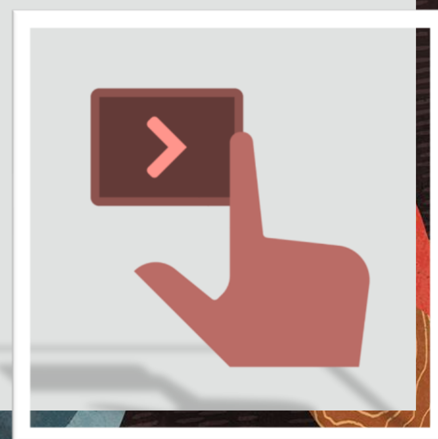
ORACLE
Academy

Java Programming

8-1
JDK Tools



ORACLE
Academy

Objectives

- This lesson covers the following topics:
  - Introduce the javac command
  - Introduce the java command
  - How to use jps command
  - How to use jstat command
  - Introduce and use the javap command
  - How to use the jdb command
  - Introduce the jvisualvm tool
  - Introduce the hsdis plugin

ORACLE Academy

javac Command

- The javac command reads Java class or interface definition files, compiles these into bytecode and generates class files
- The command to run javac is as follows:
  - javac [options] [source files] {@argfiles}

ORACLE Academy

javac Command

- Partial list of options:
  - -d directory: sets the destination directory for the class files. If the –d option is not specified, then javac puts each class file in the same directory as the source file
  - -g: generate all debugging information
  - -help: display information about compiler options
  - -classpath path: specify the location of the class files, override the system CLASSPATH variable
  - -sourcepath path: specify the location that the compiler will use to find the source code files

ORACLE Academy

javac Command

- For modern Java projects, javac is not often used directly, as it is rather low-level and not easy to use, especially for larger codebases
- Instead, modern Integrated Development Environments (IDEs) automatically use javac for the developer
- For deployment, most projects will make use of separate build tools, such as Maven, Ant, or Gradle
- Nevertheless, it is useful for developers to understand how to use javac

Have a look online to find out more about Maven, Ant and Gradle.

javac Command

- Example SourceCode:

```
package com.app;

public class Hello {
   public void sayHello() {
      System.out.println("Hello Java");
   }//end method sayHello
}//end class Hello
```

```
package com.app;

public class TestHello {

   public static void main(String[] args) {
      Hello hello = new Hello();
      hello.sayHello();
   }//end method main
}//end class TestHello
```

There are two Java source files **Hello.java** and **TestHello.java**

For each class definition in the Java source file, it is declared in the **com.app** package

Source code file names must have **.java** suffix, for example **Hello.java**

ORACLE
Academy

javac Command

- Example Folder structure for the source code.

```
 Directory of C:\src\com\app

08/06/2019  18:36    <DIR>          .
08/06/2019  18:36    <DIR>          ..
08/06/2019  18:35               165 Hello.java
08/06/2019  18:35               201 TestHello.java
               2 File(s)            366 bytes
               2 Dir(s)  129,107,288,064 bytes free
```

javac -sourcepath src -d classes src\com\app\TestHello.java

```
src
|       com
|       |       app
|       |       |
|       |       |       TestHello.java
|       |       |       Hello.java
```

ORACLE
Academy

javac Command

- Arranges source files in a directory tree that reflects the package tree name
- For example, all source code files are put in the src\com\app folder
- The Java source code files will be located under the src folder
- javac command uses the –sourcepath option to specify the location of the Java source files

ORACLE
Academy

javac Command

- The Java class files are located under the classes folder
- The javac command uses the –d option to specify the location of the Java class files

ORACLE
Academy

java Command

- The java command launches a Java application which contains the main method
- The command syntax to run the java utility is as follows:
  - java [options] classname [args]

java Command

- Partial list of options:
  - -jar filename: Executes a program in a JAR file
  - -client: choose the Java HotSpot Client VM
  - -server: choose the Java HotSpot Server VM
  - -X: Display information about nonstandard options

java Command

- The java command starts the Java Runtime environment (JRE), which will load the followed class and will try to call the class main() method
- The args argument will be passed to the main() methods String[] parameter
- The JVM searches for the class from the bootstrap class path, the extension path and the application class path

java Command

- Example

```
C:\test>java -classpath classes com.app.TestHello
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
Hello Java
```

  - classes
  - | com
  - | | app
  - | | |
  - | | | TestHello.class
  - | | | Hello.class
- In this example, the java command launches the JVM which searches for the TestHello class from the classes classpath.

