Understanding Just-In-Time Compilation and Optimization

Write once, run anywhere, optimize just in time

Java source code is turned to bytecode by the javac compiler, and then run by the JVM which then compiles it to Assembly and feeds it to the CPU.

The generated bytecode is an accurate representation of the original Java source code, without any optimizations.

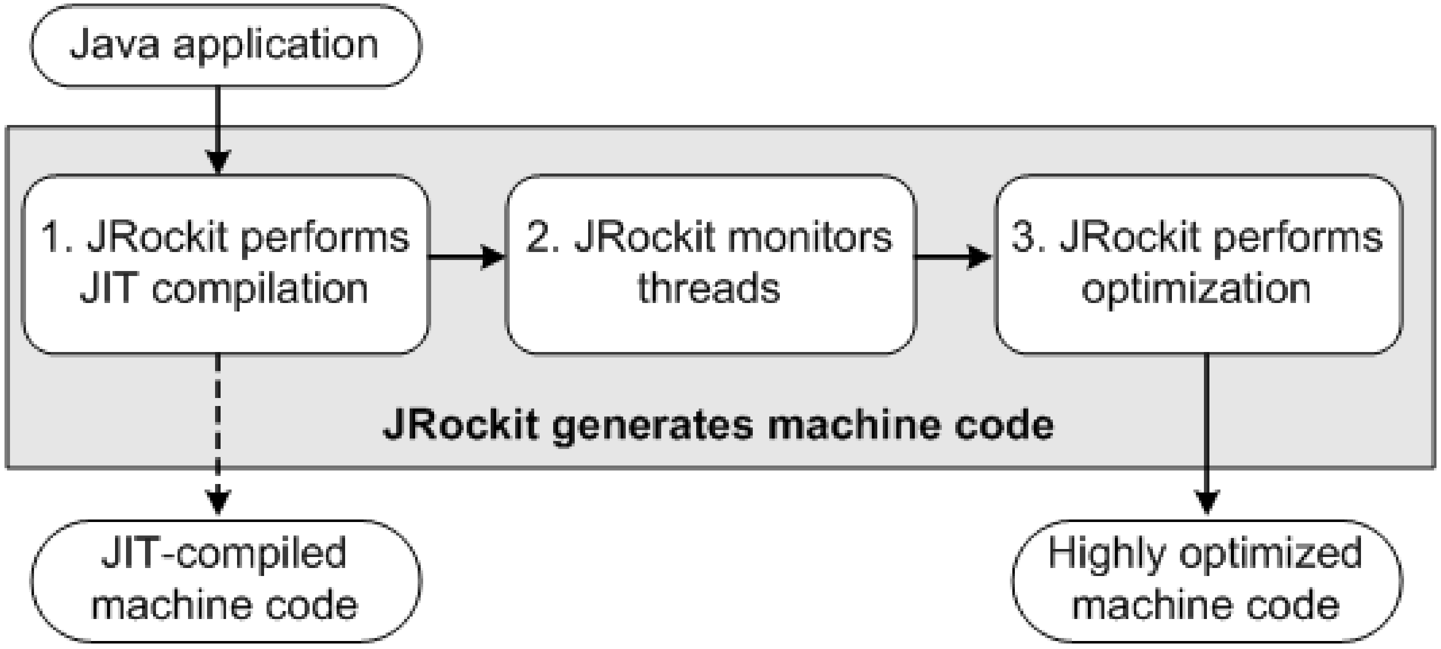
JVM translates it to Assembly using the below modes:

Interpreted mode – Where the JVM reads and runs the bytecode itself, as is.

Compiled mode (bytecode to assembly) – Where the JVM loosens its grip and there’s no bytecode involved.

How the JRockit JVM Generates Machine Code for Java Applications

The code generator in the JRockit JVM runs in the background during the entire run time of your Java application, automatically adapting the code to run optimally. The code generator works in three steps


Java application
1. JRockit performs JIT compilation
2. JRockit monitors threads
3. JRockit performs optimization
JRockit generates machine code
JIT-compiled machine code
Highly optimized machine code

Compilation Modes

Inside Java HotSpot VM, there are actually two separate JIT compiler modes,which are known as C1 and C2.

C1 is used for applications where quick startup and rock-solid optimization are required;
Ex : GUI applications

C2, on the other hand, was originally intended for long-running, predominantly server-side applications.
EX : Java Server's

Prior to some of the later Java SE 7 releases, these two modes were available using the -client and -server switches, respectively.

