

## Module 2 Demo 1: Using the prstat Utility

This demo presents a the use of the prstat utility to identify the LWPID that has the highest CPU usage.

### Required Software

- Solaris OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Demo objectives

- To identify the LWPID that that has the highest CPU usage
- Identify the Java technology thread that maps to LWPID

### Instructions

1. Open two (Solaris OS) terminal (or xterm) windows.
2. In one window, run the command:  

```
prstat -Lm
```

The prstat -Lm command will report micro-state information on a per LWP basis. You should note the following:

  - The PID and LWPID are reported in the right most column.
  - The LWPIDs are arranged in rows with the highest CPU utilizing LWPID at the top.
3. In the second window, run the Java2Demo and re-direct its output to a file:  

```
java -server -jar <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar > /tmp/demo1.txt
```
4. Observe the prstat -Lm window and note the PID and LWPID number of the busiest LWPID (top row of output).
5. Quit the prstat command.
6. Use the following format of the kill command to kill the busiest LWPID you identified in step 4.  

```
kill -3 <PID>
```

 where the PID is the PID you noted in step 4. For example: 

```
kill -3 957
```
7. Use the File menu to exit the Java2Demo application.
8. Open the file /tmp/demo1.txt to examine the Java thread dump.
9. Search the /tmp/demo1.txt for an nid value that matches the hex converted LWPID noted in step 4.  

For example, if the decimal value of the LWPID identified in step 4 is 10, then the corresponding hex value is 0xa. A search for the string 'nid=0xa' will find the Java thread mapping to LWPID 10.
10. If time permits repeat the exercise. If you are very quick, the busiest LWPID might correspond to the JVM's JIT compiler.

## Module 3 Demo 1: Examining the basics of how the garbage collector works

This demo uses VisualVM to examine object allocation and the basics of GC operation along with examining young generation and old generation Java heap spaces.

Slide id: 88

### Required Software

- Any OS
- JDK 1.5 or higher
- VisualVM

VisualVM is included in Java SE 6 update 7 (jdk 1.6.0\_u7). If you are running an earlier version of Java SE you can download VisualVM from: <http://visualvm.dev.java.net>.

- Java2Demo

Located in: `<JDK install dir>/demo/jfc/Java2D/Java2Demo.jar`

### Demo objective

- Examine the JVM's heap spaces and basics of how the JVM's garbage collector works with a particular focus on the young generation spaces (Eden and survivor spaces S1 and S2) and old (tenured) generation space.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 6 update 7 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: `java version "1.6.0_07"` or later you can bypass the following VisualVM installation step.

2. Download and install VisualVM from <http://visualvm.dev.java.net>.
3. Add the `bin` subdirectory of the VisualVM install directory to the OS PATH variable.

Install VisualGC plugin for VisualVM.

1. Launch VisualVM

If you are using Java SE 6 update 7 or later run the following command in a terminal window:

```
jvisualvm
```

If you had installed VisualVM in step 1, run the following command in a terminal window:

```
visualvm
```

2. Within Java VisualVM, download and install the VisualGC plug-in by selecting `Tools > Plugins` menu option.

On the Available Plug-ins tab, select the VisualGC Plug-in and press the "Install" button. Click through the installer wizard to install VisualGC plug-in.

Close the Plug-ins window.

## Instructions

1. Launch VisualVM, if you bypassed performing the preparation steps.

If you are using Java SE 6 update 7 or later run the following command in a command line window:

```
jvisualvm
```

If you had installed VisualVM in step 1, run the following command in a terminal window:

```
visualvm
```

2. In a command line window, run the Java2Demo. It is located in the <JDK install dir>/demo/jfc/Java2D/ directory.  

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:PermSize=5m -  
XX:MaxPermSize=5m -jar <JDK install  
dir>/demo/jfc/Java2D/Java2Demo.jar
```
3. Examine the VisualVM application GUI and notice the left panel automatically detects the Java2D demo application and displays its PID. Note this PID.
4. Refresh the VisualGC tab (the right panel of VisualVM GUI). Select the PID that matches the Java2D demo you noted in 5 from the drop down box labeled "Select Local VM".
5. Examine the VisualGC application now running in the right panel. In particular observe the Java heap spaces: Perm, Old, Eden, S0 and S1.
6. Click on the "Transforms" tab of Java2Demo and observe the Java heap spaces on the VisualGC GUI.

Notice how Eden fills quickly with new allocating objects. Eden is where new objects are allocated. The objects that survive an eden collection show up in S0 or S1. Each young generation (Eden) collection also includes a collection of the survivor space (S0 or S1) that contains objects. The survivors of the collection are moved to the other survivor space. Objects that survive up to 15 times get promoted to the Old generation space.

When you clicked on the "Transforms" tab, you probably observed Old generation increasing. This occurs since new objects to support the Transforms" tab have been allocated. They will need to stay around until the Java2D application leaves the Transform. You should also observe a nice saw tooth pattern on the Eden Space. You can also see objects being copied to and from S0 and S1.

## Module 3 Demo 2: Examining the Permanent Generation

This demo uses VisualVM to examine the Permanent Generation heap space, what it does and how it works.

Slide id: 91

### Required Software

- Any OS
- JDK 1.5 or higher
- VisualVM

VisualVM is included in Java SE 6 update 7 (jdk 1.6.0\_u7). If you are running an earlier version of Java SE you can download VisualVM from: <http://visualvm.dev.java.net>.

- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Examine the JVM's heap spaces and basics of how the JVM's garbage collector works with a particular focus on the permanent generation space.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 6 update 7 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version "1.6.0\_07" or later you can bypass the following VisualVM installation step.

2. Download and install VisualVM from <http://visualvm.dev.java.net>.
3. Add the bin subdirectory of the VisualVM install directory to the OS PATH variable.

Install VisualGC plugin for VisualVM.

- Launch VisualVM

If you are using Java SE 6 update 7 or later run the following command in a terminal window:

```
jvisualvm
```

If you had installed VisualVM in step 1, run the following command in a terminal window:

```
visualvm
```

- Within Java VisualVM, download and install the VisualGC plug-in by selecting Tools > Plugins menu option.

On the Available Plug-ins tab, select the VisualGC Plug-in and press the "Install" button. Click through the installer wizard to install VisualGC plug-in.

Close the Plug-ins window.

## Instructions

1. Launch VisualVM, if you bypassed performing the preparation steps.

If you are using Java SE 6 update 7 or later run the following command in a command line window:

```
jvisualvm
```

If you had installed VisualVM in step 1, run the following command in a terminal window:

```
visualvm
```

2. In a different command line window, run the Java2Demo. It is located in the <JDK install dir>/demo/jfc/Java2D/ directory.

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:PermSize=1m -XX:MaxPermSize=3m -jar  
<JDK install dir>/demo/jfc/Java2D/Java2Demo.jar
```

3. Examine the VisualVM application GUI and notice the left panel automatically detects the Java2D demo application and displays its PID. Note this PID.
4. Refresh the VisualGC tab (the right panel of VisualVM GUI). Select the PID that matches the Java2D demo you noted in step 4 from the drop down box labeled "Select Local VM".
5. Examine the VisualGC application now running in the right panel. In particular observe the Java heap spaces: Perm, Old, Eden, S0 and S1.
6. Java2Demo and slowly click on each of the tabs.

As you click on each tab, watch what happens to the "Perm" space in VisualGC. Do this for each of the tabs. Some of the tabs will result in Perm filling / raising more than others. This is expected.

### Explanation:

The clicking of each tab causes the Java2Demo classes required to meet the requested functionality to load. As classes load into Permanent Generation, you can notice its use, or occupancy grow. If Permanent Generation runs out of space, it will try to unload unused / unreferenced classes by invoking a FullGC. If that fails to create enough space, then the JVM will throw an OutOfMemoryError. This is probably the most common root cause of OutOfMemoryErrors in Java applications. It historically has been very common in Java EE applications.

## Module 3 Demo 3: Using -verbose:gc

This demo obtains and interprets GC output produced by the -verbose:gc option of the java command.

Slide id: 96

### Required Software

- Any OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Use the -verbose:gc option of the java command to obtain GC output.
- Interpret the GC output.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the Java2D demo directory:  

```
cd <JDK install dir>/demo/jfc/Java2D
```
2. Run the Java2Demo with the following command line switches:  

```
java -client -verbose:gc -XX:+PrintGCTimeStamps -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m -jar Java2Demo.jar
```
3. Observe the command line window for output after launching the applications.
4. Use the *GC Output Interpretation Notes* below to interpret the output.

### GC Output Interpretation Notes

1. GC output contains lines such as this:  
1.035: [GC 2898K->2407K(3388K), 0.0050878 secs]  
And this:

1.050: [Full GC 2407K->2407K(3388K), 0.0425842 secs]

Each line of output indicates the garbage collector has done some work. These lines are interpreted as follows:

- The first column is the number of seconds the application had been running, That is, 1.035 would mean the application has been 1.035 seconds when the garbage collection event occurred.
  - Inside the left '[' and right ']' is the GC activity. The label 'GC' indicates a minor, or young generation GC event type. If this label is 'Full GC', then the GC event type is a full GC event type.
  - The first number after the GC event type, and before the '->' label is the amount of Java heap space, young generation space plus old generations space, being utilized at the time the GC event occurred, i.e. 2898K, where K indicates kilobytes.
  - The number after the '->' is the amount of Java heap space being utilized after the GC event. In other words, if you take the different between the number to the left & right of the '->', you will get the number of bytes reclaimed, or garbage collected.
  - The number inside the '(' and ')' is the size of the overall Java heap space size, young generation space and old generation space, that is. (3008K) means 3008K of Java heap space has been allocated.
  - The number after the ',' trailing the '(' and ')' is the amount of time it took to perform the GC event.
2. If you observe an increasing trend over a period of time in number in either of the values to the left or right of the '->', then the old generation space is filling up. An increasing trend that culminates in a Full GC, and tends to repeat itself thereafter is an indicator Java heap space sizing will improve the performance of the application.

Here's an example that illustrates the pattern:

0.368: [GC 896K->410K(3008K), 0.0075630 secs]  
0.526: [GC 1305K->800K(3008K), 0.0075095 secs]  
0.581: [GC 1695K->1128K(3008K), 0.0047813 secs]  
0.644: [GC 2020K->1337K(3008K), 0.0026936 secs]  
0.678: [GC 2228K->1450K(3008K), 0.0026653 secs]  
0.758: [GC 2343K->1580K(3008K), 0.0024320 secs]  
0.916: [GC 2476K->1791K(3008K), 0.0052515 secs]  
0.980: [GC 2687K->2002K(3008K), 0.0036495 secs]  
1.035: [GC 2898K->2407K(3388K), 0.0050878 secs]  
1.050: [Full GC 2407K->2407K(3388K), 0.0425842 secs]  
1.134: [GC 3046K->2588K(4972K), 0.0026583 secs]  
1.161: [GC 3484K->3217K(4972K), 0.0050143 secs]  
1.210: [GC 3954K->3445K(4972K), 0.0033603 secs]  
1.252: [GC 4341K->3851K(4972K), 0.0042290 secs]  
1.306: [GC 4747K->4044K(4972K), 0.0043967 secs]  
1.416: [GC 4426K->4135K(5100K), 0.0023449 secs]  
1.420: [Full GC 4135K->4135K(5100K), 0.0410667 secs]  
2.000: [GC 5031K->4736K(7856K), 0.0065664 secs]  
2.081: [Full GC 4781K->4735K(7856K), 0.0397819 secs]  
2.832: [Full GC 5018K->4504K(8856K), 0.0454400 secs]

Notice that there's an increasing trend of Java heap space usage, culminating in a Full GC, then it repeats itself several times. Tuning Java heap spaces would likely help the performance of this application.

3. In the previous example output, notice the overall Java heap space, the number inside the '(' and ')' increases in several GC events. What is happening here is the old generation space after a GC event is above a preset threshold of available free Java heap space. As a result, the Java heap space is being re-sized to a larger value.
4. It is possible you might also see output that looks like the following:  
11.742: [Full GC[Unloading class sun.reflect.GeneratedMethodAccessor17] 5572K->5132K(9504K), 0.0574893 secs]

This output indicates a full GC event triggered by permanent generation space running out of space. The garbage collector unloads some unreferenced Java classes in an effort to reclaim permanent generation Java heap space. If sufficient space cannot be reclaimed, the JVM may re-size permanent generation space. If it cannot re-size to a larger needed space, it will throw an `OutOfMemoryError`.

It is also important to note that permanent generation space re-sizing / growing requires a Full GC event. Hence it is important to monitor permanent generation space usage.



## Module 3 Demo 4: Using -XX:+PrintGCDetails

This demo obtains and interprets GC output produced by the -XX:+PrintGCDetails option of the java command.

Slide id: 98

### Required Software

- Any OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objectives

- Use the -XX:+PrintGCDetails option of the java command to obtain GC output.
- Interpret the GC output.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the Java2D demo directory:  

```
cd <JDK install dir>/demo/jfc/Java2D
```
2. Run the Java2Demo with the following command line switches:  

```
java -client -XX:+PrintGCDetails -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m -jar Java2Demo.jar
```
3. Observe the command line window for output after launching the applications.
4. Use the *GC Output Interpretation Notes* below to interpret the output.

### GC Output Interpretation Notes

GC output contains lines such as this:

```
[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K(261184K), 0.0459067 secs][Times: user=0.05, sys=0.02, real=0.03]
```

Each line of output indicates the garbage collector has done some work. These lines are interpreted as follows:

- The 'GC' label in the first column of the output indicates that a minor collection was performed.

Other values for the first column are:

- Full GC indicating a full GC event.
  - Full GC (System) indicating the Full GC was provoked by an explicit Java source code call to System.gc method.
- 
- The output portion DefNew: 64575K->959K(64576K), 0.0457646 secs indicates the minor collection event shown recovered about 98% of the young generation heap space, and took approximately 46 milliseconds.
  - The output portion 196016K->133633K(261184K), 0.0459067 secs indicates a reduction of the entire Java heap usage from by about 51%.
- The time of 0.0459067 secs shows the slight additional overhead for the collection (over and above the collection of the young generation space). This is the likely time associated with stopping and starting both application threads and GC threads along with some additional internal work associated with the collector.
1. The output portion [Times: user=0.05, sys=0.02, real=0.03] shows the user, system and the real CPU (usr + sys) times (used in the GC event) respectively.
  2. The first column is the type of gc event, GC = minor collection event, Full GC = full gc event. It is also possible to see a "Full GC (System)" gc event. This means the Full GC was provoked by an explicit Java source code call to System.gc(). Full GC gc event types may include perm generation space statistics. When permanent generation is re-sized, the output includes permanent generation statistics.

In addition to the value of GC (shown in the sample output) other values for the first column are:

- Full GC indicating a full GC event.
- Full GC (System) indicating the Full GC was provoked by an explicit Java source code call to System.gc method.

Full GC events can include permanent generation space statistics. This happens when the permanent generation is re-sized.

You might also see output such as this:

```
[GC: [DefNew: 8128K->8128K(8128K), 0.0000505 secs]
[Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K),
0.1293306 secs][Times: user=0.02, sys=0.01, real=0.03]
```

This output is interpreted as follows:

1. The 'GC' label in the first column of the output indicates that a minor collection was performed.
2. Notice that the young generation space was not collected, it stayed full at 8128K bytes. This activity took about 50 microseconds (0.0000505 seconds).
3. Notice that there is information shown for a major collection occurring delineated by the 'Tenured' label. The tenured, or old generation usage was reduced to about 10% 18154K->2311K(24576K) and took about .13 seconds, 0.1290354 secs. Again, you can see the amount of CPU time consumed after the 'Times' label.

The following list contains In short, output that includes

- The DefNew: labels indicates a young generation collection. The output will also show how much was collected and how long it took.
- The Tenured: label indicates an old generation collection. The output will also show how much was collected and how long it took
- The Perm: label indicates a permanent generation collection.

Note that on older versions of JDK 6 and on JDK 5, you may not see the [Times: .... ] information when specifying -XX:+PrintGCDetails.

## Collection Patterns to Watch for

When examining GC output you should look out for the following patterns.

- Frequent collections and/or increasing old generation heap sizes.
- Full GC events that include the "(System)" label indicate the application is explicitly calling System.gc(). In general, this is not an advisable practice.
- See permanent generation being re-sized could mean explicit sizing of permanent generation might be useful.
- Many, or often, or lengthy Full GC events imply an application in need of Java heap space tuning.

The goal in tuning the garbage collector and Java heap spaces is to minimize the occurrence of Full GC events, minimize the frequency of GC events in general while maintaining acceptable pause times.

## Module 3 Demo 5: Obtaining Application Stopped Time

This demo obtains and interprets application stopped time when using the serial collector and the concurrent mark sweep collector.

Slide id: 100

### Required Software

- Any OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objectives

- Use the -XX:+PrintGCApplicationStoppedTime and -XX:+PrintGCApplicationConcurrentTime options of the java command to obtain GC output.
- Interpret the GC output.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the Java2D demo directory:

```
cd <JDK install dir>/demo/jfc/Java2D
```

2. Run the Java2Demo with the following command line switches:

```
java -client -XX:+PrintGCDetails -XX:+PrintGCApplicationStoppedTime -  
XX:+UseSerialGC -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m  
-jar Java2Demo.jar
```

3. Observe the command line window for output after launching the applications.

Use the following sample output of application stopped time and the accompanying explanation to interpret the output you observe in your command line window.