ORACLE Academy

jps Tool

- jps is a process status tool that is most commonly used to determine the Process ID (PID) of the Java process
- It works in an OS-independent way and is a very convenient tool
- With no options, jps will list each Java application's lvmid (the operating system's process identifier for the JVM process) followed by the short form of the application's class name or jar file name
- The short form of the class name or JAR file name omits the class's package information or the JAR files path information

jps Tool

- From the following output of jps, there are 4 JVMs
- Note that the lvmid is listed along with a short name for the main method that started the JVM

```
D:\JP_NEW\Section 1\Lesson 2>jps
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
15184 SizeTest
15968 jar
2568 Jstat
16172 Jps

D:\JP_NEW\Section 1\Lesson 2>
```

ORACLE Academy

jps Tool

- The jps command supports a number of options that modify the output of the command
  - -q : Suppresses the output of the class name, JAR file name, and arguments passed to the main method, producing only a list of local JVM identifiers
  - -m : Displays the arguments passed to the main method. The output may be null for embedded JVMs
  - -l : Displays the full package name for the application's main class or the full path name to the application's JAR file

jps Tool

- jps command options (cont)
  - -v : Displays the arguments passed to the JVM
  - -V : Suppresses the output of the class name, JAR file name, and arguments passed to the main method producing only a list of local JVM identifiers

jstat Tool

- The jstat tool displays performance statistics for an instrumented HotSpot JVM
- You can enable command line options to select the specific statistic you want from a running JVM
- Before we can use jstat, we need to get the list of JVM pids on the system, for example using jps

jstat Tool

- With a pid, the jstat command with the -gcutil option can be used to monitor the garbage collector
- For example, to display gc information every five seconds we would use the following command :
  - jstat -gcutil 7140 5000
- The jstat command-line tool displays detailed performance statistics for a local or remote HotSpot VM
- The –gctuil option is used most frequently to display garbage collection information

jstat Tool

- jstat command –gc option
  - S0C Current survivor space 0 capacity (KB)
  - S1C Current survivor space 1 capacity (KB)
  - S0U Survivor space 0 utilization (KB)
  - S1U Survivor space 1 utilization (KB)
  - EC Current eden space capacity (KB)
  - EU Eden space utilization (KB)
  - OC Current old space capacity (KB)
  - OU Old space utilization (KB)
  - PC Current permanent space capacity (KB)
  - PU Permanent space utilization (KB)
  - YGC Number of young generation GC Events
  - YGCT Young generation garbage collection time
  - FGC Number of full GC events
  - FGCT Full garbage collection time
  - GCT Total garbage collection time

ORACLE
Academy

jstat tool

- jstat –gcutil tool output sample

命令提示符 - jstat -gcutil 15968 10000

```
D:\JP_NEW\Section 1\Lesson 2>jps
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
15184 SizeTest
15968 jar
17708 Jps

D:\JP_NEW\Section 1\Lesson 2>jstat -gcutil 15968 10000
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
  0.00   0.00  74.55  67.38  94.68  90.74      4    0.036     5    0.151    0.187
```

jstat Tool

- From the output on the previous slide, you can see 4 minor garbage collections have taken place
- Here is an explanation of columns shown in the output:
  - S0: Survivor space 0 utilization as a percentage of the space's current capacity
    S1: Survivor space 1 utilization as a percentage of the space's current capacity
    E: Eden space utilization as a percentage of the space's current capacity
    O: Old space utilization as a percentage of the space's current capacity
    M: Metaspace utilization as a percentage of the space's current capacity
    YGC: Number of young generation GC events
  - CCS: Compressed class space utilization as a percentage
  - YGC: Number of young generation GC events
  - YGCT: Young generation garbage collection time
    FGC: Number of full GC events
    FGCT: Full garbage collection time
    GCT: Total garbage collection time

jstat Tool

- jstat Tool Options
- jstat [generalOption | outputOptions vmid [interval[s|ms] [count]]

  - -h n: Displays a column header every n samples (output rows).
  - -t: Displays a timestamp column as the first column of output
  - -statOption : Determines the statistics information the jstat command displays