Later Java SE 7 releases, a new feature **called *tiered compilation*** became available.

This feature uses the C1 compiler mode at the start to provide better startup performance.

Once the application is properly warmed up, the C2 compiler mode takes over to provide more-aggressive optimizations and, usually, better performance.

With the arrival of Java SE 8, tiered compilation is now the default behavior.

The two compilers

The JVM that ships with OpenJDK contains two compiler back-ends:
C1, also known as 'client'
C2, also known as 'server�much'

The C1 compiler has a number of different modes, and will alter its response to a compilation request given a number of system factors, including, but not limited to, the current workload of the C1 & C2 compiler thread pool.

Given these different modes, the JDK refers to different *tiers*, which can be broken down as follows:

Tier 0- interpreter
Tier1 - client compiler with no profiling information
Tier2 - client compiler with basic counters
Tier3 - client compiler with profiling information
Tier4 - server compiler

jvm flags

To view all the available flags that can be passed to the jvm, run the following command:

```
java -XX:+PrintFlagsFinal
```

Full Logging of JIT Compilation

The switch for enabling full logging is
-XX:+LogCompilation

and it must be preceded by the option
-XX:+UnlockDiagnosticVMOptions.

Using the -XX:+LogCompilation switch produces a separate log file, hotspot_pid<PID>.log, in the startup directory.

To change the location of the file :

use -XX:LogFile=<path to file>.

Some JIT Compilation Techniques

One of the most common JIT compilation techniques used by Java HotSpot VM is inlining, which is the practice of substituting the body of a method into the places where that method is called. Inlining saves the cost of calling the method; no new stack frames need to be created. By default, Java HotSpot VM will try to inline methods that contain less than 35 bytes of JVM bytecode.

Another common optimization that Java HotSpot VM makes is monomorphic dispatch, which relies on the observed fact that, usually, there aren’t paths through a method that cause an object reference to be of one type most of the time but of another type at other times.

-XX:+PrintCompilation flag

The first step to understanding how JIT compilation in Java HotSpot VM is affecting the code is to see which of the methods are getting compiled.

add the -XX:+PrintCompilation flag to the script used to sart the Java processes.

Note: The resulting log of compilation events will end up in the standard log (that is, the standard output), and there is currently no way to redirect the entries to another file.

The -XX:+PrintCompilation flag output looks something like this:

```
1  sb   java.lang.ClassLoader::loadClassInternal  (6 bytes)
2  b    java.lang.String::lastIndexOf  (12 bytes)
3  s!b  java.lang.ClassLoader::loadClass  (58 bytes)
```

Flags correspond to:

b	Blocking compiler (always set for client)
*	Generating a native wrapper
%	On stack replacement
!	Method has exception handlers
s	Synchronized method

.hotspot_compiler file in the current working directory is used to customized the compiler

.hotspot_compiler

```
exclude Main main
dontinline Main doTest
compileonly Main doTest
```

Examples of Code Optimization

The following code examples show how the Hotspot JVM optimizes Java code.

Code Before Optimization:

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + y;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