Total time for which application threads were stopped: 0.0061061 seconds

Total time for which application threads were stopped: 0.0071550 seconds

Total time for which application threads were stopped: 0.0031720 seconds  
Total time for which application threads were stopped: 0.0019333 seconds  
Total time for which application threads were stopped: 0.0013185 seconds  
Total time for which application threads were stopped: 0.0017162 seconds  
Total time for which application threads were stopped: 0.0039934 seconds  
Total time for which application threads were stopped: 0.0034975 seconds  
Total time for which application threads were stopped: 0.0439696 seconds

The Serial garbage collector specified in step 2 (-XX:+UseSerialGC), stops application threads at each garbage collection. The amount of seconds reported is the amount of elapsed seconds the application was paused due to a garbage collection event.

If you add -verbose:gc to the command line, you will see -verbose:gc output follow the application stopped time output shown in this section.

4. Shutdown the Java2Demo.
5. Restart the Java2Demo, using the following command line:

```
java -client -XX:+PrintGCApplicationStoppedTime -XX:+UseConcMarkSweepGC  
-XX:+PrintGCApplicationConcurrentTime -Xmx20m -Xms10m -Xmn1m  
-XX:MaxPermSize=12m -XX:PermSize=1m -jar Java2Demo.jar
```

6. Observe the command line window for output after launching the applications.

Use should observe output similar to the following:

Application time: 0.7187232 seconds

Total time for which application threads were stopped: 0.0839802 seconds

This output shows:

- The application threads stopped (paused) for about 83 ms.
  - The application ran for about 720 ms since the last gc event.

These two measures together provide an estimate of the gc overhead of the application. For example, in the example output above, you can conclude that the concurrent collector incurred an observed  $0.0839802 / 0.7187232 = 11.7\%$  overhead in this gc event.

These **two switches** (-XX:+PrintGCApplicationStoppedTime and -XX:+PrintGCApplicationConcurrentTime) can be used together to estimate an application's gc overhead at any time while the application is running. You could sum all the values for 'stopped' and 'concurrent' time to obtain an overall gc overhead for an application.

## Module 3 Demo 6: Using jstat to monitor GC

This exercise demonstrates the use of the jstat utility to monitor GC.

Slide id: 103

### Required Software

- Any OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objectives

- Use the jstat utility to monitor GC of a locally executing Java application.
- Use the jstat utility to monitor GC of a remotely executing Java application.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions for Monitoring a Local Application

1. Open a command line window and change directory to the Java2D demo directory:

```
cd <JDK install dir>/demo/jfc/Java2D
```

2. Run the Java2Demo with the following command line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m  
-jar Java2Demo.jar
```

3. Click the transforms tab of the Java2Demo GUI.
4. Obtain the lvmid of the Java2Demo process using the following command on another command window.

```
jps -l
```

You should observe an output (that maps lvmids to executing Java processes) similar to the following:

```
2564
```

2874 Java2Demo.jar  
1948 sun.tools.jps.Jps

Note: The *lvmid* is a platform-specific value that uniquely identifies a JVM on a system. The *lvmid* is the only required component of a virtual machine identifier. The *lvmid* is typically, but not necessarily, the operating system's process identifier for the target JVM process.

5. Use a command line window to monitor the JVM heap spaces using `jstat` with the `-gcutil` option along with the *lvmid* obtained in step 4 (for example 2874) and a reporting frequency of 250 milliseconds.

```
jstat -gcutil 2874 250
```

6. Observe the command line window for the output of the `jstat` utility.

Use the following sample output of application stopped time and the accompanying explanation to interpret the output you observe in your command line window.

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
12.44	0.00	27.20	9.49	96.70	78	0.176	5	0.495	0.672
12.44	0.00	62.16	9.49	96.70	78	0.176	5	0.495	0.672
12.44	0.00	83.97	9.49	96.70	78	0.176	5	0.495	0.672
0.00	7.7	0.00	9.51	96.70	79	0.177	5	0.495	0.673
0.00	7.74	23.37	9.51	98.10	79	0.177	5	0.495	0.673
0.00	7.74	43.82	9.51	98.50	79	0.177	5	0.495	0.673
0.00	7.74	58.11	9.51	98.50	79	0.177	5	0.495	0.673

In the `jstat` output above, the columns have the following meaning:

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.  
P – Permanent space utilization as a percentage of the space's current capacity.  
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.

In the example output above, a young generation garbage collection occurred between the 3<sup>rd</sup> and 4<sup>th</sup> sample which is indicated by the change in the YGC column from 78 to 79. The collection took 0.001 seconds and promoted objects from the eden space (E) to the old space (O), resulting in an increase of old space utilization from 9.49% to 9.51%. Before the collection, the survivor spaces (S0 and S1) were 12.44% utilized, but after this collection they are only 7.74% utilized. Also in this example output, there was an increase in the permanent generation space (P) usage between the 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> reporting intervals. These increases are likely the result of classloading activity.

## Instructions for Monitoring a Remote Application

To monitor remote HotSpot JVMs the `jstatd` daemon is required to be installed and configured on the system where the remote HotSpot JVM is running. The `jstatd` daemon launches a Java RMI server application that monitors for the creation and termination of HotSpot JVMs and provides a interface to allow remote monitoring tools to attach to JVMs running on the local system such as `jstat`. The `jstatd` daemon must be run with the same user credentials as those of the JVMs to be monitored. Since `jstatd` can expose instrumentation of JVMs, it employs a security manager and requires a

security policy file. Consideration should be given to the level of access granted so monitored JVMs are not compromised. The policy file used by jstatd must conform to Java's policy specification. The following is an example policy file which can be used with jstatd.

Perform the following steps on the remote machine.

1. Create a policy file similar to the following example policy file.

```
grant codebase "file:${java.home}/../lib/tools.jar" {  
    permission java.security.AllPermission;  
};
```

Give the file a suitable name such as: jstatd.policy

Note the example policy file will allow jstatd to run without any security exceptions. This policy is less liberal than granting all permissions to all codebases, but is more liberal than a policy that grants the minimal permissions to run the jstatd server. More restrictive security can be specified in a policy to further limit access than this example. However, if security concerns cannot be addressed with a policy file, then the safest approach is to not run jstatd and use the monitoring tools locally rather than connecting remotely.

\*Additional details on how to configure jstatd can be found at  
<http://java.sun.com/javase/6/docs/technotes/tools/share/jstatd.html>\*

2. To use the policy file and start the jstatd daemon, execute the following command at the command line of a terminal window:

```
jstatd -J-Djava.security.policy=<path to policy file>/jstatd.policy
```

3. Open a command line window and change directory to the Java2D demo directory:  
`cd <JDK install dir>/demo/jfc/Java2D`

4. Run the Java2Demo with the following command line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m  
-jar Java2Demo.jar
```

5. Click the transforms tab of the Java2Demo GUI.
6. Obtain the lvmid of the Java2Demo process using the following command on another command window.

```
jps -l
```



Perform the following steps on the local machine.

7. Use a command line window to monitor the JVM heap spaces using jstat with the -gcutil option along with the lvmid obtained in step 6 (for example 2874) and a reporting frequency of 250 milliseconds.

```
jstat -gcutil 2874 250
```

8. Observe the command line window for the output of the jstat utility.

## Module 3 Demo 7: Using JConsole

This exercise introduces the JConsole utility and demonstrates its monitoring capabilities.

Slide id: 107

### Required Software

- Any OS
- JDK 1.5 or higher
- JConsole (included with JDK 1.5 or higher distribution)  
Located in: <JDK install dir>/bin directory
- Java2Demo  
Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objectives

- Develop competency in using the JConsole utility to monitor a Java application.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:  

```
java -version
```

  
If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.
2. Download and install the latest version of Java SE from  
<http://java.sun.com/javase/downloads/index.jsp>.  
Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

JConsole is a JMX (Java Management Extensions) compliant GUI tool which can connect to a running Java 5.0 version or later JDK. The following steps illustrate how to connect JConsole to an example demo application shipped with the JDK called Java2Demo:

1. Open a command line window and change directory to the Java2D demo directory:  

```
cd <JDK install dir>/demo/jfc/Java2D
```
2. Run the Java2Demo with the following command line switches:  

```
java -Dcom.sun.management.jmxremote -jar Java2Demo.jar
```

  
Note: If you are using JDK 6 and later you do not require the -Dcom.sun.management.jmxremote property which allows JConsole to connect to the Java application.
3. Start JConsole by entering the following command on a command line window.  

```
jconsole
```

When JConsole is launched it automatically discovers running Java applications and presents a connect to dialog.

4. Connect to the Java2Demo application.

Use the Connect button on the dialog to navigate to the New Connection dialog and select the Java2Demo application.

5. When JConsole connects (to an application) it loads a set of six tabs. Use the Memory tab to observe heap memory spaces.

## Module 3 Demo 8: Examining VisualVM Capabilities

This exercise demonstrates the monitoring capabilities of the VisualVM utility.

Slide id: 111

### Required Software

- Any OS
- JDK 1.6 or higher

Note: Although you must launch VisualVM using JDK 1.6 or higher, but VisualVM can monitor applications using JDK 1.4.2 and later. The more recent the JDK running the monitored application, the more VisualVM features are available. (For additional information, see the feature matrix on <https://visualvm.dev.java.net>).

- VisualVM

VisualVM is included in Java SE 6 update 7 (jdk 1.6.0\_u7). If you are running an earlier version of Java SE you can download VisualVM from: <http://visualvm.dev.java.net>.

- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Examine the GC monitoring capabilities of the VisualVM utility.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 6 update 7 or later. To check enter the following command on a terminal window:  
  
`java -version`  
  
If the response is: java version "1.6.0\_07" or later you can bypass the following VisualVM installation step.
2. Down load and install VisualVM from <http://visualvm.dev.java.net>.
3. Add the bin subdirectory of the VisualVM install directory to the OS PATH variable.

### Instructions

1. Open a command line window and and change directory to the Java2D demo directory:  
  
`cd <JDK install dir>/demo/jfc/Java2D`
2. In a different command line window, run the Java2Demo. It is located in the <JDK install dir>/demo/jfc/Java2D/ directory.  
  
`java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m -jar Java2Demo.jar`
3. Launch VisualVM

- If you are using Java SE 6 update 7 or later run the following command in a command line window:

jvisualvm

6. If you had installed VisualVM in step 1, run the following command in a command line window:

visualvm

Examine the resulting VisualVM GUI. You should notice the following tabs: Start Page; Application.

4. Examine the Start Page tab. This tab displays the first time you launch VisualVM. It contains links to a set of user guides.

#### 9. Applications tab

The Applications tab contains three nodes:

4. Local , Remote and Snapshot. The local shows auto-discovered Java processes which can be monitored.

5. Examine the Applications tab and notice that it contains three nodes:

- Local

The Local node shows auto-discovered Java processes which can be monitored. Specific JMX connections can be added locally (or remotely).

- Remote

The Remote node lists Java processes running on remote hosts. Monitoring remote JVMs require either specific JMX connection and or the jstatd daemon to be running on the remote machine. If the jstatd daemon is running on the remote machine, Java processes which can be monitored are automatically discovered and displayed under the remote hosts.

3. Snapshot

The Snapshots node enables profiler snapshots to be saved and re-opened later.

If you right click on the Java2Demo, you will see there are options to; capture a thread dump, heap dump, profile, take an application snapshot, enable a heap dump on an out of memory error and open which will open a panel on the right. Thread dumps and heap dumps are added below the Java process node on the right. They can be analyzed, renamed or removed at any time.

6. Select and open the Java2Demo application. You should notice an overall panel for the Java2Demo application open on the right containing 4 sub-panels. The number of sub-panels displayed for a monitored application varies depending on the version of the JVM running the application and whether the application is remote or local. (See the feature matrix on <https://visualvm.dev.java.net>).

If you are running the Java2Demo as a local application running on a Java 6 JVM, you should see four sub-panels; Overview, Monitor, Threads, Profiler.

7. Open and examine each (VisualVM) sub-panel of the Java2Demo application.

The Threads panel provides a picture of what Java threads are doing what in the application. The Profiler tab provides the ability to do either CPU profiling or heap (memory) profiling. Any saved profiled snapshots will be placed under the Snapshots node in the left panel.

8. You should note the following.

VisualVM can be custom extended via plug-in module for any application to be monitored. Select the Tools > Plugins option. If selected, you'll see some already bundled plug-ins. A custom plugin could be developed and installed using this mechanism too.

An existing JConsole plug-in can be integrated and used in VisualVM by installing the VisualVM-JConsole plug-in.

9. Exit the Java2Demo and the VisualVM application.

## Module 3 Demo 9: Examining VisualGC Capabilities

This exercise demonstrates the monitoring capabilities of the VisualGC plugin.

Slide id: 113

### Required Software

- Any OS
- JDK 1.6 or higher

Note: Although you can launch VisualVM using JDK 1.4.2, the use of a later version of the JDK will enable the use of additional VisualVM features. (For additional information, see the feature matrix on <https://visualvm.dev.java.net>).

- VisualVM

VisualVM is included in Java SE 6 update 7 (jdk 1.6.0\_u7). If you are running an earlier version of Java SE you can download VisualVM from: <http://visualvm.dev.java.net>.

- VisualGC

Visual GC is available as a plugin for VisualVM.

- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Examine the monitoring capabilities of the VisualGC plugin for VisualVM.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 6 update 7 or later. To check enter the following command on a terminal window:  
`java -version`  
If the response is: java version "1.6.0\_07" or later you can bypass the following VisualVM installation step.
2. Download and install VisualVM from <http://visualvm.dev.java.net>.
3. Add the bin subdirectory of the VisualVM install directory to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the Java2D demo directory:  
`cd <JDK install dir>/demo/jfc/Java2D`
2. In a different command line window, run the Java2Demo. It is located in the <JDK install dir>/demo/jfc/Java2D/ directory.  
`java -client-Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m -jar Java2Demo.jar`

3. Click the Transforms tab of the Java2Demo application to increase object allocation. This step is optional, but it will increase the frequency full GC events.

4. Launch VisualVM

- If you are using Java SE 6 update 7 or later run the following command in a command line window:

```
jvisualvm
```

- If you had installed VisualVM in step 1, run the following command in a command line window:

```
visualvm
```

Examine the resulting VisualVM GUI. You should notice the following tabs: Start Page; Application.

5. Perform the following step if you have not already installed the VisualGC plugin into VisualVM.

Within Java VisualVM, download and install the VisualGC plug-in by selecting Tools > Plugins menu option.

6. Note the process id of the Java2Demo on Applications panel and select that as the process to monitor in the VisualGC panel on the right.

7. Observe the VisualGC panel.



## An Overview of the VisualGC Display

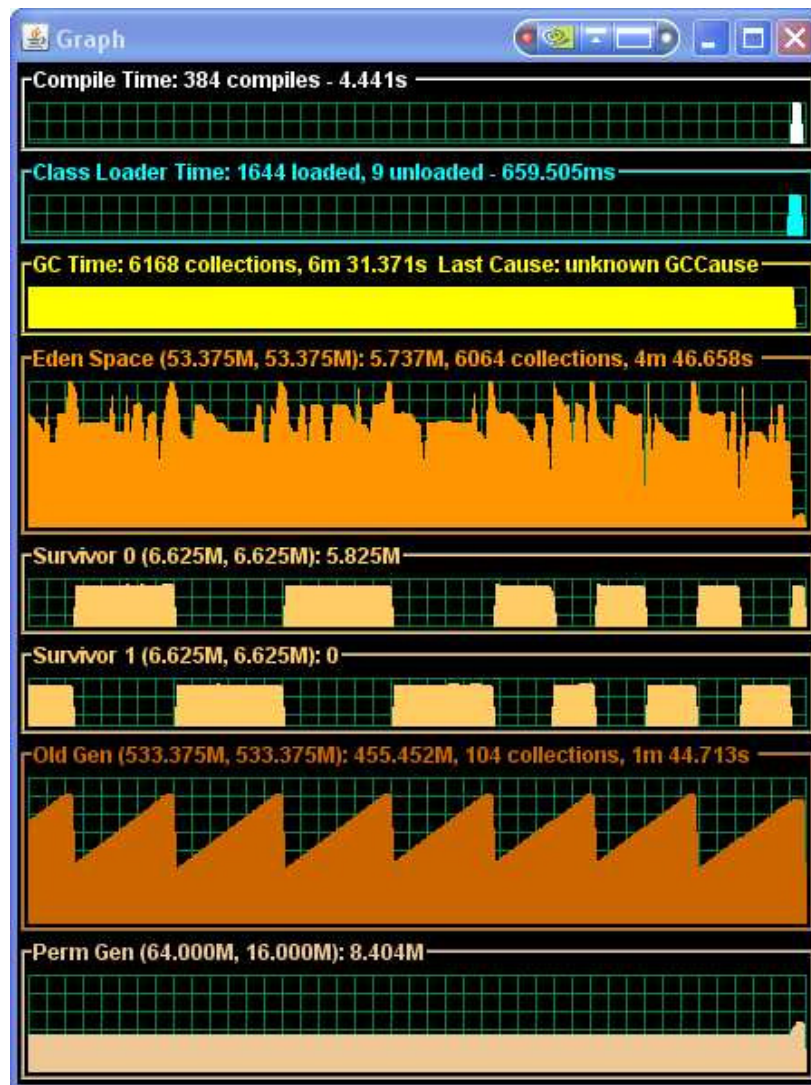
VisualGC will display two or three windows in the right panel of VisualVM depending the garbage collector being used by the monitored JVM. When the throughput collector is used, VisualGC will show only two windows; the VisualGC window and the Graph window. Otherwise, when the serial collector or concurrent collector is being used a third window, the Survivor Age Histogram window is also shown. The upper leftmost window of VisualGC is called the VisualGC window. It is comprised of the following two panels.