ORACLE Academy

jstat Tool

- jstat Tool Options (cont)
  - -class : Class loader statistics
  - -compiler : Java HotSpot VM Just-in-Time compiler statistics
  - -gc : Garbage-collected heap statistics
  - -gccapacity : Memory pool generation and space capacities

javap Command

- The javap command disassembles one or more Java class files
- The javap utility can help you to understand how Java source code is mapped to bytecode
- The command syntax to run the java utility is as follows:
  - javap [options] classfiles
- Partial list of options:
  - -help : Prints a help message
  - -verbose : prints details information for stack size, number of locals and arguments for methods

javap Command

- The benefits of using the javap tool is that it enables us to understand how the compiler deals with our code in case we have doubts about some of the Java programming techniques
- Example:
  - Let's take a look at the simple Java class example to understand the tool output to use it in our own investigations and investigate how a Java compiler deals with and optimizes the Java code

ORACLE Academy

javap Command

- The following code prints out a simple, “Hello Java” message:

```
package com.app;

public class Hello {
   public void sayHello() {
      System.out.println("Hello Java");
   }//end method sayHello
}//end class Hello
```

ORACLE
Academy

javap Command

- Taking the code from the previous slide and running the command :
  - javap –classpath classes –verbose com.app.Hello
- Will provide the following output :

```
C:\test>javap -classpath classes -verbose com.app.Hello
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
Classfile /C:/test/classes/com/app/Hello.class
  Last modified Jun 14, 2017; size 401 bytes
  MD5 checksum dee51083f45e08752bdfc97c60200a43
  Compiled from "Hello.java"
public class com.app.Hello
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/Print
Stream;
   #3 = String             #17            // Hello Java
   #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/
lang/String;)V
   #5 = Class              #20            // com/app/Hello
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               sayHello
  #12 = Utf8               SourceFile
  #13 = Utf8               Hello.java
  #14 = NameAndType        #7:#8          // "<init>":()V
  #15 = Class              #22            // java/lang/System
  #16 = NameAndType        #23:#24        // out:Ljava/io/PrintStream;
  #17 = Utf8               Hello Java
  #18 = Class              #25            // java/io/PrintStream
  #19 = NameAndType        #26:#27        // println:(Ljava/lang/String;)V
```

ORACLE
Academy

javap Command

- javap Command Example - Continued

```
   #20 = Utf8               com/app/Hello
   #21 = Utf8               java/lang/Object
   #22 = Utf8               java/lang/System
   #23 = Utf8               out
   #24 = Utf8               Ljava/io/PrintStream;
   #25 = Utf8               java/io/PrintStream
   #26 = Utf8               println
   #27 = Utf8               (Ljava/lang/String;)V
{
  public com.app.Hello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>
":()V
         4: return
      LineNumberTable:
        line 2: 0

  public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljav
a/io/PrintStream;
         3: ldc           #3                  // String Hello Java
         5: invokevirtual #4                  // Method java/io/PrintStream.prin
tln:(Ljava/lang/String;)V
         8: return
```

Academy

Java Debugger jdb

- The J2SE JDK includes a Java debugger (jdb)
- The jdb program is a useful tool for programmers trying to understand application errors

ORACLE
Academy

Java Debugger jdb

- Before running the debugger, it is important to compile the target classes with the –g option
- This option results in the inclusion of symbol information in the class file
- Symbol information assists the debugger to associate the source code with the byte codes
- To effectively use the debugger, you must avail it the source code and the specially compiled class files

ORACLE
Academy

Java Debugger jdb

- You use the debugger to analyze the internal state of a running program
- It does this by controlling the JVM in several ways :
  - The debugger decides when the VM will execute code and when it will wait for instructions
  - The debugger can list source code and the call stack to indicate what source line the program is currently running in each thread
  - The debugger can also inspect and modify data internal to the program

ORACLE
Academy

Java Debugger jdb

- We use jdb instead of java to run the Java Application.
- The following is an example:
  - jdb Example arg1 arg2
- This command will :
  - Start and initialize the JVM
  - Load the followed class
  - Execute static blocks (static initializers) including initializing static variables
  - Execute the main method

Java Debugger jdb

- Basic jdb commands :
  - Help or ? : Provides help on the command
  - Dump : The dump command prints the current value of each field defined in the object,including static and static fields
  - Stop : sets a breakpoint
  - Print: displays Java object and primitive values
  - Cont: continues execution of the debugged application after a breakpoint

Java Debugger jdb

- When using jdb to step through Java code or set breakpoints using the debugger, you will reference lines in the source code
- To best understand how to use the debugger, you must understand some details about the Java programming language and bytecode generation

Java Debugger jdb

- Bytecode instructions can be showed using the following command :
  - javap –c ClassName
- This command displays bytecode associated with an index or address to each instruction
- The index is bytecode index (bci) in the jdb tool