Code After Optimization

```
class A {
B b;
public void newMethod() {
   y = b.value;
   ...do stuff...
   sum = y + y;
}
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

Inspecting Compilation with jstat

Seeing the output of print compilation requires that the program be started with the -XX:+PrintCompilation flag. If the program was started without that flag, you can get some limited visibility into the working of the compiler by using jstat.

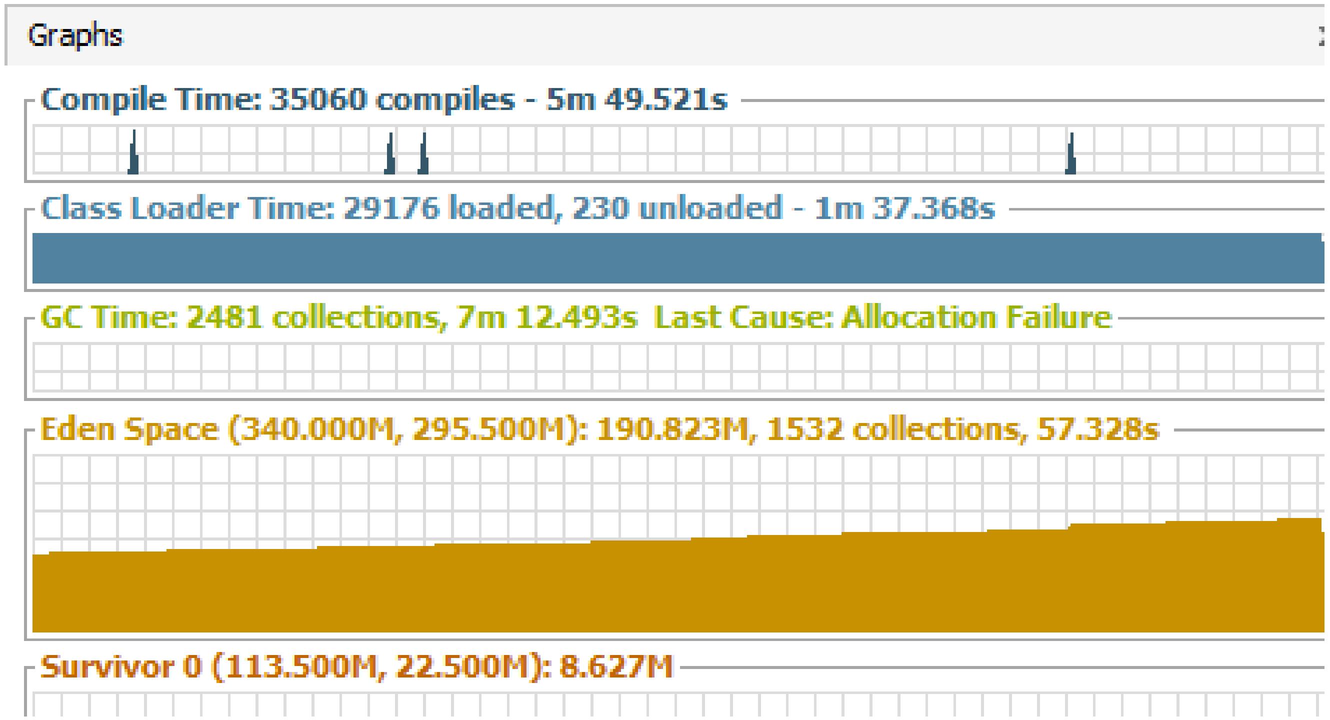
jstat has two options to provide information about the compiler. The -compiler option supplies summary information about how many methods have been compiled (here 5003 is the process id of the program to be inspected):

```
% jstat -compiler 5003
Compiled Failed Invalid   Time   FailedType FailedMethod
     206      0       0   1.97          0
```

Alternately, we can use the -printcompilation option to get information about the last method that is compiled. Because jstat takes an optional argument to repeat its operation, you can see over time which methods are being compiled. In this example, jstat repeats the information for process id 5003 every second (1000 milliseconds):

```
% jstat -printcompilation 5003 1000
Compiled  Size  Type Method
     207     64    1 java/lang/CharacterDataLatin1 toUpperCase
     208      5    1 java/math/BigDecimal$StringBuilderHelper getCharArray
```

VisualGC -> JIT Compiled classes details



Compile Time: 35060 compiles - 5m 49.521s

Class Loader Time: 29176 loaded, 230 unloaded - 1m 37.368s

GC Time: 2481 collections, 7m 12.493s Last Cause: Allocation Failure

Eden Space (340.000M, 295.500M): 190.823M, 1532 collections, 57.328s

Survivor 0 (113.500M, 22.500M): 8.627M

Compilation Thresholds

If the method we are interested in hasn’t been compiled yet, JITWatch will notify the same.

We can generate some more load for our application or lower the thresholds for JIT so it will treat the method as hot enough to compile.

-XX:+CompileThreshold=*N* flag.

The default value of N for the client compiler is 1,500; for the server compiler it is 10,000.

Reading the compiler's mind

The -XX:+LogCompilation flag produces a low-level XML file about compiler and runtime decisions

```
-XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation -
XX:+PrintInlining -XX:+PrintCompilation
```

Print Assembly:

```
-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly
```

Exception:

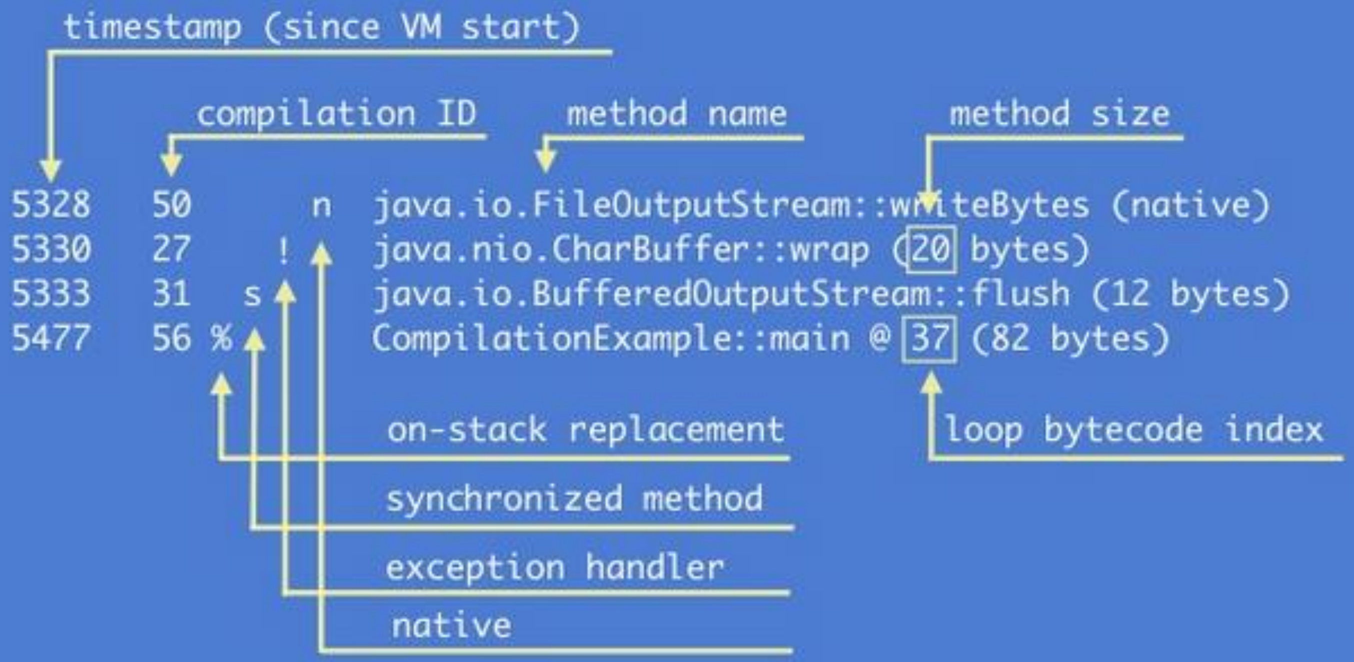
“Could not load hsdis-amd64.dll; library not loadable; PrintAssembly is disabled“.

Solution:

Download hsdis-1.1.1-win32-amd64.zip file from http://fcml-lib.com/download.html
and copy hsdis-amd64.dll file in jre/jdk bin folder.

Ref : x86_64 Assembly to understand generate code

Compilation Log



```
5328   50      n  java.io.FileOutputStream::writeBytes (native)
5330   27     !   java.nio.CharBuffer::wrap (20 bytes)
5333   31   s     java.io.BufferedOutputStream::flush (12 bytes)
5477   56 %       CompilationExample::main @ 37 (82 bytes)
```

Using this reference, we can break down the information in the log output from our test program:

```
    137   48 %     3       
demo.C1LoopBackedgeThresholdMain::exerciseTier3LoopBackedgeThreshold @ 5 (25 bytes)
```

This compile happened 137 milliseconds after JVM startup
Compilation ID was 48
This was an on-stack replacement
This compilation happened at Tier3
The OSR loop bytecode index is 5
The compiled method was 25 bytecodes

On-Stack Replacement

On-Stack replacement is a mechanism that allows the interpreter to take advantage of compiled code, even when it is still executing a loop for that method in interpreted mode. The below flow explains :

- ✓ Start executing a method loopyMethod in the interpreter
- ✓ Within loopyMethod, we execute an expensive loop body 1,000,000 times
- ✓ The interpreter will see that the loop count has exceeded the
- ✓ Tier3BackedgeThreshold setting
- ✓ The interpreter will request compilation of loopyMethod
- ✓ The method body is expensive and slow, and we want to start using the compiled version immediately. Without OSR, the interpreter would have to complete the 1,000,000 iterations of slow interpreted code, dispatching to the complied method on the next call to loopyMethod()
- ✓ With OSR, the interpreter can dispatch to the compiled frame at the start of the next loop iteration
- ✓ Execution will now continue in the compiled method body

Not Entrant Code

If a bunch of calls are made to http://localhost:8080/StockServlet (that is, without the log parameter), the compiler will see that the actual type of the sph object is StockPriceHistoryImpl. It will then inline code and perform other optimizations based on that knowledge.

Later, say a call is made to http://localhost:8080/StockServlet?log=true. Now the assumption the compiler made regarding the type of the sph object is false; the previous optimizations are no longer valid. This generates a deoptimization trap, and the previous optimizations are discarded.

