* have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory.**

3. **Access:** The stack is always reserved in a *LIFO order*, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer. Element of the heap have no dependencies with each other and can always be *accessed randomly* at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.

When to use stack and when to use heap?
You can use the stack if you know exactly how much data you need to allocate before compile time and it is not too big. You can use heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

In a multi-threaded situation each thread will have its own completely independent stack but they will share the heap. Stack is thread specific and Heap is application specific. The stack is important to consider in exception handling and thread executions.