Java Debugger jdb

- Java Source and Bytecode:
  – When you trace through the flow of a program with the step or next command in the jdb, the debugger runs each of the bytecode instructions that corresponds to the statements on a single line of the source code
  – When using step you can step through only a single bytecode instruction

Java Debugger jdb

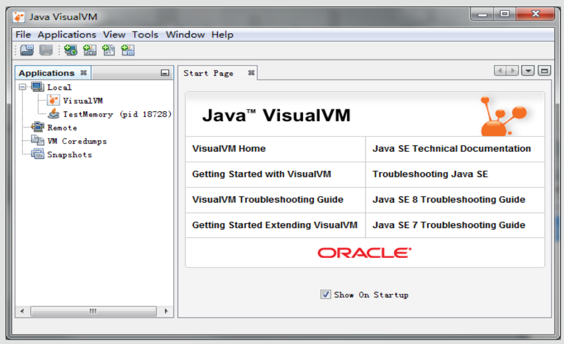
- For example, you might see a line in the jdb
  - Step completed: "thread=main", StackExample.main(), line=9 bci=16
- You can find the corresponding source code at line 9 by using the list command
- Use the javap tool with –c option to obtain a bytecode listing

Java Debugger jdb

- A knowledge of the source code and bytecode relationship details can assist you better understand the next, step, and stepi instructions, and correctly track through the flow of an application

Java VisualVM

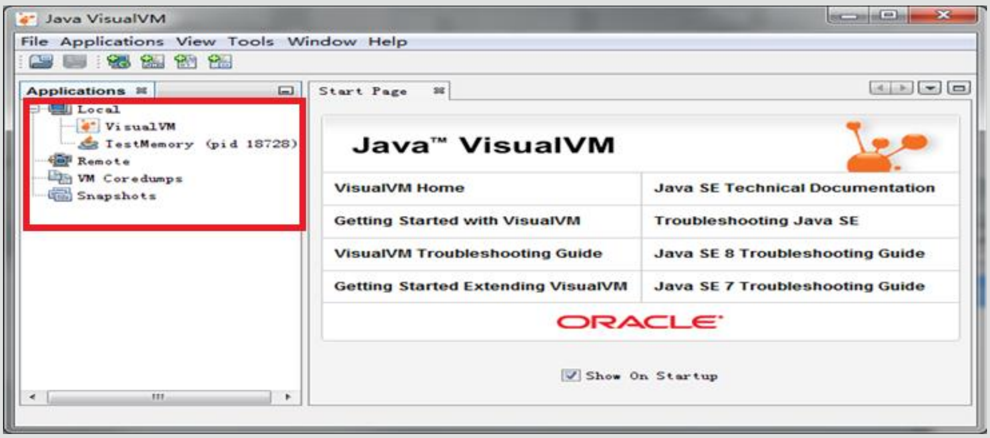
- The JVM provides tools to yield metrics linked to performance and, in particular, the CPU and memory
- JVisualVM is such a tool packaged with the JDK (not the JRE)
- Once launched, you will see the welcome screen of jvisualvm:



ORACLE Academy

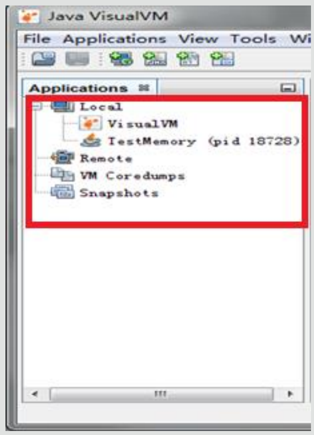
Java VisualVM

- To start using jvisualvm, you will need to select a JVM
- This is done through the tree on the left-hand side of the screen :