```
if (log != null && log.equals("true")) {
   sph = new StockPriceHistoryLogger(...);
}
else {
   sph = new StockPriceHistoryImpl(...);
}
```

-XX:-TieredCompilation disables intermediate compilation tiers (1, 2, 3), so that a method is either interpreted or compiled at the maximum optimization level (C2).

As a side effect TieredCompilation flag also changes the number of compiler threads, the compilation policy and the default code cache size. Note that with TieredCompilation disabled

there will be less compiler threads;

simple compilation policy (based on method invocation and backedge counters) will be chosen instead of advanced compilation policy;

default reserved code cache size will be 5 times smaller.

To disable C2 compiler and to leave only C1 with no extra overhead, set -XX:TieredStopAtLevel=1.

To disable all JIT compilers and to run everything in interpreter, use -Xint.

Null pointer Exception

Explicit null checks can be eliminated from optimized Assembly code in most cases, and it holds true for any kind of condition that happens rarely.
Because they are likely to introduce jumps into the assembly code, slowing down its execution.

```
private static void runSomeAlgorithm(Graph graph) {

        if (graph == null) {
                return;
        }
        // do something with graph
}
```

JIT sees that graph is never called with null, it optimize the code as below:

```
private static void runSomeAlgorithm(Graph graph) {
        // do something with graph
}
```

Note: when a rare condition does kick in, the JVM would know to “**deoptimize**” and get the desired result

Branch Prediction

Jit decide whether certain lines of code are “hotter” than others and happen more often.

In Null Pointer exception, we discussed if a certain condition rarely or never holds, it’s likely that the “uncommon trap” mechanism will kick in and eliminate it from the compiled Assembly.

Jit checkes branches of an IF condition, if one happens more than the other, the JIT compiler can reorder them according to the most common one

```
if (x >= y) {
                    veryHardCalculation = x * 1000 + y;
          } else {
                    veryHardCalculation = y * 1000 + x;       }
```

<u>optimized to</u>

```
if (x < y) {
                    // this would not require a jump
                    veryHardCalculation = y * 1000 + x;
                    return veryHardCalculation;
          } else {
                    veryHardCalculation = x * 1000 + y;
                    return veryHardCalculation;   }
```

Note : assuming that most of the time x < y, the condition will be flipped to statistically reduce the number of Assembly jumps

Loop Unrolling

Each iteration is actually an Assembly jump back to the beginning of the instruction set. With loop unrolling, the JIT compiler opens up the loop and just repeats the corresponding Assembly instructions one after another.

For example, let's take a look at a method that multiplies a matrix by a vector:

```
private static double[] loopUnrolling(double[][] matrix1, double[] vector1) {

        double[] result = new double[vector1.length];

        for (int i = 0; i < matrix1.length; i++) {
        for (int j = 0; j < vector1.length; j++) {
        result[i] += matrix1[i][j] * vector1[j];
                }       }

        return result; }
```

The unrolled version would look like this:

```
for (int i = 0; i < matrix1.length; i++) {

                    result[i] += matrix1[i][0] * vector1[0];

                    result[i] += matrix1[i][1] * vector1[1];

                    result[i] += matrix1[i][2] * vector1[2];