The top panel, labeled Application Information contains a live-ness indicator and the elapsed time since the start of the target Java application. It also contains a scrollable text area that lists miscellaneous information about the configuration of the Java application being run and the JVM being monitored. This section includes the target application's main class name (or jar file if the application was started with the -jar option), and the arguments to the class's main method. It also includes the arguments passed to the JVM being monitored and the values of certain Java properties exported as instrumentation objects. The bottom panel provides a graphical view of the garbage collection spaces. This panel is divided into three vertical sections, one for each of the of the spaces:

Perm (Permanent) space, Old (or Tenured) space, and the young generation section, (the right most vertical slice of this panel). The young generation section is comprised of three spaces; an Eden space and two Survivor spaces, S0 and S1. The screen areas representing all these spaces are sized proportionately to the maximum capacities of the spaces as they are allocated by the JVM. The screen area real estate representing the spaces in the window are of fixed size and do not vary over time. Each space is filled with a unique color indicating the current utilization of the space relative to its maximum capacity. A unique color for each space is used consistently among this window, the Graph window and Survivor Age Histogram window. The memory management system within the JVM is capable of growing and shrinking the garbage collected heap. This is accomplished by reserving memory for the requested maximum Java heap size but committing real memory to only the amount currently needed. The relationship between committed and reserved memory is represented by the color of the background grid in each space. Uncommitted memory is represented by the dark gray colored portion of the grid whereas committed memory is represented by the green colored portion of the grid. In many cases, the utilization of a space may be nearly identical to the committed amount of memory making it difficult to determine the exact transition point of the green portion of the grid to the gray portion of the grid. The relationship between the sizes of the Eden and Survivor spaces in the young generation portion of the VisualGC frame are usually fixed in size. The two survivor spaces are usually identical in size and fully committed. The Eden space may be only partially committed, with the uncommitted portion of the space represented by dark gray portion of the grid. When the throughput collector (-XX:+UseParallelGC or -XX:+UseParallelOldGC) is used along with the adaptive size policy feature (-XX:+UseAdaptiveSizePolicy), which is enabled by default, the relationship or ratio between the sizes of the young generation spaces can vary over time. When the adaptive size policy is in effect, the sizes of the of the survivor spaces are may not be identical and the space in young generation can be dynamically redistributed among the three spaces. In this configuration, the screen areas representing the Survivor spaces and the colored region representing the utilization of the space are sized relative to the current size of the space, not the maximum size of the space. When adaptive resizing occurs, the screen area associated with the young generation spaces will update accordingly.

The Graph window, shown below, which is to the right of the VisualGC window, plots performance statistics as a function of time for a historical view. This window displays garbage collection statistics along with dynamic (JIT) compiler and class loader statistics. The latter two statistics will be discussed later in this chapter. The resolution of the horizontal axis in each display is determined by the interval command line argument. Each sample occupies 2 pixels of screen real estate. The height of each display depends on the statistic being plotted.



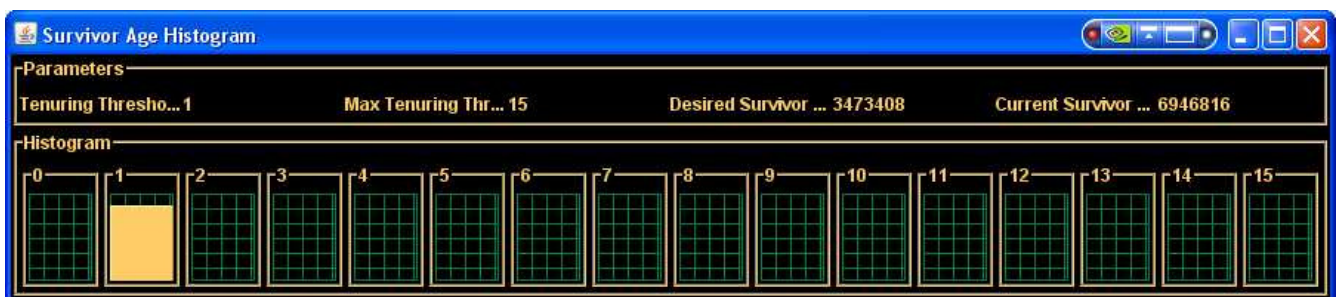
The Graph window has the following displays:

- Compile Time – discussed a later in this chapter
- Class Loader Time – discussed a little later in this chapter
- GC Time – Displays the amount of time spent in garbage collection activities. The height of this display is not scaled to any particular value. A non zero value in this graph indicates that garbage collection activity occurred during the last interval. A narrow pulse indicates a relatively short duration and a wide pulse indicates a long duration. The title bar indicates the total number of GC events and the accumulated GC time since the start of the application. If the JVM being monitored maintains the `hotspot.gc.cause` and `hotspot.gc.last_cause` statistics, the cause of the most recent GC event will also be displayed in the title bar.
- Eden Space – Displays the utilization of the Eden Space over time. The Eden Space is one of three spaces that make up the young generation space. The height of this display is fixed and, by default, the data is scaled according to the current capacity of the space. The current capacity of the space can change depending on the collector being used as the space shrinks and grows over time. The title bar displays the name of the space and its maximum and current capacity in parenthesis followed by the current utilization of the space. In addition, the

- title also contains the number and accumulated time of minor garbage collections.
- Survivor 0 and Survivor 1 – Displays the utilization of the two survivor spaces over time. The Survivor Spaces are the remaining two spaces which comprise the Young Generation space. The height of each of these two displays is fixed and, by default, the data is scaled according to the current capacity of the corresponding space. The current capacity of these spaces can change depending on the collector being used as a space shrinks and grows over time. The title bar displays the name of the space and its maximum and current capacity in parenthesis followed by the current utilization of the space.
- Old Gen – Displays the utilization of the Old Generation space over time. The height of the display is fixed and, by default, the data is scaled according to the current capacity of the space. The current capacity of this space can change depending on the collector being used as the space shrinks and grows over time. The title bar displays the name of the space and its maximum and current capacity in parenthesis followed by the current utilization of the space. In addition, the title also contains the number and accumulated time of full garbage collections.
- Perm Gen – Displays the utilization of the Permanent Generation space over time. The height of the display is fixed and, by default, the data is scaled according to the current capacity of the space. The current capacity of this space can change depending on the collector being used as the space shrinks and grows over time. The title bar displays the name of the space and its maximum and current capacity in parenthesis followed by the current utilization of the space.

The Eden Space, Survivor 0, Survivor 1, Old Gen and Perm Gen displays can be updated to show reserved memory utilization by right clicking the mouse over any of these spaces in the Graph window and checking the “Show Reserved Space” box. The default is to show committed memory utilization and can be switched back by right-clicking and unchecking the “Show Reserved Space” box. In reserved mode the data is scaled according to the maximum capacity of the space while in committed mode the data is scaled according to the current capacity of the space. In reserved mode, the background grid is painted in dark gray to represent that portion of the reserved memory that is uncommitted memory and in green to represent the portion that is committed.

The Survivor Age Histogram Window, shown below, which is displayed below the VisualGC window and Graph window when the low pause collector or serial collector is being used by the monitored JVM shows object survivor statistics. The throughput collector does not maintain a survivor age since it uses a different mechanism for maintaining objects in the survivor spaces. Hence, the Survivor Age Histogram Window is of no value and is not displayed when monitoring a JVM which is using the throughput collector.



## Module 3 Demo 10: Examining JIT Compilation Activity

This exercise uses various tools to examine the JIT compilation activity.

Slide id: 116

### Required Software

- Any OS
- JDK 1.6 or higher

Note: Although you can launch VisualVM using JDK 1.4.2, the use of a later version of the JDK will enable the use of additional VisualVM features. (For additional information, see the feature matrix on <https://visualvm.dev.java.net>).

- VisualVM

VisualVM is included in Java SE 6 update 7 (jdk 1.6.0\_u7). If you are running an earlier version of Java SE you can download VisualVM from: <http://visualvm.dev.java.net>.

- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Examine the JIT compilation activity using with JConsole, VisualVM, VisualGC, jstat tools and -XX:+PrintCompilation command line option.

### Background

There are several ways to monitor HotSpot dynamic (JIT) compilation activity. Although the result of dynamic compilation results in a faster running application, dynamic compilation requires computing resources such as CPU cycles and memory to do its work. Hence, it is useful to observe dynamic compiler behavior. Monitoring dynamic compiler activity is also useful when wanting to identify methods which are being optimized or in some cases de-optimized and re-optimized. A method can be de-optimized and re-optimized when the dynamic compiler has made some initial assumptions in an optimization which later turned out to be incorrect. To address this scenario, the HotSpot dynamic compiler will discard the previous optimization and re-optimize the method based on the new information it has obtained.

### Preparations

Verify and if required download and install VisualVM.

1. Verify if you are running Java SE 6 update 7 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version "1.6.0\_07" or later you can bypass the following VisualVM installation step.

2. Download and install VisualVM from <http://visualvm.dev.java.net>.
3. Add the bin subdirectory of the VisualVM install directory to the OS PATH variable.

Install VisualGC plugin for VisualVM.

1. Launch VisualVM

If you are using Java SE 6 update 7 or later run the following command in a terminal window:

```
jvisualvm
```

If you had installed VisualVM in step 1, run the following command in a terminal window:

```
visualvm
```

2. Within Java VisualVM, download and install the VisualGC plug-in by selecting Tools > Plugins menu option.

On the Available Plug-ins tab, select the VisualGC Plug-in and press the "Install" button. Click through the installer wizard to install VisualGC plug-in.

Close the Plug-ins window.

3. Exit VisualVM.

## Instructions

1. Open a command line window and change directory to the Java2D demo directory:

```
cd <JDK install dir>/demo/jfc/Java2D
```

2. In a different command line window, run the Java2Demo. It is located in the <JDK install dir>/demo/jfc/Java2D/ directory.

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m  
-jar Java2Demo.jar
```

3. Start JConsole by entering the following command on a command line window.

```
jconsole
```

When JConsole is launched it automatically discovers running Java applications and presents a connect to dialog.

4. Connect to the Java2Demo application.

5. Click on the VM Summary tab in JConsole and look at the values for Total Compile time, JIT Compiler, Process CPU Time and Uptime. These fields provide a sense of how many CPU cycles are being spent making optimizations.

6. Examine the attributes accessible through MBean instances on the MBeans tab, under the java.lang.Compilation.Attributes node.

7. Exit JConsole.

8. Launch VisualVM

- If you are using Java SE 6 update 7 or later run the following command in a command line window:

```
jvisualvm
```

- If you had installed VisualVM in step 1, run the following command in a command line window:

```
visualvm
```

9. Click on the Java2Demo list item in the left panel Java2Demo monitoring information on



the right.

You should observe the JVM listed as the `HotSpot Client VM` on the `Overview` tab.

10. Click on the `VisualGC` tab and select the `Java2Demo` as the application to monitor.
11. Watch the `Compile Time` panel on the right. You will observe a spike or blip associated with each time a compilation activity is takes place.
12. Go to the `Java2Demo` GUI and slowly click on each tab. Observe what occurs in the `Compile Time` panel of `VisualGC`. As optimizations are performed by the JVM's JIT compiler, you will observe blips or spikes in `VisualGC`'s `Compile Time` panel. The `Compile Time` panel will also display the number of compilations and the time spent doing the optimizations.
13. Quit the `VisualVM` utility.
14. Run the `jps` command (with the `-l` option) from a command line window to obtain the `vmid` of the executing `Java2Demo` application.

```
jps -l
```

15. Use the `vmid` obtained in the previous step, to run the `jstat` command as follows:

```
jstat -printcompilaton <vmid> 500
```

The `jstat` utility prints the last JIT compiled method in the 500 milli second duration window. This information is of limited interest since it basically indicates the JIT compiler did something.

16. Quit `jstat` and shutdown the `Java2Demo` application.
17. Restart the `Java2Demo` with the `-XX:+PrintCompilation` flag:  

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m  
-XX:PermSize=1m -XX:+PrintCompilation -jar Java2Demo.jar
```

The `-XX:+PrintCompilation` option prints output for every JIT compiled method. The next overhead slide provides an explanation of the output.

## Instructor Notes

To get further information about what the JIT compiler is doing, you can use a *debug* JVM. Debug JVMs are special builds with additional troubleshooting capabilities. With a debug JVM you can see the optimized assembly code using the `-XX:+PrintOptoAssembly`. But, if you are not well versed in understanding assembly language, you may want to enlist some help from a JIT compiler engineer. If the instructor feels comfortable talking about assembly language, a debug version of the JVM could be installed and the `Java2Demo` program started with the additional `-XX:+PrintOptoAssembly`.

# Module 4 Demo 1: Application Profiling Using NetBeans Profiler

This exercise performs application profiling using the NetBeans profiler.

Slide id: 123

## Required Software

- Any OS
- JDK 1.6 or higher
- NetBeans version 6.1 or later.

You can download NetBeans from: <http://www.netbeans.org>

- NetBeans Project  
Java2D project in the demos/mod04/sw directory

## Objective

- Introduce NetBeans Profiler to profile Java technology applications.

## Preparations

Check for required version of Java SE.

1. Use the `java -version` command to check the installed Java SE version.

If the installed version is earlier than Java SE 6, download and install the latest Java SE version from the following URL.

<http://java.sun.com/javase/downloads/index.jsp>

Download and install NetBeans 6.1 (***MAKE SURE TO DOWNLOAD THE FULL JAVA EE NETBEANS IDE THAT CONTAINS THE TOMCAT WEB CONTAINER AND GLASSFISH***).

2. Download and and install NetBeans from the following URL:

<http://www.netbeans.org>

3. Add the `bin` subdirectory of the NetBeans 6.1 install directory to the OS PATH variable.



## Instructions

This demo is essentially the same as the demo at:

<http://wiki.netbeans.org/ProfilerIntroDemoAndTour>

But it has been updated use NetBeans IDE 6.1 (with Java EE support).

## Main Points to Hit

- The NetBeans Profiler has a variety of powerful and easy to use capabilities to help you track down performance problems.
- Using root method selection you can limit the overhead imposed by the profiler to only those parts of an application that you suspect have a problem.
- With its tight integration into the developer work flow, it is easy to select which parts of your application should be profiled and to start/stop profiling sessions.

## Prerequisites

- JDK5 or JDK6
- NetBeans IDE 6.1 with Java EE support

## Setup (IMPORTANT: Read this before doing the demo!!!)

1. Unzip the Java2D project that's in the demos/mod04/sw directory. It is from the JDK sample programs.
2. Open the Java2D project in NetBeans IDE
3. If you get a chance as part of training preparation, profile the Java2D project at least once so that the profiler can modify the build.xml file.
4. Again, if you get a chance as part of training preparation, right before starting the demo, run the Java2D project at least once - this makes subsequent startups a bit quicker since much of the program is still in the CPU and disk caches.