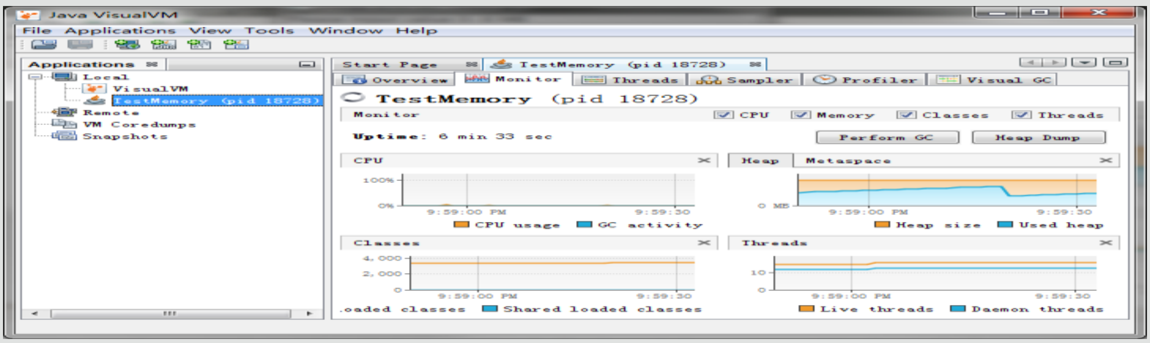
Java VisualVM

- The two options are Local and Remote
- In this case, we'll run the server on our local machine, so it is automatically detected by jvisualvm



Java VisualVM

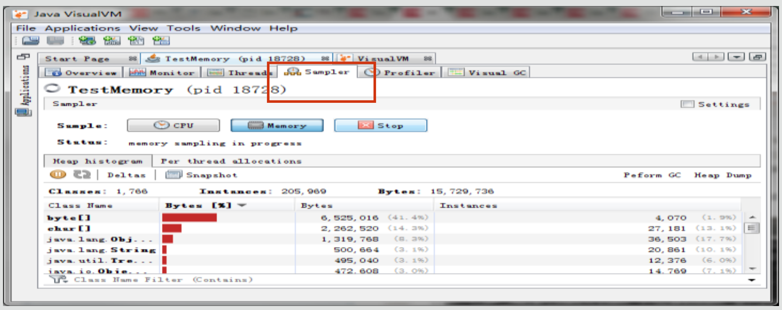
- There are two processes:
  - VisualVM: This is a Java process and detects itself
  - TestMemory: Once you have identified your process in the list, you need to double-click on it and you will get the following screen showing high-level information about the process:



ORACLE
Academy

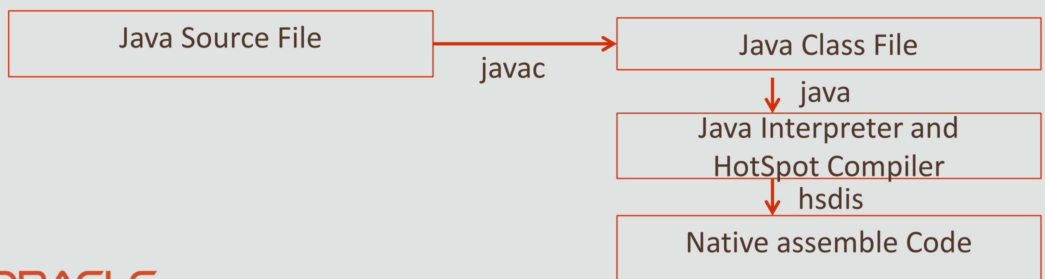
Java VisualVM

- Sampler:
  - This tab is useful and allows you to capture what the application is doing in terms of CPU and memory
  - Here is the memory view that you will get once some samples have been captured:



hsdis: JIT Hotspot Disassemble

- As discussed previously, we use the javac utility to compile java source code, the java utility to launch the JVM, jdb to debug and jstat to collect the runtime data
- Now we will introduce hsdis to display the JVM Client or Server Compiler



ORACLE
Academy

hsdis: JIT Hotspot Disassemble

- hsdis Example

```
public class JitTest{

        int i=1;
        static int j=2;
        public int add(int k){
                return i+j+k;
        }

        public static void main(String[] args){
                JitTest jt=new JitTest();
                jt.add(3);
        }
}
```

ORACLE
Academy

hsdis Command Line

- java -XX:+UnlockDiagnosticVMOptions
  - -XX:+PrintAssembly -Xcomp
  - -XX:CompileCommand=dontinline,*JitTest.add
  - -XX:CompileCommand=compileonly,*JitTest.add JitTest
  - -Xcomp: Forces compilation of methods on first invocation
  - -XX:+PrintAssembly: Enables printing of assembly code for bytecoded and native methods

hsdis Command Line

- XX:CompileCommand=command,method[,option]:
- Specifies a command to perform on a method
  - dontinline: Prevent inlining of the specified method
  - compileonly: Exclude all methods from compilation except for the specified method
- -XX:+UnlockDiagnosticVMOptionsUnlocks the options intended for diagnosing the JVM