                    // and maybe it will expand even further - e.g. 4 iterations, thus

                    // adding code to fix the indexing

                    // which we would waste more time doing correctly and efficiently

          }
```

Inlining Methods

Another jump killer optimization. A huge source for Assembly jumps are in fact method calls. JIT compiler inlines methods with 2 JVM arguments:

-XX:MaxInlineSize – The maximum size of the bytecode of a method that can be inlined (done for any method, even if not being executed frequently). Default is about ~35 bytes.

-XX:FreqInlineSize – The maximum size of the bytecode of a method that is considered a hot spot (executed frequently), that should be inlined. Default varies between platforms.

Thread fields and Thread Local Storage (TLS)

Thread fields are much faster than regular variables. The thread object is stored on the actual CPU register, making its fields a very efficient storage space.

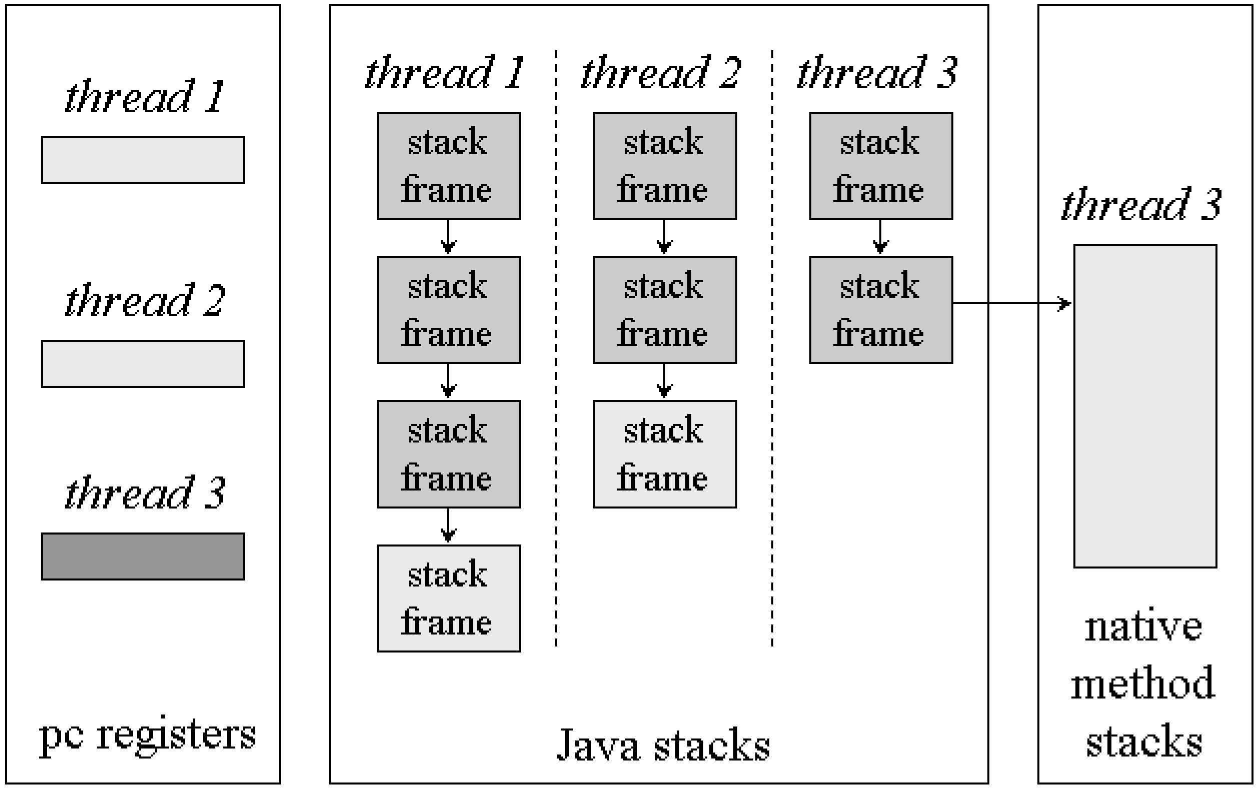
With thread local storage, we are able to create variables stored on the Thread object.

Ex:

```
private static final ThreadLocal<Integer> counter = new
ThreadLocal<Integer>();

private static void handleRequest() {
        if (counter.get() == null) {
                counter.set(0);
        }

        counter.set(counter.get() + 1);
```


thread 1
thread 2
thread 3
pc registers
thread 1
thread 2
thread 3
stack frame
Java stacks
thread 3
native method stacks

When to use ThreadLocal?

When you need a variable's value to depend on the current thread and it isn't convenient to attach the value to the thread in other ways, you can consider using ThreadLocal.

1. If the variable is not thread safe, it can become thread safe by saving it into ThreadLocal.

For example: Hibernate Session is not thread safe, and can be save into the ThreadLocal. In this case, only the thread owning the Session can access the session object saved in ThreadLocal.

2, Implement per thread context information like transaction ID

How to use ThreadLocal?

1. Create a ThreadLocal variable

```
private static final ThreadLocal<Session> threadLocal =
    new ThreadLocal<Session>();
```

2. To put Object into ThreadLocal

```
threadLocal.set(session);
```

3. Get Object form ThreadLocal

```
Session session = (Session) threadLocal.get();
```

4. Remove object from ThreadLocal

```
threadLocal.set(null);    (or)  threadLocal.remove();
```