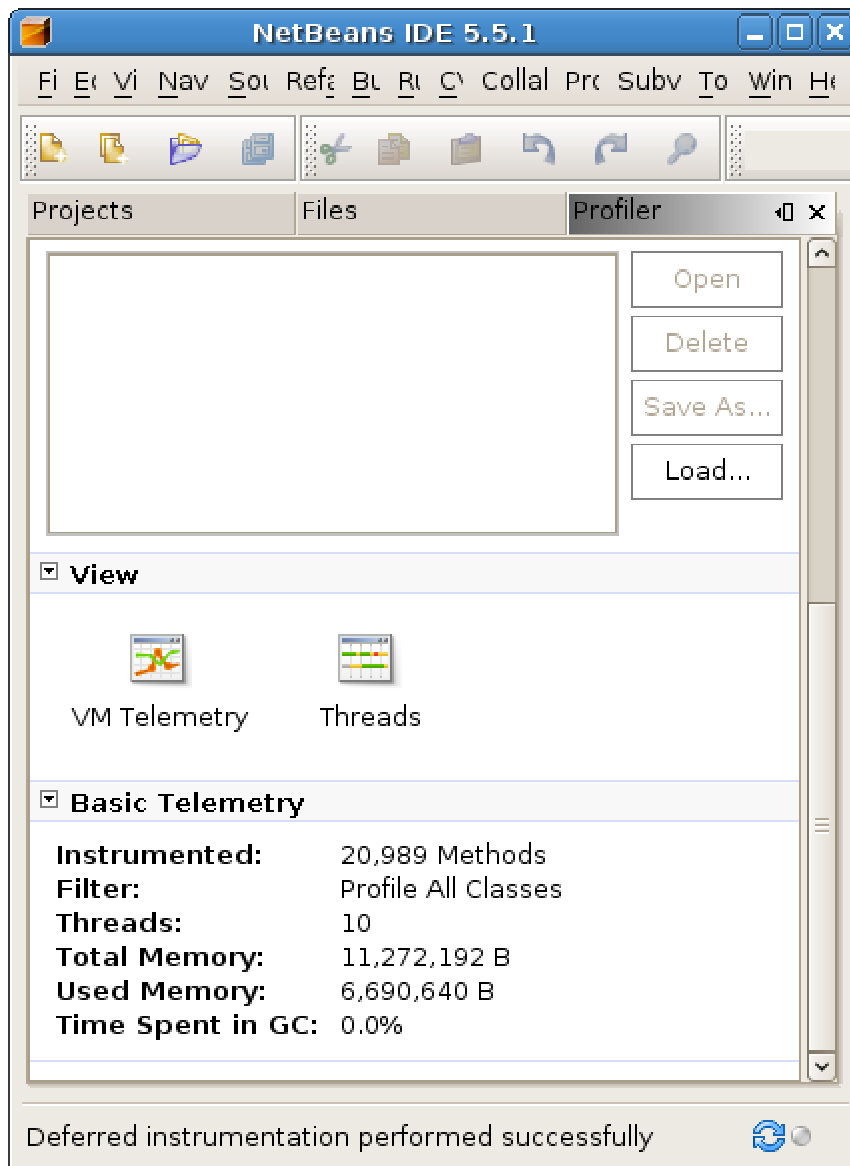
## Demo Steps

### Profile the Entire Application

1. Run the Java2D project with the Run command (no profiling). Suggested Comment (SC): "This is a sample application from the JDK. Notice how fast it starts up."
2. Shutdown the Java2D application. SC: "For those of you who missed it, I will do this again. Watch closely as I click Run... we get a progress meter and then we're up and running."
3. Click on a few of the tabs in the application. Click on the Colors tab last. SC: "This part of the application looks interesting - the animation at the bottom has some statistics that look interesting, for example Frames Per Second (FPS)."
4. Shutdown the Java2D application. SC: "One of the problems we are trying to solve with the NetBeans Profiler most profilers profile the entire application. That's an intrusive operation and it complicates the analysis with lots of additional data to sift through. We want you to be able to profile only the parts of your application that are really of interest. Of course, you can, if you want, profile the entire application."
5. Right-click the Java2D project and choose Profile Project
6. Click the large Analyze Performance link in the CPU box on the left to expand it (if it is not already expanded) SC: "Notice here you have the option to profile the entire application of

parts of the application. You can also create and modify filters to profile a smaller sub-set of the application. The filters apply to both profiling of the entire application and part of the application. A commonly used filter is "Exclude Java core classes."

7. Select Entire Application radio button to profile the entire application.
8. For a Filter, select Profile All Classes. SC: "Just to show you the impact, I will profile all methods of all classes - so this is everything in the application itself and all libraries that it uses. Watch what happens when I click Run."
9. Click the Run button. A couple of dialogs go by for the profiler. Eventually the application's progress meter appears. SC: "The profiler gets started and then eventually it starts the application. Notice the difference in how long we wait for the progress meter to complete... and we wait. And we wait some more." :-)
10. Click a few of the tabs again. Click the Colors tab last. The animation will usually be a bit sluggish looking initially and then it smooths out. SC: "We are seeing a bit of an impact on performance - the application took much longer to start and does not respond to mouse clicks as quickly."
11. The FPS (frames per second) value **might** be different when doing profiling - it depends upon the version of the JDK you are using, the operating system's video driver, and your graphics card. If the value is consistently lower then point this out.
12. Switch back over to the IDE. Scroll the Profile window until the Basic Telemetry information is visible. SC: "Notice the number of methods that are being profiled - over 20,000. And this is a small sample application. With a real-world application that number could be 10x or more."



13. Click the Live Results button in the Profile window. Scroll through it to show how many rows it has. SC: "The other problem with profiling everything is that you end up with too much information to wade through in order to find the cause of performance problems."

Hot Spots - Method	Self time [%]	Self time	Invocations
sun.java2d.pipe.AlphaColorPipe.renderPathTile (Object, byte[], i...	22.9%	58786 ms	6376814
java.lang.System.gc ()	18.2%	46720 ms	365
sun.dc.pr.PathFiller.writeAlpha (byte[], int, int, int)	10.1%	25973 ms	1932364
sun.java2d.pipe.DuctusShapeRenderer.renderPath (sun.java2d....	3.3%	8523 ms	381166
sun.java2d.pipe.AlphaPaintPipe.renderPathTile (Object, byte[], i...	2.9%	7497 ms	471277
java.awt.GradientPaintContext.clipFillRaster (int[], int, int, int, int...	2%	5128 ms	893520
sun.dc.pr.Rasterizer.setOutputArea (float, float, int, int)	1.9%	4982 ms	380882
java2d.intro\$Surface\$DdE.render (int, int, java.awt.Graphics2D)	1.6%	4165 ms	756
sun.java2d.pipe.DuctusRenderer.createShapeRasterizer (java....	1.4%	3474 ms	381166
java.lang.Math.min (int, int)	1.1%	2900 ms	15987624
java.util.concurrent.locks.ReentrantLock.unlock ()	1.1%	2863 ms	7617429
sun.awt.SunToolkit.awtUnlock ()	1.1%	2801 ms	7617076
java.util.concurrent.locks.ReentrantLock\$NonfairSync.lock ()	1.1%	2782 ms	7617429
java.util.Arrays.binarySearch0 (Object[], int, int, Object, java.util....	1.1%	2770 ms	1471084
java.util.concurrent.locks.ReentrantLock.lock ()	1.1%	2767 ms	7617429
java.util.concurrent.locks.AbstractQueuedSynchronizer.release (int)	1.1%	2751 ms	7617429

[Method Name Filter]

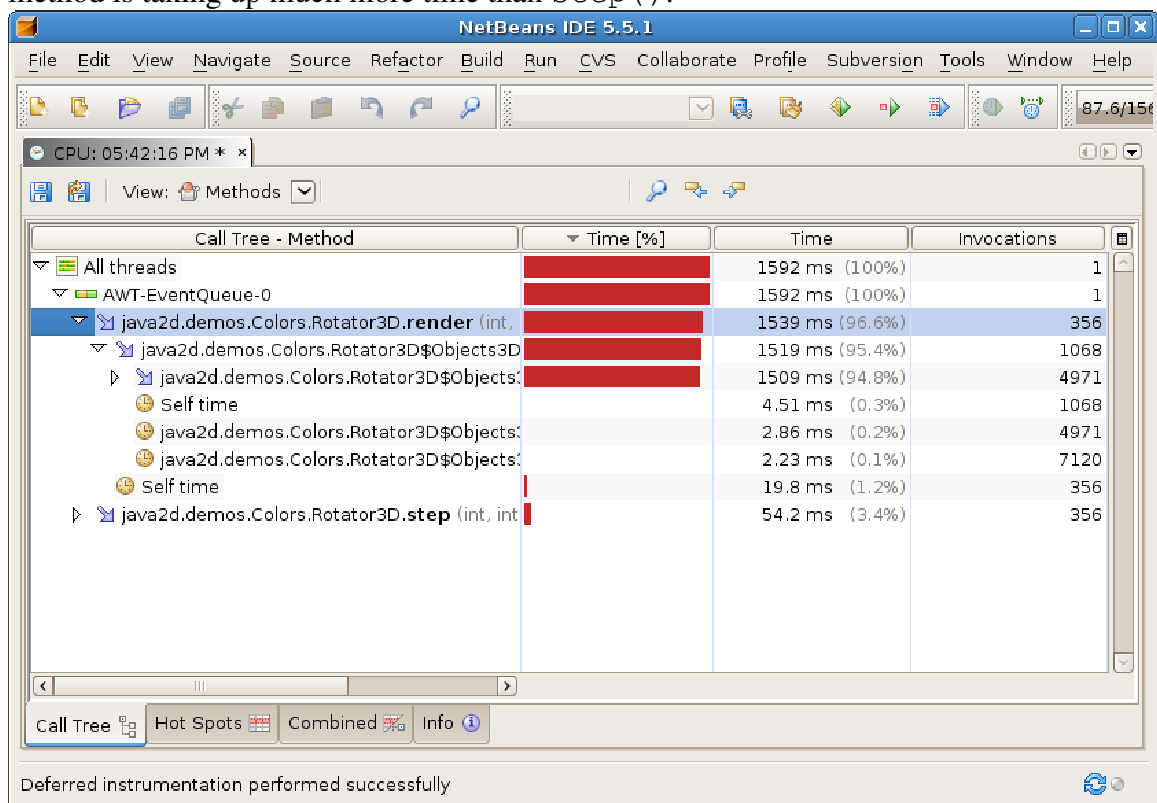
Deferred instrumentation performed successfully

14. Select Profile > Stop.
15. Right click on the Java2D project in the Projects Window and select Profile. Again, profile the entire application, but this time use the “Exclude Java core classes” filter. Then, press Run. SC: “This is a more common scenario you see with other profilers. That is, applying some kind of filter.”
16. Click the Live Results button in the Profile window. Compare the output to what was shown before. Notice that the core Java classes are not shown. SC: “We can see this filter capability has helped in two ways. We have fewer classes to sift through and the profiling is less intrusive on the application.
17. Select Profile > Stop.
18. SC: "I am going to shut this down and show you an even more powerful technique. As you've seen, what some profiling tools force you to do is define filters of package and/or class names for what should or should not be instrumented by the profiler. As I've just shown, the NetBeans Profiler supports filters, but it also has a more powerful feature: root methods."

## Profiling Root Methods

1. In NetBeans 6.1:
  - In the Projects window expand the source packages under Java2D until you can see the `java2d.demos.Colors.Rotator3D` class.
  - Open the source file for `java2d.demos.Colors.Rotator3D` in the NetBeans editor.
  - In the editor window, right-click the `render()` method and choose Profiling > Add as Profiling Root Method. Add them to the Analyze Performance configuration. Do the same thing for the `step()` method. SC: "If all I am interested in is checking the performance of the methods used to do that animation on the Colors tab then I can just navigate to those methods in the IDE (a key advantage of an *integrated* tool) and then select them as what the profiler calls root methods."
1. Go back to the Projects Window, select the Java2D project, right click and select Profile.
2. In the Profile Java2D / Analyze Performance dialog click Part of Application, then click the Run button

- Startup should be much faster. SC: "Now watch what happens when I profile only the part of the application for which I need information: startup is much faster - closer to what we saw with no profiling."
- Click some tabs. Click the Colors tab last.
- Switch back to the IDE. Scroll the Profile window until the Basic Telemetry information is visible. SC: "Now notice the number of methods that are being profiled, much smaller. This is because the profiler examined the application, starting at the root methods that I selected. It instrumented those root methods and then examined the methods that they call and instrumented them and then it examined the methods that those methods call, etc. It does this until it has figured out the call graph that originates at the root methods - those are the only methods that get instrumented. Everything else runs at full speed."
- Click the Take Snapshot button in the Profile window. SC: "Even more important - with less instrumentation we have less data to wade through. We can easily see that the `render()` method is taking up much more time than `step()`."



- SC: "Let's go back and show some of the other options available with the NetBeans Profiler in the editor." Stop the profiler using either the "X" button in the Profile Controls panel. Or, using Profile > Stop Profiling Session. Go back to the editor window where the Rotator3D.java source file is displayed, right click on the render() method and select Profiling > Insert Profiling Point. Here you have several options of actions the Profiler can take when it hits this method." Click select each one and you'll see the action it will take in the "Description" dialog. There are uses cases where each of these options may be useful.

## Viewing Threads

- Restart the Profiling session again by selecting Profile > Rerun Profiling Session.
- Go to the Profile window in the upper left panel, scroll down to find the "View" sub-panel. Click the "Threads" icon. Then, enable threads monitoring on the right panel that is displayed

in order to enable collection of thread information. SC: "The goal of this part of the demo is provide a quick tour so let's look at some additional features. I can see thread state very easily; with the state information color-coded."

3. Select and double click a thread such as the AWT-EventQueue thread. SC: "I can drill into a specific thread to see a break down of how much time it has spent in each state."
4. Click the Details tab in the thread window to switch to the text view. SC: "I can see the state changes for a thread."

## Modify Profiling

1. Select Profile > Modify Profiling Session. Click the Monitor button on the left and then click OK. SC: "One of the nicest features is that I can change the profiler settings on-the-fly, without having to restart my application. I have just switched to a monitor-only mode, so there is no instrumentation at all now - the entire application is running at full speed."
2. Select Profile > Modify Profiling. Notice by clicking on any of the 3 buttons on the left you can modify the profiling session.
3. To show heap profiling, click on the Memory button. SC: "In addition to reporting CPU performance, the profiler can do detailed monitoring of your application's memory usage. This is especially helpful for tracking down memory leaks. (Note: a full demo of memory leak detection is done a little later in this module). It can also help identify areas of excess object allocations. Excess object allocations can be an issue when frequent garbage collection events occur, or there's an effort to reduce the number or frequency of garbage collections."
4. SC: "Also notice you can fine-tune the profiler's settings by defining a custom configurations, filters, etc."
5. Click the Cancel button to close the Modify Profiling dialog
6. Select Profile > Stop Profiling Session. SC: "It is important to note that while this demo was done with a desktop Java application, there are other types of applications that can easily be profiled with the NetBeans Profiler such as web applications."

## Web Application Profiling

1. Select File > New Project. Expand the Samples category and select Web. Then select the Tomcat "Servlet Examples" project and click Next. SC: "Let's create a project that contains the standard Tomcat servlet examples." Note this issue, [Can't Profile Tomcat-targeted applications](#), on Mac OS X. Use GlassFish instead.
2. Change the project name if you want and then click Finish.
3. Right-click the Tomcat ServletExamples project and choose Profile.
4. Click OK on the dialog that asks for permission to modify the build.xml. SC: "The default project system for Java projects in the NetBeans IDE is based on Ant. Here I will click on OK to allow the IDE to modify the build.xml in order to add a target for running the profiler."
5. The Profile ServletExamples dialog is displayed. Click the CPU button on the left if it is not already selected.
6. Select the Entire Application option. Choose to filter to profile only project classes.
7. It's ok if the "Use defined Profiling Points" is enabled. There are none defined at the moment. But, they could be defined by loading the Project files in the NetBeans editor.
8. Click on the "Run" button. It'll take about 30 – 60 seconds to start and deploy the application to GlassFish application server. Once it is deployed, the web browser will automatically display the deployed Tomcat Servlet Examples.
9. Switch to the browser after it displays <http://localhost:8080/servlets-examples/> and show that the application is running. SC: "So this was basically the same as running the application - the IDE started the server, started the browser, etc. Only now, we're profiling. :-) NetBeans Profiler has this same level of one-click profiling support for other application servers such as JBoss. If an application server is not supported directly, you can use the "Attach Profiler"

feature to attach to the application server once it is up and running. “Attach Profiler” is the next shown feature.

10. Feel free to click on “Live Results” in the Profiler window in the upper left or take a snapshot.
11. You might also notice that the NetBeans IDE shows an HTTP Monitor window at the bottom of its display. The HTTP Monitor shows every HTTP GET/POST operation and its response. This can be useful in debugging HTTP request / response flow between an application server and a web browser.
12. Select Profile > Stop Profiling Session.

## **Attach Profiler**

1. Select Profile > Attach Profiler. The Attach Profiler task dialog comes up. SC: "One last feature! While all this integration is nice, the profiler also supports just attaching to any JVM. It will even show you the JVM command-line flags that are needed and will edit a startup script for you if you want it to."
2. Click the Attach button. Make selections and start going through the steps of the wizard. SC: "I can specify my application type, and server if I am using one, JDK version, etc. and the IDE will offer to edit my application's startup script or show me the changes that need to be made."
3. Click the Cancel button. "If I had actually started the JVM for my application the JVM would see the command line flags and wait before proceeding. This is the same way that debuggers attach to a JVM, by the way. Once the JVM is waiting I can click the Attach button and start profiling."
4. Click the Cancel button.

## **Cleanup**

- (Optional) Undeploy and delete the Tomcat servlet application

## Module 4 Demo 2: Application Profiling Using Sun Studio

This exercise introduces application profiling using the Sun Studio Collector Analyzer.

Slide id: 134

### Required Software

- Solaris OS (Sparc or X86)
- JDK 1.5 or higher
- Sun Studio version 11 or 12

You can download Sun Studio from <http://developers.sun.com/sunstudio/downloads/>.

- Demo software

The following software is located in the demo/mod04/sw directory.

- BatchProcessor.java
- ReadThread.java
- WorkQueue.java
- WriteThread.java

### Objective

- Use Sun Studio Collector and Analyzer tools to profile Java technology applications.

### Preparations

Check for required version of Java SE.

1. Use the `java -version` command to check the installed Java SE version.

If the installed version is earlier than Java SE 5, download and install the latest Java SE version from the following URL.

<http://java.sun.com/javase/downloads/index.jsp>

Download and install Sun Studio.

2. Download and and install Sun Studio from the following URL:

<http://developers.sun.com/sunstudio/downloads/>

3. Add the bin subdirectory of the Sun Studio install directory to the Solaris OS PATH variable. For example, if you installed Sun Studio in the default location `/opt/SUNWspro` you can enter the following commands at a terminal window.

```
PATH=/opt/SUNWspro/bin:$PATH
export PATH
```

### Instructions

1. Open a command line window.  
`cd <course_install_dir>/demo/mod04 directory`
2. Use the `javac` command to compile the following Java source files.



```
javac BatchProcessor.java
javac ReadThread.java
javac ReadThread.java
javac WriteThread.java
```

3. Run the `BatchProcessor` program to ensure it runs in your environment.

```
Java BatchProcessor
```

This program will run for 90 seconds and print some statistics. What it prints is unimportant. The objective is to test that it runs.