hsdis Output

Argument passed to add method stored in r8 register

First argument passed in

```
[Disassembling for mach='i386:x86-64']
[Entry Point]
[Constants]
  # {method} {0x000000001b9002a8} 'add' '(I)I' in 'JitTest'
  # this:     rdx:rdx   = 'JitTest'
  # parm0:    r8        = int
  #           [sp+0x40]  (sp of caller)
  0x000000000289a660: mov    0x8(%rdx),%r10d
  0x000000000289a664: shl    $0x3,%r10
  0x000000000289a668: cmp    %rax,%r10
  0x000000000289a66b: jne    0x00000000027d5f60  ;   {runtime_call}
  0x000000000289a671: data16 data16 nopw 0x0(%rax,%rax,1)
  0x000000000289a67c: data16 data16 xchg %ax,%ax
[Verified Entry Point]
  0x000000000289a680: mov    %eax,-0x6000(%rsp)
  0x000000000289a687: push   %rbp
  0x000000000289a688: sub    $0x30,%rsp
  0x000000000289a68c: movabs $0x1b900558,%rax    ;   {metadata(method data for {method
} {0x000000001b9002a8} 'add' '(I)I' in 'JitTest')}
  0x000000000289a696: mov    0xdc(%rax),%esi
  0x000000000289a69c: add    $0x8,%esi
  0x000000000289a69f: mov    %esi,0xdc(%rax)
  0x000000000289a6a5: movabs $0x1b9002a0,%rax    ;   {metadata({method} {0x000000001b9
002a8} 'add' '(I)I' in 'JitTest')}
  0x000000000289a6af: and    $0x0,%esi
  0x000000000289a6b2: cmp    $0x0,%esi
  0x000000000289a6b5: je     0x000000000289a6dc  ;*aload_0
                                                 ; - JitTest::add@0 (line 6)
```

ORACLE
Academy

hsdis Output (Cont)

```
  0x00000000027fa77b: mov    0xc(%rdx),%eax      ;*getfield i
                                                 ; - JitTest::add@1 (line 6)

  0x00000000027fa77e: movabs $0x76c460670,%rsi  ;   {oop(a 'java/lang/Class' = 'JitTe
st')}
  0x00000000027fa788: mov    0x68(%rsi),%esi     ;*getstatic j
                                                 ; - JitTest::add@4 (line 6)

  0x00000000027fa78b: add    %esi,%eax
  0x00000000027fa78d: add    %r8d,%eax
  0x00000000027fa790: add    $0x30,%rsp
  0x00000000027fa794: pop    %rbp
  0x00000000027fa795: test   %eax,-0x23ba69b(%rip)        # 0x0000000000440100
                                                 ;   {poll_return}
  0x00000000027fa79b: retq
```

ORACLE
Academy

Java Runtime and Real Machine Runtime

- According to the specification that defines the Java virtual machine (usually called the VM Spec), the JVM is a stack based interpreted machine
- This means that rather than having registers (like a physical hardware CPU), it uses an execution stack of partial results, and performs calculations by operating on the top value (or values) of that stack

Java Runtime and Real Machine Runtime

- The following is Stack oriented Java Runtime.
  - aload_0
  - getfield #2 // Field i:I
  - getstatic #3 // Field j:I
  - iadd
  - iload_1
  - iadd
  - ireturn

hsdis Output Analysis

- Physical machine code:
  - %eax,-0x6000(%rsp)   check stack overflow
  - push   %rbp   save the stack frame pointer
  - sub $0x30,%rsp   allocate space for the current stack frame
  - mov   0xc(%rdx),%eax   ;*getfield I get the variable I and stored in the eax register
  - movabs $0x76c460670,%rsi

ORACLE
Academy

hsdis Output Analysis

- Physical machine code:

  - mov 0x68(%rsi),%esi ;*getstatic j get the variable J and store in the esi register
  - add %esi,%eax calculate i plus j store the result in the eax register
  - add %r8d,%eax calculate variable K(in r8d register) plus the value in eax register
  - add $0x30,%rsp release current stack frame space

hsdis Output Analysis

- Physical machine code:
  - pop %rbp resume the previous stack frame pointer
  - test %eax,-0x270ad5b(%rip) safepoint
  - Retq return

ORACLE
Academy

Summary

- In this lesson, you should have learned:
  - Introduce the javac command
  - Introduce the java command
  - How to use jps command
  - How to use jstat command
  - Introduce and use the javap command
  - How to use the jdb command
  - Introduce the jvisualvm tool
  - Introduce the hsdis plugin

ORACLE

Academy