4. Run the Sun Studio Collector utility using the following command line.

```
collect -j on java BatchProcessor
```

This invokes the Sun Studio Collector to capture a running profile of the Java program. It creates a directory in the current directory called `test.er.1`. If the file is not created check that you have write permissions to the directory from which you ran the program.

The default sampling interval is 1 second and the default output directory is the present working directory. Please refer to the Sun Studio Collector documentation for command line options that override default values.

5. When the Collector completes, run the Sun Studio Analyzer program to read the results:

```
analyzer test.er.1
```

6. Where the `test.er.1` is the output file created by the Collector. If you see a warning dialog The Analyzer might produce a warning dialog when it loads the profile data file. This is expected in this demo and can be ignored.

By default Analyzer displays *exclusive* user CPU consumption in descending order. *Exclusive* here means only the method body and no methods called by the method. *Inclusive* means the method body and any methods it calls and methods calls by those methods and so on.

Unlike traditional Java profilers, Sun Studio Collector / Analyzer also capture JVM information in the profile. This is a key differentiator for Collector / Analyzer.

7. You can optionally add additional columns by selecting the `View > Set Data Presentation` menu item.

Of the options available on the `Metrics` tab, the most useful tend to be: `System CPU`, `User CPU` and `User Lock`. Also notice that you can display overall % of time. `System CPU` data is useful for tracking down methods consuming a lot of system or kernel CPU. If high system or kernel CPU is not a concern for an application under profile, then there's probably not much to be gained from looking at that data. The same is true for `User Lock` data.

8. Explore the `Formats` tab by clicking on `Formats` tab in the `Set Data Presentation` dialog.

The most important part of this display is the `View Mode`. By default, the `View Mode` is `User`. `User` mode shows Java call stacks for Java threads, and does not show housekeeping threads. `Expert` mode shows Java call stacks for Java threads when the user's Java code is being executed, and machine call stacks when JVM code is being executed or when the JVM software does not report a Java call stack. It shows machine call stacks for housekeeping threads. The `Expert Mode` is very useful for Java code since it will show the generated assembly code for Java methods. In addition, it can also show which assembly instruction is

the highest consumer of CPU cycles for a given method. In Expert Mode and User Mode, Java byte code is viewable for Java methods. Likewise, Java byte codes for a method which are the most expensive can be viewed too. (We'll come back to this a little later).

Close the `Set Data Presentation` dialog.

9. Explore the use the `Filter` dialog to focus on the data of interest.

The Analyzer can also filter data passed on the samples it took. For instance, if you are only interested in the middle portion of the executing program, perhaps it had a warm up phase, or an initialization phase and you are not interested in that data, you can filter out that data by selecting the `View > Filter Data` menu item. On the `Filter Data` menu item, you can provide a set of samples to include in the displayed profile. So, if the application ran for 90 seconds and it collected 1 sample per second and you are only interested in results after an initialization phase of 15 seconds up to 5 seconds before shutdown, you can tell Analyzer to include a range of samples from 16 – 75. Likewise, if you knew there were thread ids you were not interested in, you could exclude them too. Similarly for LWPs and CPUs. However, filtering on a range of samples is probably the most valuable use of this feature.

Close the `Filter` dialog.

10. Explore the method callee display capability of the Analyzer.

On the main dialog, select the `Callers – Callees` tab to view who calls a given selected method.

11. Explore the source code display capability of the Analyzer.

To view the source code, the *search path* must be configured in the `View > Set Data Presentation` dialog under the `Search Path` tab.

12. Explore the byte code display capability of the Analyzer.

Click on the `Disassembly` tab, you can see the Java byte code for Java method selected. For each Java byte code it will show how much time is spent on a Java byte code. The most expensive will be colored green.

13. Check the `Functions` tab for the following entries:

- `JVM-System`

`JVM-System` represents internal JVM methods

7. `Unknown`

`Unknown` represents methods the Collector/Analyzer is not able to determine. They are most likely JVM related.

14. Observe the change to the display that results from adding `System CPU` as a metric and ordering it in *exclusive* descending order.

After viewing the change remove `System CPU` results since this application has very little `System CPU` activity.

15. View the profile in `Expert Mode`.

Be sure to press `Apply` after switching modes on the `View > Set Data Presentation Formats` tab. Notice that Analyzer has resolved the information from `JVM-System` and `Unknown` entries of the `Functions` tab. You should probably see method names like; `Block::is_Empty` and `IndexSetIterator::next`, `Interpreter`, and so on. These are internal JVM methods.

16. Select `ReadThread.doWorkOnItem` method and click on the `Disassembly` tab, if the source for `ReadThread` class is not available, it will display the Java byte code for the `ReadThread.doWorkOnItem` method. It will also show which Java byte code is consuming the most time.
17. Select a Java method name such as `java.lang.Thread.sleep` where the Java source is available and click on the `Disassembly` tab. Find the class declaration for `java.lang.Thread` and a green bar to the right of the scroll bar. If you click on the green bar, it will take you to the line of code in `java.lang.Thread` consuming the most time. You will also see the Java byte code associated with time consumer.
18. Select an internal JVM method, such as `IndexSetIterator::next`, and click on the `Disassembly` tab, and you will see the assembly code for this method. Again, the green bars to the right of the scroll bar will highlight which assembly instructions are the most time consuming.
19. Next View the profile in Machine Mode, In this mode any Java method selected will show the JIT generated assembly code for the Java method. (Be sure to press `Apply` after switching modes on the `View > Set Data Presentation Formats` tab). You might not see the assembly on the first try of a Java method, you might have to switch back and forth amongst a couple Java methods before it will appear. This could be a lingering bug in Analyzer. Select the `ReadThread.doWorkOnItem` method and click the `Disassembly` tab. You should see the assembly code for this method. This is the assembly code generated by the JVM JIT compiler. Again, the green bars to the right of the scroll indicate the assembly instructions where the most time is spent.
20. Switch back to User Mode now and set the data presentation to include User Lock. Remember to press `Apply` button. Sort by in exclusive User Lock in descending order.  
  
You will probably notice that `java.lang.Object.wait` method is at the top. If you select it and press the `Callers - Callees` tab, you will see who calls the `java.lang.Object.wait` method. If you continue to click on the upper portion of the `Callers - Callees` tab, you will eventually find the `java.lang.Object.wait` method results from a `WorkQueue.dequeue` method which in turn is called by `ReadThread.getWorkItemFromQueue` method. This method contains an `Object.wait` that waits to be notified when there is work available on the queue. You could conclude from this profile that it does not having a locking contention problem.
21. You can further explore what's in the JVM-System area. To explore this area, you need to switch to Expert Mode. After switching modes, you will see `__lwp_cond_wait` and `__lwp_park` show up. If you look at the `Callers - Callees` display, you should see that these are related to the JVM's garbage collector and JVM's JIT compiler.

## Module 4 Demo 3: Heap Memory Profiling

This exercise demonstrates the use of the the JDK bundled jmap and jhat profiling tools.

Slide id: 136

### Required Software

- Any OS
- JDK 1.5 or higher
- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objectives

- Use the jmap utility to create a heap dump.
- Use the jstat and VisualVM utilities to analyze the heap dump.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Down load and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the Java2D demo directory:

```
cd <JDK install dir>/demo/jfc/Java2D
```

2. Run the Java2Demo with the following command line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=3m -XX:PermSize=1m  
-jar Java2Demo.jar
```

3. Click the transforms tab of the Java2Demo GUI.
4. Obtain the lvmid of the Java2Demo process using the following command on another command window.

```
jps -l
```

You should observe an output (that maps lvmids to executing Java processes) similar to the following:

```
2564
```

2874 Java2Demo.jar  
1948 sun.tools.jps.Jps

Note: The *lvmid* is a platform-specific value that uniquely identifies a JVM on a system. The *lvmid* is the only required component of a virtual machine identifier. The *lvmid* is typically, but not necessarily, the operating system's process identifier for the target JVM process.

5. Run `jmap` in a command line window to dump a binary heap profile of the Java2Demo:

```
jmap -dump:format=b, file=<location of heap dump file>/heapdump.hprof <lvmid>
```

Note: This command creates a binary heap dump from the Java2Demo. The `.hprof` extension enables viewing of the heap dump using VisualVM (performed later in this demo).

6. Use `jhat` utility to examine the heap dump created in the previous step.

```
jhat <location of heap dump file>/heapdump.hprof
```

The `jhat` command will launch a web server at the URL: <http://localhost:7000>.

7. Use a browser to connect to the URL at: <http://localhost:7000>.

The browser should display the heap profile ordered by Java packages.

8. Scroll to the bottom of the web page to view general operations you can perform on the heap dump. Of particular interest is the `Show heap histogram` operation. This operation is very useful for identifying biggest allocator of memory.

Another useful link is the `Object Query Language` link. This powerful capability of `jhat` supports queries against the binary heap. It provides the ability to ask specific questions about the captured heap dump.

9. Close the web browser.

10. Quit `jhat` by executing a `^C` in the window where it was started.

11. Start Java VisualVM or open source VisualVM

12. Inside VisualVM, use the `File > Load` menu item and filter by `Heap Dumps` to find the `heapdump.hprof` file.

When the heap dump loads, you will see a Summary screen which provides some high level information.

13. Use the `Classes` button to display a heap profile view per class in a traditional heap profiler manner.

Right click any class and select the `Show Instances View` menu item. The `Show Instances View` allows you to walk the heap to see who has references to the selected object.

14. Close the VisualVM utility.

## Module 4 Demo 4: Application Profiling Using Sun Studio

This exercise demonstrates the use of the complementary profiling capabilities of the NetBeans profiler and the Sun Studio Collector Analyzer.

Slide id: 143

### Required Software

- Solaris OS (Sparc or X86)
- Java 6 JDK or higher
- Sun Studio version 11 or 12

You can download Sun Studio from <http://developers.sun.com/sunstudio/downloads/>.

- Java2Demo

Located in: <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar

### Objective

- Use the NetBeans profiler and the Sun Studio Collector and Analyzer tools to profile Java technology applications.

### Preparations

Check for required version of Java SE.

1. Use the `java -version` command to check the installed Java SE version.

If the installed version is earlier than Java SE 6, download and install the latest Java SE version from the following URL.

<http://java.sun.com/javase/downloads/index.jsp>

Download and install Sun Studio.

2. Download and and install Sun Studio from the following URL:  
<http://developers.sun.com/sunstudio/downloads/>
3. Add the bin subdirectory of the Sun Studio install directory to the Solaris OS PATH variable. For example, if you installed Sun Studio in the default location `/opt/SUNWspro` you can enter the following commands at a terminal window.

```
PATH=/opt/SUNWspro/bin:$PATH
export PATH
```

Download and install NetBeans 6.1.

1. Download and and install NetBeans from the following URL:  
<http://www.netbeans.org>
2. Add the bin subdirectory of the NetBeans 6.1 install directory to the OS PATH variable.

### Instructions

Profiling using the NetBeans profiler.

1. Start the NetBeans IDE using the NetBeans desktop icon by entering the following command

in a command line window.

```
netbeans
```

2. In NetBeans IDE, select **Profile > Attach Profiler** menu option. Click on the CPU icon on the left. Select to profile **Entire Application** on the right. Leave the default filter as **Profile all classes** and click on **Attach**. Select **Application** as the target type of profiling. Select **Attach Method of local** and **Attach invocation of Dynamic**. Finally, select **Java SE 6.0** as the Java platform to run your application to complete the wizard interaction.

3. When the **Select Process** dialog is displayed in NetBeans IDE, use a command line window and launch the **Java2Demo** application with the following command line options.

```
java -server -Xmx15m -Xms15m -Xmn3m -XX:+UseSerialGC  
-jar <JDK install dir>/demo/jfc/Java2D/Java2Demo
```

4. On NetBeans IDE, press the **Refresh** button in the **Select Process** dialog and press **OK**.

5. NetBeans will instrument the running **Java2Demo**.

6. Click on the **Live Results** icon in NetBeans IDE in the left panel **Profiling Results**. When NetBeans has finished instrumenting the **Java2Demo** application, it will start displaying live profiling results.

7. Now click on the **Transforms** tab of the **Java2Demo** application.

8. You can now observe the Java level methods which are consuming the most time in the application in NetBeans IDE.

The NetBeans Profiler should show that approximately 45% of its time is spent in `java2d.AnimatingSurface.run` method.

### Profiling using the Sun Studio Collector and Analyzer tools

1. Open a command line window and change to `exercises/mod04/working` directory  

```
cd exercises/mod04/working
```

2. Run **Java2Demo** using the following command line.

```
collect -j on java -server -Xmx15m -Xms15m -Xmn3m -  
XX:+UseSerialGC -jar <JDK install dir>/demo/jfc/Java2D/Java2Demo
```

This invokes the Sun Studio Collector to capture a running profile of the Java program. It creates a file in the current directory called `test.er.1`.

The default sampling interval is 1 second and the default output directory is the present working directory. Please refer to the Sun Studio Collector documentation for command line options that override default values.

3. After the **Java2Demo** is launched, click on the **Transforms** tab and leave the **Java2Demo** run for about 15 or 20 seconds. Then, quit the **Java2Demo** program.
4. Now use the Sun Studio Analyzer to open the `test.er.1` experiment just collected.

```
analyzer test.er.1
```

## Analysis

This section compares and contrasts what was seen in Collector / Analyzer with what is seen in NetBeans Profiler.

The NetBeans Profiler showed that approximately 45% of its time was spent in `java2d.AnimatingSurface.run` method. However, it is very hard to find that method in the Collector / Analyzer profile.

If you examine the source code for `java2d.AnimatingSource.run` method, (viewable at `<jdk install dir>/demo/jfc/Java2D/src/java2d/AnimatingSurface.java`) you will see that there is not much to the `run` method. That is because much of what was seen in Collector / Analyzer `<JVM-System>` is what is executing **within the body of** `java2d.AnimatingSource.run` method.

This example illustrates how the two profilers can compliment each other. The NetBeans Profiler shows much of the time is spent in `java2d.AnimatingSource.run` method which might not say much about what is really going on inside of that method. You need to look at, or re-look at the Collector / Analyzer profile to get a better idea of what's happening within `java2d.AnimatingSource.run` method.

If you considered the task under way was to identify performance tuning opportunities with the Java2Demo, the view in Sun Studio Analyzer does not provide much help if you want to focus on Java level optimization opportunities. It appears that in the default view the majority of the time is spent within `<JVM-System>`. Changing the User Mode in the Set Data Presentation dialog to Expert or Machine shows the JVM methods where the time is being spent. This is great for JVM developers. But, not much benefit for a developer wanting to improve the performance of Java2Demo at the Java code level.

NOTE: For server side applications, especially those that run an extended period of time, using Collector / Analyzer works very well at identifying performance optimization opportunities. It is these same class of applications that are more difficult to setup, configure and profile with traditional Java profilers such as JProbe and OptimizeIt or with NetBeans Profiler.



## Module 4 Demo 5: Perform Memory Leak Profiling

This exercise performs memory leak profiling using the NetBeans Profiler.

Slide id: 154

### Required Software

- Any OS
- JDK 1.6 or higher
- NetBeans version 6.1 or later.

You can download NetBeans from: <http://www.netbeans.org>

- NetBeans projects
  - HTTPUnit

### Objective

- Use NetBeans profiler to perform memory leak profiling.

### Preparations

Check for required version of Java SE.

1. Use the `java -version` command to check the installed Java SE version.

If the installed version is earlier than Java SE 6, download and install the latest Java SE version from the following URL.

<http://java.sun.com/javase/downloads/index.jsp>

Download and install NetBeans 6.1.

2. Download and and install NetBeans from the following URL:

<http://www.netbeans.org>

3. Add the `bin` subdirectory of the NetBeans 6.1 install directory to the OS PATH variable.

# Instructions

## Background

This is based on an actual usage of the NetBeans Profiler as part of the development and testing of a production application. Andrés González of Spain used the NetBeans Profiler to track down a memory leak in HttpUnit. HttpUnit is an open source project that provides a framework for unit testing of web pages. It essentially acts as a web browser so that you can write unit tests to verify that the correct pages are being sent back from your web application.

Andrés was using HttpUnit to run multi-day tests of his application (so it sort of sounds like he was using it as a convenient way to do long-term stress testing). He discovered something sort of odd - the JVM that was running the tests he created with HttpUnit would eventually report an `OutOfMemoryError`. The tests had to run for long periods of time before the `OutOfMemoryError` would occur though, so apparently each memory leak was relatively small.

Andrés used the NetBeans profiler to track down the problem and he wrote a [blog entry](#) about it (the entry is in Spanish, but Google can translate :-). There is also [this thread](#) out on the HttpUnit mailing lists where he reported what he found.

The sample application included here is not the program that Andrés wrote - it does however encounter the same problem in HttpUnit. The underlying issue has to do with the way that HttpUnit processes web pages that include JavaScript. The framework does have support for a subset of JavaScript, but not the entire language. If it encounters JavaScript that it does not understand it will throw an exception. If the web page you are attempting to test includes JavaScript that HttpUnit does not support and you do not want to wade through all those exceptions in the output then the HttpUnit documentation recommends that your test program call

```
HttpUnitOptions.setExceptionsThrownOnScriptError( false );
```

The side effect, however, is that HttpUnit will store every exception thrown during its JavaScript processing in an `ArrayList` and it *never* removes them. So if your tests access enough web pages that either have JavaScript errors or that include JavaScript that HttpUnit does not support, you can eventually get an `OutOfMemoryError`.

One additional note on the sample application - it does not require a web server in order to run. HttpUnit has a nice feature where if what you want to test is the response from a servlet then it can host the servlet for you, in the same JVM as your test application. So the sample application consists of:

1. a servlet that returns JavaScript that has an error
2. a `main()` method that repeatedly requests a page from that servlet.

## Main Points to Hit

1. The NetBeans Profiler has a powerful capability to help you track down memory leaks.
2. Using instrumentation you can watch allocations on the heap happen in real time.
3. The profiler provides statistical values you can use to watch for patterns in your application's memory allocations. This "behavioral" approach can help you quickly identify the most likely memory leak candidates, even in situations such as this one where each individual leak is very small.
4. With its tight integration into the developer work flow, it is easy to start/stop profiling sessions and more importantly, to go from profiler results directly into the source code that has the problem.

## Setup

- NetBeans IDE 6.1 with Java EE support or higher
- JDK 5 update 8 or higher
- Open the HttpUnit sample project, which is available in demos/mod04/sw folder.
- To avoid delay with the demo, profile the application at least once so that the IDE can modify the build.xml file

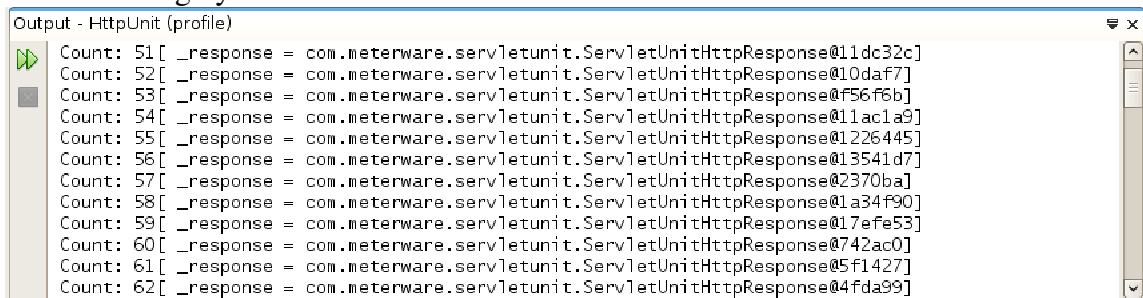
## Gotchas

- This demo does not work on OS X ([Failed to obtain snapshot. The profiled application terminated.](#)). Update: This appears to be working now that I'm using the new [Java SE 6 Developer Preview 9](#) from Apple.

This problem has also been known to occur on Windows. A possible workaround is to set -Xnoclassgc in the project properties of the HttpUnit project. Note, however, that using -Xnoclassgc skews the results a bit. The memory leak is still the same, but the String allocations done by java.lang.String.substring() will also be climbing in Generation count because classes are not being unloaded (the Rhino JavaScript engine apparently makes heavy use of dynamically generated classes to interpret JavaScript).

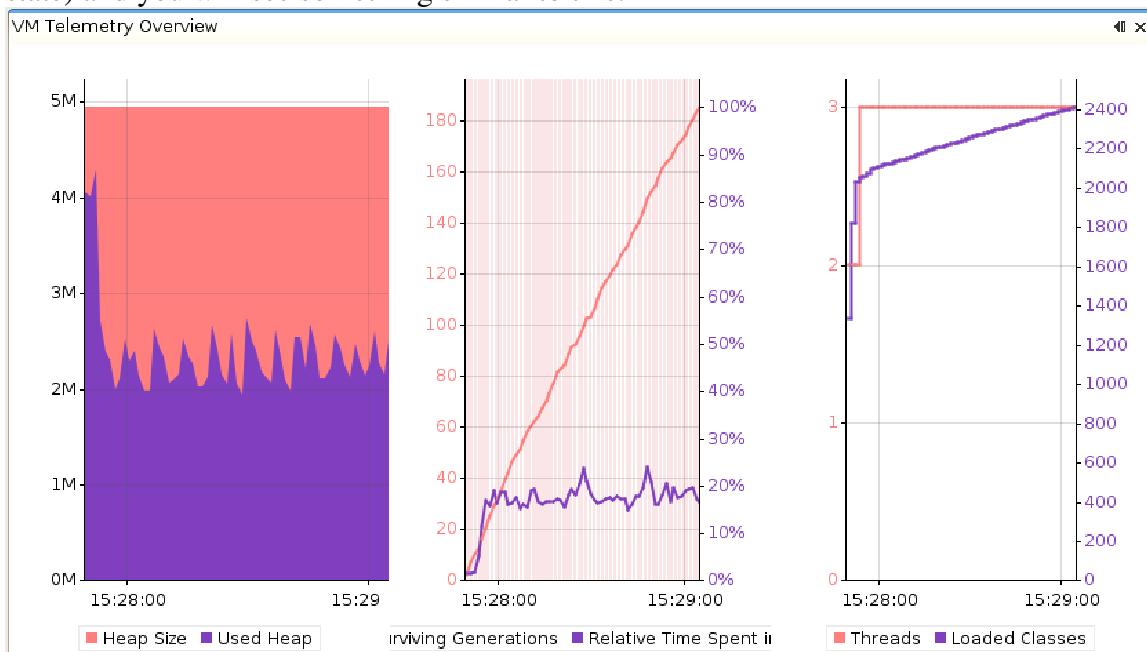
## Instructions

1. Open the HttpUnit project
2. Right-click the HttpUnit project and choose Profile
3. Select Memory from the list of tasks on the left of the dialog.
4. Select Record both object creation and garbage collection
5. Select 10 for the value in Track every object allocations
6. Check the Record stack trace for allocations option
7. Make sure the Use defined Profiling Points option is **NOT** checked
8. Click the Run button
9. The application begins running. In the Output window of the IDE you will see lines such as these scrolling by:



```
Output - HttpUnit (profile)
Count: 51[ _response = com.meterware.servletunit.ServletUnitHttpResponse@11dc32c]
Count: 52[ _response = com.meterware.servletunit.ServletUnitHttpResponse@10daf7]
Count: 53[ _response = com.meterware.servletunit.ServletUnitHttpResponse@f56f6b]
Count: 54[ _response = com.meterware.servletunit.ServletUnitHttpResponse@11ac1a9]
Count: 55[ _response = com.meterware.servletunit.ServletUnitHttpResponse@1226445]
Count: 56[ _response = com.meterware.servletunit.ServletUnitHttpResponse@13541d7]
Count: 57[ _response = com.meterware.servletunit.ServletUnitHttpResponse@2370ba]
Count: 58[ _response = com.meterware.servletunit.ServletUnitHttpResponse@1a34f90]
Count: 59[ _response = com.meterware.servletunit.ServletUnitHttpResponse@17efe53]
Count: 60[ _response = com.meterware.servletunit.ServletUnitHttpResponse@742ac0]
Count: 61[ _response = com.meterware.servletunit.ServletUnitHttpResponse@5f1427]
Count: 62[ _response = com.meterware.servletunit.ServletUnitHttpResponse@4fda99]
```

10. Suggested Comment (SC): "This is a simple test application that emulates the behavior that the developer of this application saw. It uses HttpUnit to process the HTML that is returned by a servlet. Requests are being repeatedly sent to that servlet by the test."
11. Click the Telemetry Overview icon in the Profile Window under the Controls panel heading (the icon looks like a graph). This will open the Telemetry Overview window at the bottom of the NetBeans IDE. Maximize it (a double click on the VM Telemetry Overview window title bar will maximize it, a double click on it again will restore it back to its original state) and you will see something similar to this:



12. SC: "The purple graph on the left shows heap usage. The value is trending upward over time. But very slowly." **NOTE:** It takes over 30 seconds for this pattern to emerge, so skip this step if you are pressed for time or think the audience is getting restless. :-)
13. Un-maximize the Telemetry Overview so that you can see the Profile window.
14. Click the Live Results icon in the Profile window.
15. SC: "This Live Results window shows activity on the heap. The column on the left

contains class names. For each class you can see information about the number of objects created, the number that are still in use (live), and a particularly interesting statistic called Generations."

16. Click the Generations column to sort the display by Generations. This will sort it in descending order and will look like this:

Welcome x

Main.java x

Live Profiling Results x

Memory: 02:58:42 PM \* x

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Allocated ...	Avg. Age	Generations
java.lang.String	<div></div>	4,320 B (5.5%)	180 (16.3%)	23,803	134.4	40
char[]	<div></div>	31,216... (39.4%)	244 (22.2%)	38,264	120.1	31
java.lang.Object[]	<div></div>	7,048 B (8.9%)	31 (2.8%)	3,552	151.2	9
java.lang.Class[]	<div></div>	1,480 B (1.9%)	84 (7.6%)	10,064	202.0	9
java.lang.reflect.Constructor	<div></div>	1,024 B (1.3%)	16 (1.5%)	853	287.5	9
java.util.Hashtable\$Entry	<div></div>	648 B (0.8%)	27 (2.5%)	4,116	174.4	7
java.lang.ref.SoftReference	<div></div>	352 B (0.4%)	11 (1%)	128	359.0	7
java.util.Vector	<div></div>	312 B (0.4%)	13 (1.2%)	586	270.0	7
java.lang.reflect.Method	<div></div>	4,320 B (5.5%)	54 (4.9%)	6,177	201.7	6
java.util.Hashtable\$Entry[]	<div></div>	696 B (0.9%)	9 (0.8%)	962	239.9	6
java.util.HashMap\$Entry	<div></div>	624 B (0.8%)	26 (2.4%)	183	346.2	6
java.util.Hashtable	<div></div>	320 B (0.4%)	8 (0.7%)	619	270.0	6
byte[]	<div></div>	14,248... (18%)	30 (2.7%)	8,566	60.4	5
java.util.HashMap\$Entry[]	<div></div>	720 B (0.9%)	9 (0.8%)	363	272.6	5

[Class Name Filter]

Live Results

Class History

17. SC: "Two of the classes have huge values for Generation, in comparison to all the other classes: String and char array. More importantly, the Generations value for both of them continues to increase as the application runs. Note how for the rest of the classes this is not the case - they have stabilized."
18. SC: "The generation count for a class is easy to calculate. All you have to understand is that each object has an age. The age of an object is simply the number of the Java virtual machine's garbage collections it has survived. So if for example an object is created at the beginning of an application and the garbage collector has run 466 times then the age of that object at that point in time is 466. To calculate the Generation value for a class, just count up the number of different ages across **all** of its objects that are currently on the heap. That count of different ages is the number of generations."
19. SC: "Please note: there is no 'correct' value for generation count. The key thing to watch out for is classes that have generation counts that are **always** increasing. If the generation count is always increasing then that means objects of that class are being created repeatedly as the program runs and more importantly, *not all existing object instances* are being garbage collected. As a result, as time goes on and the garbage collector continues to run you get more and more objects created at different points in time and therefore with different ages."
20. SC: "The increasing generation count for the String class (and its little friend char array :-)) indicates Strings are being created repeatedly."
21. Right-click the entry for String and select Take Snapshot and Show Allocation Stack Traces:

Method Name - Allocation Call Tree	Live Bytes...	Live Bytes	Live Objects	Allocated...	Avg. Age	Generat...
java.lang.String	3,...	3,...	144 (100%)	12,319	190.7	46
java.lang.StringBuffer.toString ()	1,...	1,...	54 (37.5%)	2,419	141.9	39
java.lang.String.substring (int, int)	64...	64...	27 (18.8%)	6,732	59.6	5
Objects allocated by reflection	31...	31...	13 (9%)	20	398.4	3
java.lang.StringBuilder.toString ()	72...	72...	3 (2.1%)	452	131.3	2
java.util.Properties.loadConvert (ch...	76...	76...	32 (22.2%)	32	394.3	2
org.apache.xerces.xni.XMLString.toS	96...	96...	4 (2.8%)	353	0.8	2
java.lang.String.toLowerCase (java.	48...	48...	2 (1.4%)	210	0.0	1
java.lang.Integer.toString (int)	24...	24...	1 (0.7%)	39	0.0	1
com.meterware.httpunit.WebRespons	48...	48...	2 (1.4%)	1,059	0.0	1
java.lang.String.toUpperCase (java	48...	48...	2 (1.4%)	163	0.0	1
org.cyberneko.html.HTMLScanner.sc	24...	24...	1 (0.7%)	239	0.0	1
com.meterware.httpunit.WebRespons	48...	48...	2 (1.4%)	561	0.0	1
org.mozilla.javascript.TokenStream.c	24...	24...	1 (0.7%)	40	0.0	1

22. Select Profile > Stop Profiling Session. SC: "We have the information we need, so we can stop the profiling session."
23. SC: "The allocation stack traces view shows all the places in the code where Strings were allocated. In a typical Java application there can be dozens or even hundreds or thousands of places where Strings are allocated. What you want to know is: which of those allocations are resulting in memory leaks? You can use the generation count as a key indicator. Notice that only one of the allocation locations in this group has created Strings that have a large value for Generation count: `java.lang.StringBuilder.toString()`. If we were to continue running the application and take more snapshots we would see larger values each time."
24. SC: "So we know that `StringBuilder's toString()` is allocating strings that appear to be candidate memory leaks. So what? How do we tie that back to our application's usage of `HttpUnit`?"
25. Click the icon next to the entry for `StringBuilder.toString()` to expand it:

Method Name - Allocation Call Tree	Liv...	Liv...	Liv...	All...	Avg...	...
java.lang.String	0.0%	100%	12,...	190.7	46	
java.lang.StringBuffer.toString ()	7.5%	17.5%	2,4...	141.9	39	
com.meterware.httpunit.javascript.JavaScript\$JavaScriptEngine.handleScriptException	0.6%	10.6%	44	174.2	38	
org.mozilla.javascript.ScriptableObject.defineClass (org.mozilla.javascript.Scriptable, Cl	2.8%	2.8%	1,8...	0.0	1	
com.meterware.httpunit.URLEncodedString.getString ()	2.1%	2.1%	349	0.0	1	
org.mozilla.classfile.ClassFileWriter.add (byte, String, String, String, String)	0.7%	0.7%	42	0.0	1	
org.mozilla.classfile.ConstantPool.addMethodRef (String, String, String)	0.7%	0.7%	54	0.0	1	
org.mozilla.classfile.ConstantPool.addFieldRef (String, String, String)	0.7%	0.7%	44	0.0	1	
java.lang.String.substring (int, int)	8.8%	8.8%	6,7...	59.6	5	
Objects allocated by reflection	9%	9%	20	398.4	3	
java.lang.StringBuilder.toString ()	2.1%	2.1%	452	131.3	2	
java.util.Properties.loadConvert (char[], int, int, char[])	2.2%	2.2%	32	394.3	2	
org.apache.xerces.xni.XMLString.toString ()	2.8%	2.8%	353	0.8	2	
java.lang.String.toLowerCase (java.util.Locale)	1.4%	1.4%	210	0.0	1	
java.lang.Integer.toString (int)	0.7%	0.7%	39	0.0	1	

26. SC: "Ah ha! :-)) The only strings allocated in `StringBuilder.toString()` with a large value for generation count are the Strings that resulted from calls to `StringBuilder.toString()` by a call from the `HttpUnit` code: `com.meterware.httpunit.javascript.JavaScript$JavaScriptEngine.h`

andleScriptException()."

27. Expand the entry for handleScriptException() and since there is only one way it is being called the profiler goes ahead and expands the entire stack trace. You will see a straight-line back to the main() method in the test application:

Method Name - Allocation Call Tree	Liv...	Liv...	Liv...	All...	Avg...	...
com.meterware.httpunit.parsing.NekoHTMLParser.parse	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.HTMLPage.parse (String, jav	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebResponse.getResponse	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebResponse.getFrame	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebResponse.getFrai	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.FrameHolder.updat	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebWindow.upda	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebClient.upda	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebWindow.u	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebWindow	0.6%	0.6%	44	174.2	38	
com.meterware.httpunit.WebCli	0.6%	0.6%	44	174.2	38	
Main.main (String[])	0.6%	0.6%	44	174.2	38	
org.mozilla.javascript.ScriptableObject.defineClass (org.mozilla.javascript.Scriptable, Class, boolean)	2.8%	1.8...	0.0	1		
com.meterware.httpunit.URLEncodedString.getString ()	2.1%	2.1%	349	0.0	1	
org.mozilla.classfile.ClassFileWriter.add (byte, String, String, String)	0.7%	0.7%	42	0.0	1	
org.mozilla.classfile.ConstantPool.addMethodRef (String, String, String)	0.7%	0.7%	54	0.0	1	

28. Right-click Main.main (for it to be visible you will have to increase the width of the Method Name column) and choose Go To Source. SC: "One of the advantages of an integrated profiler - easy access to the source code."

29. SC: "Here we are in the sample application calling HttpUnit's getResponse () method on line 46, which ends up making the call that results in the memory leak:

```

27  /*
28  public class Main {
29
30      /** Creates a new instance of Main */
31      public Main() {
32      }
33
34      /**
35       * @param args the command line arguments
36       */
37      public static void main(String[] args) {
38          try {
39              HttpUnitOptions.setExceptionsThrownOnScriptError(false);
40              ServletRunner sr = new ServletRunner();
41              sr.registerServlet("myServlet", HelloWorld.class.getName());
42              ServletUnitClient sc = sr.newClient();
43              int number = 1;
44              WebRequest request = new GetMethodWebRequest("http://test.meterware.com/myServlet");
45              while (true) {
46                  WebResponse response = sc.getResponse(request);
47                  System.out.println("Count: " + number++ + response);
48                  java.lang.Thread.sleep(200);
49              }
50          } catch (InterruptedException ex) {
51              Logger.getLogger("global").log(Level.SEVERE, null, ex);
52          } catch (MalformedURLException ex) {
53              Logger.getLogger("global").log(Level.SEVERE, null, ex);
54          }
55      }
56  }

```

30. HttpUnit has a memory leak in its call to ServletUnitClient.getResponse(WebRequest request). But, what is the source of the memory leak?

31. Go back to the top of stack traces and you will find JavaScript.handleScriptException() is the method that contains the memory leak. Traverse to the JavaScript.handleScriptException() method and right click to "Go to Source".

32. Examine the method and notice that a String is allocated with, final String errorMessage = badScript + " failed: " + e; on line 196. But, that's not the memory leak. The memory occurs at line 204, \_errorMessages.add( errorMessage ); If you select

`_errorMessages`, hold down the <control> key you will see a tool tip that says `_errorMessages` is an `ArrayList`, alternatively you can see it in the Navigator window on the left.

33. If you select `_errorMessages`, right click and select Find Usages, you find any where `_errorMessages` is used in the application. By doing so you see that there are calls to add elements, `clear()` elements and to return an array of elements. Double click on the `_errorMessages.clear()` to see where it is called. `_errorMessages.clear()` is called from `JavaScript.clearErrorMessages()`. Now, do a Find Usages on the `JavaScript.clearErrorMessages()` method. It is called only one place by `JavaScriptEngineFactory.clearErrorMessages()`. Do a Find Usages on `JavaScriptEngineFactory.clearErrorMessages()`. This is called by `HttpUnitOptions.clearScriptErrorMessages()`. Do a Find Usages `HttpUnitOptions.clearScriptErrorMessages()`. Ahh ... haa, nobody calls `HttpUnitOptions.clearScriptErrorMessages()`.
34. We just found the root cause of the memory leak. The `_errorMessages ArrayList` is never cleared. It only has elements added to it. In other words, `HttpUnit` stores \*all\* `JavaScript Exceptions` in this `ArrayList`, but never clears the `ArrayList`. So, to fix this memory leak, the `ArrayList` will need to be explicitly cleared. This is a task for the authors of `HttpUnit`.

## Cleanup

None.



## Module 4 Demo 6: Using jhat to Detect Memory Leaks

This exercise demonstrates the use of the the JDK bundled jmap and jhat to detect memory leaks.

Slide id: 156

### Required Software

- Any OS
- JDK 1.6 or higher
- Java VisualVM (included in Java SE 6 update 7 or later) or open source VisualVM
- Optional exercise uses NetBeans Profiler
- `PrimeNumbers.jar`

Located in: `demos/mod04/sw/`

### Objectives

- Use `-XX:+HeapDumpOnOutOfMemoryError` with `jmap` and `jhat` to find memory leaks.
- Use VisualVM to enable `HeapDumpOnOutOfMemoryError` on Java SE version 6 or later applications.
- Use any one of `jhat`, VisualVM or NetBeans Profiler to detect memory leaks or do heap walking.

### Preparations

Verify and if required download and install Java SE 6 or later.

1. Verify if you are running Java SE 6 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: `java version 1.6.0_00` or later you can bypass the next step and proceed to the Instructions subsection.

2. Down load and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions: Primary Exercise

1. Open a command line window and change directory to the `PrimeNumbers.jar` directory:

```
cd demos/mod04/sw/
```

2. Run the `PrimeNumbers` program with the following JVM arguments:

```
java -Xmx6m -XX:+HeapDumpOnOutOfMemoryError -jar PrimeNumbers.jar
```

You should observe the launch of a UI display.

3. Enter 1000000 (that is one million) into Enter a number field of the `PrimeNumbers` UI and click the `Calculate Prime Numbers` button.

The `PrimeNumbers` program will calculate the largest prime number less than or equal to that number of 1000000 (one million).

4. The result of 999983 should display fairly quickly. Click the button again with 1000000, and the result should display even faster as this program caches the answers it previously calculates.
5. Click the `Calculate Prime Numbers` button again. The application should hang or dies with an `java.lang.OutOfMemoryError: Java heap space`, visible in the window where you launched the `PrimeNumbers` program.
6. Kill the `PrimeNumbers` program if it did not die.
7. Use window from where you launched `PrimeNumbers` program to list the directory (`ls` for Solaris OS, `dir` for Windos OS). Look for a file with a the `.hprof` extension. This is the heap dump that was generated by JVM when it encountered an `OutOfMemoryError`. Note the name of the file.
8. Launch `jhat` using the name of the `.hprof` file as an argument. For example,  

```
jhat java_pid2944.hprof
```
9. When `jhat` reports the server is ready, use a browser to connect to the URL:  
<http://localhost:7000>
10. In the browser, scroll to the bottom of the screen and select `Show Heap Histogram`.

On the resulting page you should notice there is a very large number of bytes allocated to `int` arrays, denoted by `[I`. You will probably see some 300+ instances of `int` arrays that have allocated over 4,000,000+ bytes. This pattern should draw your attention enough to investigate further. For example, if you divide 4,000,000 by 300, you should get an average of 13,000 bytes per instance. In other words, each `int` array averaging 13,000+ bytes requires further investigation.

11. Click the back button on the browser window and click on `Execute Object Query Language (OQL) query`.
12. Enter the following query into the `Object Query Language` box and press the `Execute` button:

```
select i from [I i where i.length > 13000
```

The purpose of this step is to formulate and execute a query that identifies references to `int` arrays larger than 13,000.

13. The query returns web page containing two references (displayed as links on the browser). Follow each link by clicking on it.

Notice that each one of these are `int` arrays are references in a `HashMap.Entry`. You can see this by looking at the "References to this object:". Also take a closer look at the values in the `int` arrays. Scanning over them quickly you can see that the positive numbers listed happen to be prime numbers. The one reference looks to hold what is clearly a large set of prime numbers. Remember we were running a program that calculated prime numbers.

14. The next step is to identify who has references to the `HashMap.Entry` holding the integer arrays. Perform the following actions on both links.

If you keep clicking on the `References to this object`, you will see that a `HashMap.Entry` might be referenced by another `HashMap.Entry`. That is acceptable because it can be part of a bucket chain in a `HashMap`. Eventually you will see the `HashMap`

that references the `HashMap.Entry`. Again click on the References to this object to find who holds the reference the `HashMap`. You will eventually find that it is a `primenumbers.PrimeNumbers` instance field `completeResults_` for one of them and the other is a `primenumbers.PrimeNumbers` instance field `cache_`.

What is known about the program is that it caches previous results. So, the reference that traces back to the `primenumbers.PrimeNumber.cache_` is not a likely suspect. So the next step is to look at the source code for `primenumbers.PrimeNumbers.completeResults_` and look at how it is used.

15. Extract the `PrimeNumbers.zip` file which contains the source files for the `PrimeNumbers` program. Start looking at how `PrimeNumbers.completeResults_` is used. You will find it is declared in `PrimeNumbers` and you will find values are *put* in it in `PrimeCalculator.construct` method. But, you should find no other usage. The fact that `completeResults_` is a `HashMap` of `int[]` keys and `int[]` values and since only `put` methods are invoked suggests that the memory leak has been found.

Instructor note: An optional exercise is to suggest a fix to this memory leak. (Actually, this is not a great example since the demo program has several poor programming issues with it. But, having students find those probably would be a good exercise too. We could add this later though.)

16. You can close the browser window and exit the `jhat` application

## Instructions: Secondary Exercise

1. Launch VisualVM (or Java VisualVM).
2. Select `File > Load` and load the same `.hprof` file that was generated when the `PrimeNumber` GUI when it experienced an `OutOfMemoryError`. You should have made note of the file name in step 7 of the previous exercise.

This will load the heap dump in VisualVM. On the right panel you should a Summary.

3. Click on the `Classes` button on the right and order by `Size` in descending order.
4. Notice the same `int[]` pattern discovered in the previous exercise. Again with some 300+ instances and 4,000,000+ bytes allocated. Select the `int[]` row, right click and select `Show Instances View`.
5. In the `Instances` view, the left panel is ordered by instances allocating the most bytes. If you select the top one, you will notice that the `References` panel in the lower right shows the instance with the most bytes allocated in `int[]` as a `HashMap.Entry`. If you select and right click on the `int[]` or either the `HashMap.Entry.key` or `HashMap.Entry.value` and select `Show Nearest GC Root` you will find previously discovered `PrimeNumbers.completeResults_`.
6. For a second use case, leave VisualVM running and restart the the `PrimeNumbers` application again, this time without the `-XX:+HeapDumpOnOutOfMemoryError` command line switch.

Notice in the left panel of VisualVM automatically discovered the `PrimeNumbers` GUI Java application.

7. Select and right click on the `PrimeNumbers` GUI Java application icon in the left panel of VisualVM and select the `Enable Heap Dump on OOME` option. The `PrimeNumbers` application will create a heap dump when it hits an `OutOfMemoryError`.

8. Again, enter 1000000 in the `PrimeNumbers` GUI. Do it a second time to generate an `OutOfMemoryError`. Notice where the heap file was written in the output window where the `PrimeNumbers` application was launched.
9. Kill the `PrimeNumbers` GUI if it is hung.
10. Load the heap dump file in VisualVM as you did in step 2, but this time you be loading a new heap dump file as identified in step 8.
11. Repeat steps 3 - 5 and you will find the same results. The intent here is to show the two uses cases of one; having a heap dump generated by an explicit `-XX:+HeapDumpOnMemoryError` and second; enabling a heap dump on memory error on a running application. And, in both use cases being able to evaluate the heap dump in VisualVM.

Note: NetBeans Profiler can also load heap dumps. But, it cannot enable heap dump on out of memory errors on running applications.

### ***Part III: (Optional) HeapWalker Demo with NetBeans Profiler***

#### ***Main Points to Hit***

- The HeapWalker provides a complete picture of the objects on the heap **and** the references between the objects.
- Is especially useful for analyzing binary heap dump files produced when an `OutOfMemoryError` occurs.
- Find Nearest GC root feature can help track down memory leaks by showing the reference that prevents an object from being garbage collected.
- This same or similar demo could be with VisualVM too.

#### **Prerequisites**

- JDK6
- NetBeans 6.1 (or higher)

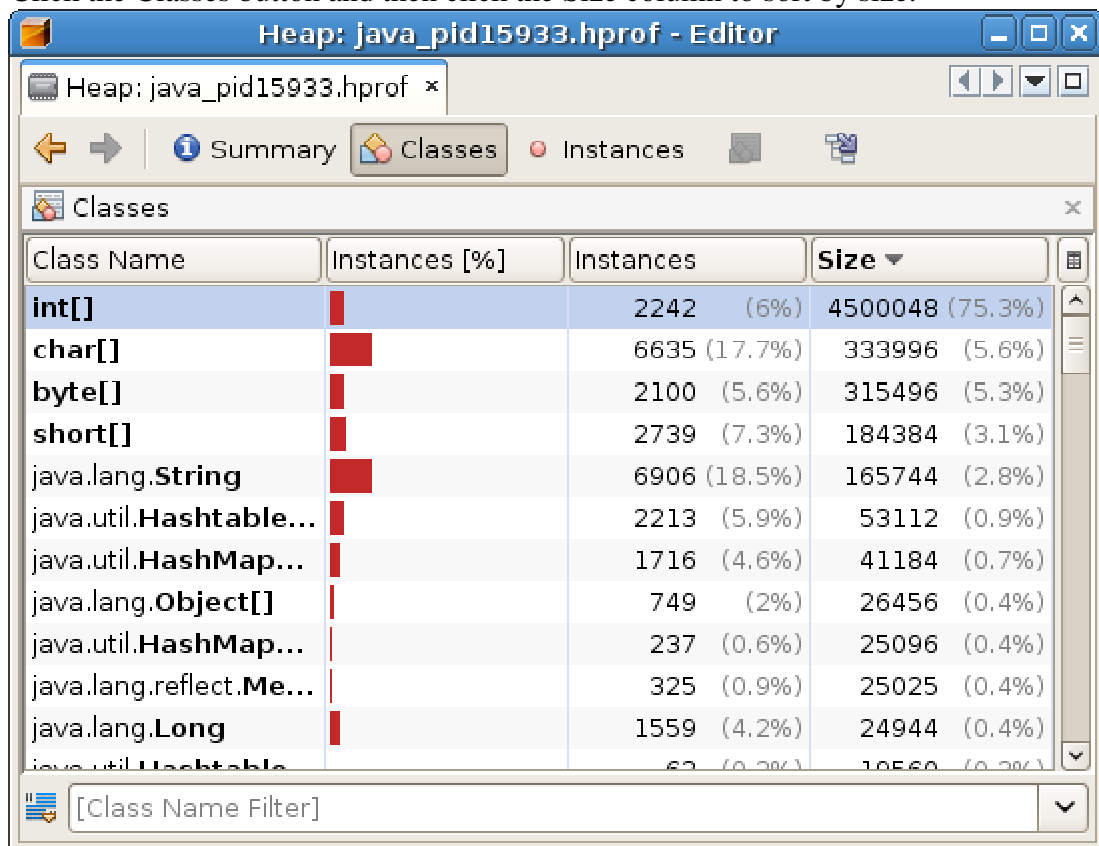
#### **Setup**

1. Grab the `PrimeNumbers.zip` file from `demos/mod04/sw` and unzip into a directory. It's a NetBeans Project.
2. Open the `PrimeNumbers` project
3. Profile the `PrimeNumbers` project at least once so that the IDE can add the profile target to `build.xml`
4. Right-click `PrimeNumbers` and select `Properties`; then select `Run`. Verify that `VM Options` is set to `-Xmx6m`

#### **Demo Steps**

1. Right-click the `PrimeNumbers` project and select `Profile Project`
2. Click the `Monitor` button on the left side of the dialog and then click the `Run` button. Suggested Comment (SC): "Okay, so here's a silly little prime number generator."
3. When the UI displays, make sure that the IDE's window is visible in the background. In the

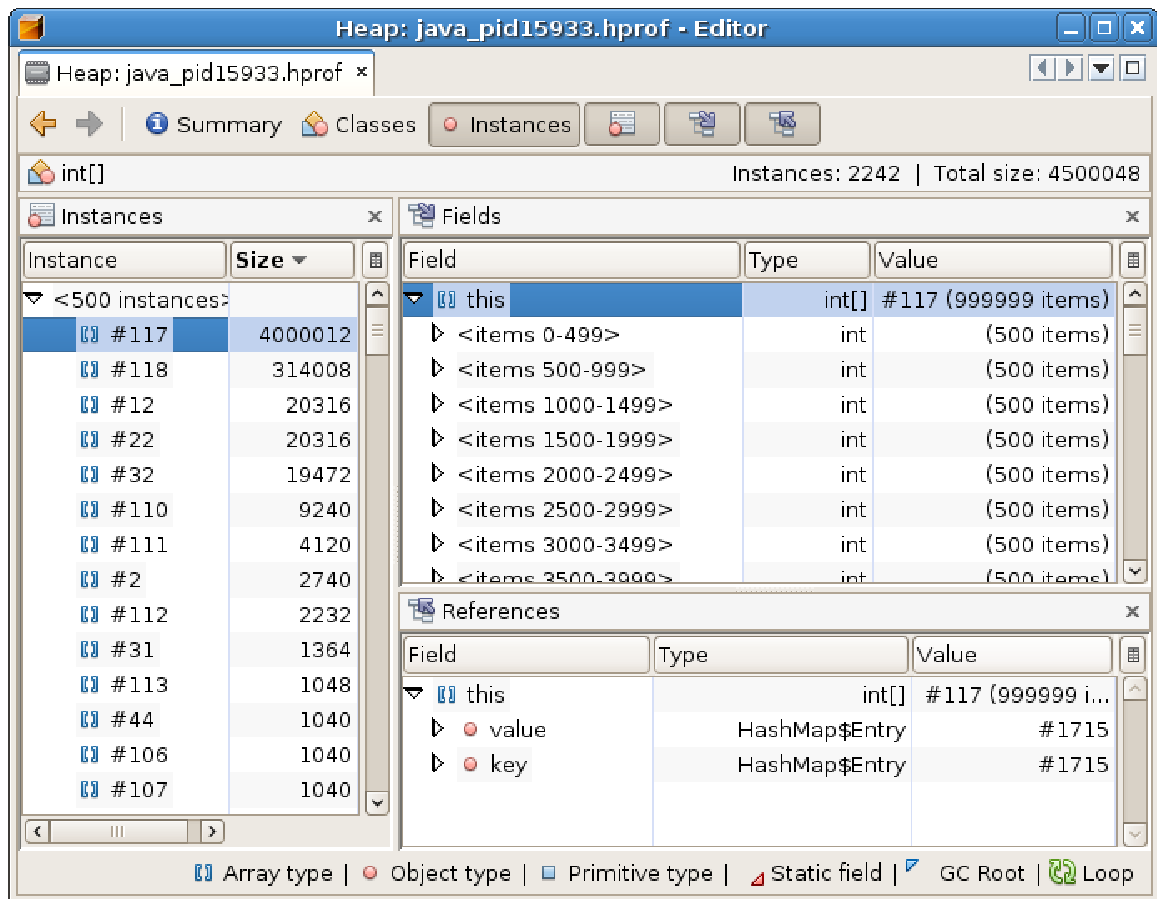
- application's "Enter a number" field type in 1000000 (that is a one followed by six zeroes) and then click the Calculate Prime Numbers button. SC: "I put in a number and then it will calculate the largest prime number less than or equal to that number."
- The result displays 999983. SC: "That was quick! Now if I click the button again it should be even faster because this program caches the answers it calculates...."
  - Click the Calculate Prime Numbers button again. The application dies with an `java.lang.OutOfMemoryError: Java heap space`, which is visible in the Output window of the IDE. SC: "Uh-oh...."
  - Switch to the IDE window. It will pop up a dialog asking if you want to open the heap dump that was generated in the heapwalker. Click Yes. SC: "An OutOfMemoryError was thrown so the profiler requested a standard binary heap dump snapshot from the JVM. We can open it in the profiler's heapwalker to take a look at what is on the heap."
  - The heapwalker opens up with a Summary view. SC: "We can see the summary here: size, operating system, and the JVM system properties."
  - Click the Classes button and then click the Size column to sort by size:



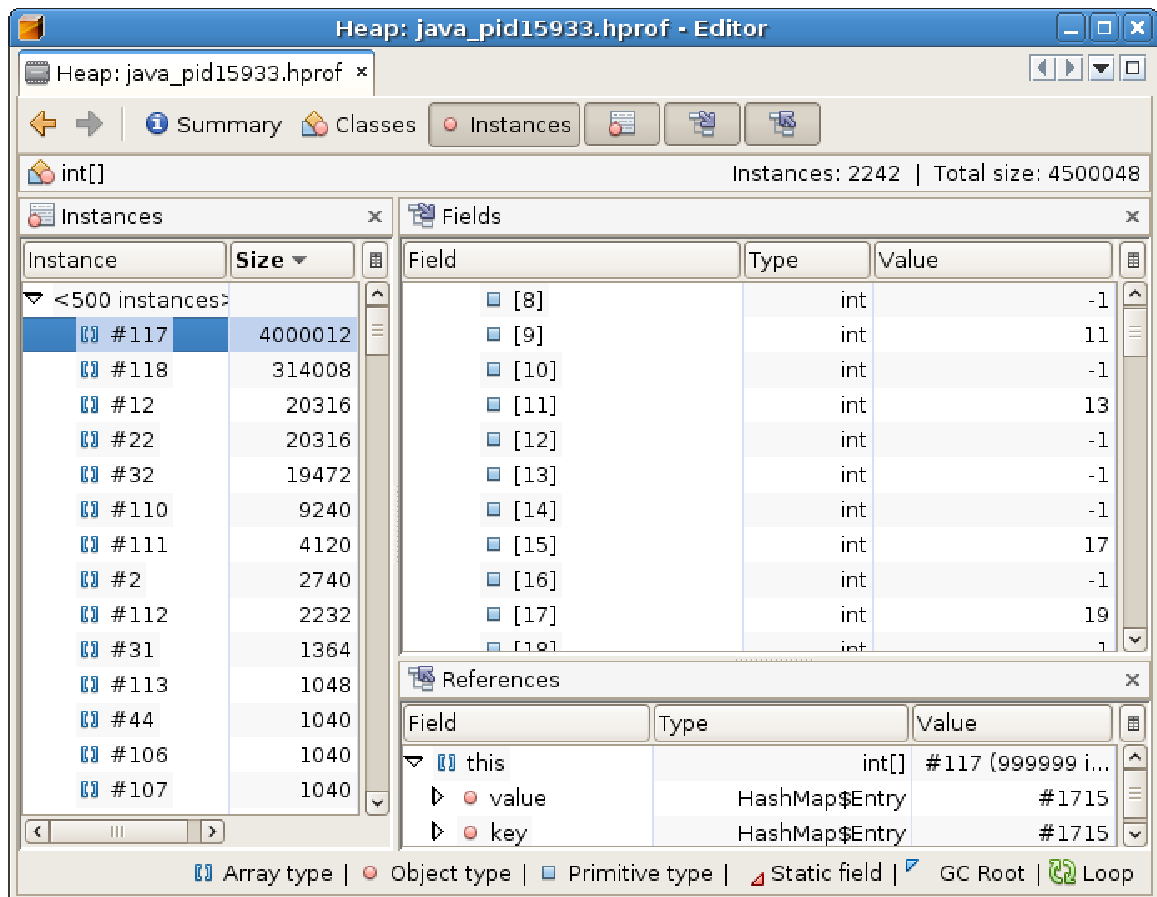
The screenshot shows the 'Heap: java\_pid15933.hprof - Editor' window. The 'Classes' tab is selected, and the list is sorted by 'Size'. The 'int[]' class is highlighted as the largest.

Class Name	Instances [%]	Instances	Size
<b>int[]</b>		2242 (6%)	4500048 (75.3%)
<b>char[]</b>		6635 (17.7%)	333996 (5.6%)
<b>byte[]</b>		2100 (5.6%)	315496 (5.3%)
<b>short[]</b>		2739 (7.3%)	184384 (3.1%)
java.lang.String		6906 (18.5%)	165744 (2.8%)
java.util.Hashtable...		2213 (5.9%)	53112 (0.9%)
java.util.HashMap...		1716 (4.6%)	41184 (0.7%)
java.lang.Object[]		749 (2%)	26456 (0.4%)
java.util.HashMap...		237 (0.6%)	25096 (0.4%)
java.lang.reflect.Me...		325 (0.9%)	25025 (0.4%)
java.lang.Long		1559 (4.2%)	24944 (0.4%)
java.util.Hashtable...		62 (0.2%)	10560 (0.2%)

- SC: "This is a list of all the classes that are on the heap, along with the number of object instances of each and the total size occupied by those object instances."
- Click on the first line, which is for int array in order to highlight it. SC: "Hmmm... looks like most of the heap is taken up by int arrays. Let's take a look at the object instances for int array."
- Right-click the int array entry and choose Show in Instances View. SC: "There are over 2,000 int arrays on the heap, but the instances view sorts them by default by size. The first one listed is the largest and it has an interesting size: 4 million bytes. Hmmm.... Let's take a look inside."
- Click on the first int array instance (the one that is 4 million bytes). SC: "Now that I have selected an instance, I can see two things over on the right: its fields and a list of any objects that reference this particular instance."



13. Expand the <items 0-499> entry. SC: "Since this is an array, the list of Fields is actually a list of array indexes, in groups of 500. I've expanded the first group, let's take a look at what is in here...."
14. Scroll down through the list. SC: "Hmmm.... Looks like this array holds **all** the prime numbers less than the requested value, with place holder entries for integers that are not prime."



15. Close the Fields panel. SC: "We have found something on the heap that should not be there - in this case, this appears to be an array that was used during the calculation that should have been garbage collected after the calculation. So why didn't the JVM's garbage collector remove it? There must be some accidental reference to it that is being made (and that should be cleared). This is where the References information comes in handy."
16. Right-click the entry for `this` in the References panel and click Show nearest GC Root. SC: "Garbage Collection roots are the objects that never get removed from the heap - they are the starting point for the JVM's garbage collector. Any object that is reachable from a GC root cannot be removed from the heap."
17. The display expands so that the entry for `primeNumbers` is shown. SC: "There are actually several GC roots in this case, but what is often more interesting is what we can learn by looking at the object references along the way, since if they were to let go of their reference this array would be eligible for removal by the garbage collector. Notice this variable called `completeResults_`."
18. Right-click `completeResults_` and choose Go To Source. "An advantage of an integrated tool - easy access to the source code."
19. The `PrimeNumbers.java` file opens up; the `completeResults_` variable is on line 31. Right-click it and select Find Usages. SC: "Interesting. This is a Map - let's see where it is used."
20. When the Usages window opens with the results, double-click the only result. SC: "Well, well, well. The complete list of candidate prime numbers is being put into this Map, but as we can see from the Usages results **it is never removed**. So we don't need the array anymore, but it cannot be garbage collected - that is a memory leak. And one easily found with the profiler's heapwalker." :-)
21. Select Profile > Load Heap Dump. SC: "One last thing to note – as you've seen earlier the profiler's heapwalker can also be used on any binary heap dump file produced by one of Sun's JVMs. There are a variety of ways to get a JVM to produce a heap dump; as an

example, there is a command line flag that you can use to have the JVM create the file whenever an `OutOfMemoryError` is thrown. As you've seen earlier, you can then open that file with this feature."

22. Here we have seen several alternative ways to find memory leaks and find who is holding references to objects who should not.



## Module 4 Demo 7: Finding Contended Locks

Large values of voluntary thread context switches in an application observed when monitoring with `mpstat`, can be a symptom of lock contention in a Java application. This exercise investigates Java applications for the presence of contended locks using the Sun Studio Collector Analyzer.

Slide id: 158

### Required Software

- Solaris OS (Sparc or X86)
- JDK 1.5 or higher
- Sun Studio version 11 or 12

You can download Sun Studio from <http://developers.sun.com/sunstudio/downloads/>.

- Demo software

The following software is located in the `demo/mod04/sw` directory.

- `BatchProcessor.jar`
- `BatchProcessor.java`
- `ReadThread.java`
- `WorkQueue.java`
- `WriteThread.java`

### Hardware Recommendation

This exercise tends to work the best on a system which two logical processors (that is single processor dual core Intel). It is hard to mimic lock contention on large multi-core systems.

### Objective

- Use Sun Studio Collector and Analyzer tools to detect lock contention in Java technology applications.

### Preparations

Check for required version of Java SE.

1. Use the `java -version` command to check the installed Java SE version.

If the installed version is earlier than Java SE 5, download and install the latest Java SE version from the following URL.

<http://java.sun.com/javase/downloads/index.jsp>

Download and install Sun Studio.

1. Download and and install Sun Studio from the following URL:  
<http://developers.sun.com/sunstudio/downloads/>
2. Add the `bin` subdirectory of the Sun Studio install directory to the Solaris OS `PATH` variable. For example, if you installed Sun Studio in the default location `/opt/SUNWspro` you can enter the following commands at a terminal window.

```
PATH=/opt/SUNWspro/bin:$PATH
export PATH
```

## Instructions

1. Open a command line window.  
`cd <course_install_dir>/demo/mod04` directory
2. Use the `javac` command to compile the following Java source files.

```
javac BatchProcessor.java
javac ReadThread.java
javac ReadThread.java
javac WriteThread.java
```

3. Run the `BatchProcessor` program to ensure it runs in your environment.

```
java BatchProcessor
```

This program will run for 90 seconds and print some statistics. What it prints is unimportant. The objective is to test that it runs.

4. Run the Sun Studio Collector utility using the following command line.

```
collect -j on java -server -Xmx64m -Xms64m -Xmn24m -XX:+UseSerialGC -
jar BatchProcessor.jar
```

This invokes the Sun Studio Collector to capture a running profile of the Java program. It creates a directory in the current directory called `test.er.1`. If the file is not created check that you have write permissions to the directory from which you ran the program.

The default sampling interval is 1 second and the default output directory is the present working directory. Please refer to the Sun Studio Collector documentation for command line options that override default values.

Let the `BatchProcessor` program run to completion.

This program tends to work the best on a system which two logical processors (that is. single processor dual core Intel). It is hard to mimic lock contention on large multi-core systems.

5. When the `BatchProcessor` application completes, run the Sun Studio Analyzer program to read the results:  

```
analyzer test.er.1
```
6. In Analyzer, select `View > Set Data Presentation` from the main menu, then check the `User Lock Exclusive` and `Inclusive Time` boxes and press `Apply`.
7. Sort the data by `Exclusive User Lock` in descending order.
8. Locate and select the `java.lang.Object.wait` method. Then, select the `Callers - Callees` tab.
9. On the `Callers - Callees` view, you should notice that `Object.wait(long)` method is called by `Object.wait` method. Click on the upper `Object.wait` method to identify who calls it. You should see that it is called by the `WorkQueue.dequeue` method. You should also notice that `WorkQueue.dequeue` method has some `Exclusive User Lock` time on it. Any time you see a method that has `Exclusive User Lock Time` on it that is not `java.lang.Object.wait` method, then it is likely a synchronized method or has a

synchronized block in it.

Click on `WorkQueue.dequeue` method to identify who is calling it. You should see that `ReadThread.getItemFromQueue` method calls the `WorkQueue.dequeue` method.

Examine the source code for `WorkQueue.dequeue` method located in the `WorkQueue.java` file. You will see that it is a synchronized method and it contains a call to the `Object.wait` method in it. Now you should be able to distinguish between the time spent on the User Lock for the `Object.wait` method in the `WorkQueue.dequeue` method versus time spent in the synchronized method `WorkQueue.dequeue` method. Underneath `WorkQueue.dequeue` method in the Callers – Callees view you will see that it calls `Object.wait` method and `LinkedList.poll` method. You can also see how much time is spent in User Lock `Object.wait` method versus `WorkQueue.dequeue` method. You probably noticed the `Object.wait` method that's in `WorkQueue.dequeue` method is an `Object.wait` method that is executed by thread waiting to be notified there is work on queue for it to grab and execute.

10. At this point you would have identified that there is a lock on `WorkQueue.dequeue` method that is being contended for and the `Object.wait` method that's being called in `WorkQueue.dequeue` method is associated with the traditional Thread wait/notify concept.
11. Go back to the Functions view and scroll to the top.
12. You may notice there is an entry for JVM–System. If you want to know **what is being contended for locks** internally within the JVM, you can change User Modes to Expert or Machine. However, looking at JVM internals is not the purpose of this exercise.
13. If you continue to look down the list of Exclusive User Locks, for the `WorkQueue.enqueue` method. Select it and press the Callers – Callees tab. Examine the source code for `WorkQueue.enqueue` method and you will notice that it is a synchronized method. You will also notice in Analyzer it is called by a `WriteThread.putWorkItemOnQueue` method. Remember that `WorkQueue.dequeue` method was called by a `ReadThread.putWorkItemOnQueue` method. So, it looks like there are reader and writer threads putting and removing from a shared queue. The amount of time spent in the User Lock for the `WorkQueue.enqueue` and `WorkQueue.dequeue` method is the amount time spent contending for the lock on the shared queue.

## Module 5 Demo 1: JVM Heap Sizing

Present the general school of thought behind sizing of GC heap spaces and using various GC monitoring tools.

Slide id: 184

### Required Software

- Any OS
- JDK 1.5 or higher
- VisualVM with VisualGC plug-in

### Objective

- To show students the general school of thought behind sizing of GC heap spaces and using various GC monitoring tools.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Overview

We are going to run the Java2Demo and try to tune its GC spaces. We'll tune it for the serial collector, concurrent collector and throughput collector.

\* The school of thought to emphasize here is to size the GC heap spaces to avoid a Full GC and keep the young gen / minor gc events rather short also. For a GUI application, responsiveness is important, so we want to keep pause times to a minimum. But, if this was a batch processing engine, we wouldn't care so much about pause times. Instead we would be interested in raw throughput, (least amount of time spent in GC). These are two different goals and require a slightly different strategy when sizing heap spaces. A point should be made as to knowing or defining what are the goals and pause time requirements for an application before starting the tuning exercise.

In tuning the Java2Demo, our goal is to minimize pause times, let's say to under 10 milliseconds and avoid all Full GC events.

## Instructions

1. Open a command line window and set your PATH environment variable to include both the JVM and other monitoring GUI tools if you will be using them.
2. Start the Java2Demo application with the following command line as a starting point: (Note we know that the Java2Demo does explicit GC calls using System.gc(). To keep the Java2Demo from responding to those explicit calls to System.gc() we will add the -XX:+DisableExplicitGC command line switch. Also note we are going to explicitly use the Serial collector -XX:+UseSerialGC)  

```
<JDK install dir>/bin/java -client -XX:+DisableExplicitGC -XX:+UseSerialGC -jar <JDK install dir>/demo/jfc/Java2D/Java2Demo.jar
```
3. From here start using the command line tools, either by adding -verbose:gc, -XX:+PrintGCDetails, or use jstat to monitor the Java2Demo application.
  - a.) If you see Full GCs occurring as a result of a growing Java heap space, you need to first figure out a maximum Java heap size first. Click across all of the tabs, which will force the loading of almost all the needed class files. Once you've done that, you should be able to look at the Java heap in use and add some "fudge" to the upper value and determine a max Java heap space, (-Xmx).
  - b.) You will probably observe some Java heap space expansion on Full GC events. If you set -Xmx and -Xms to the same value, that will prevent young gen and old gen from getting resized. Remember, re-sizing Java spaces require a Full GC to do so.
  - c.) Also remember that permanent generation space may be getting resized. So, you will need to use a tool to monitor permanent generation Java heap space and then adjust or size permanent generation space. Remember that you will probably have to set both -XX:MaxPermSize and -XX:PermSize. You may be able to get by just sizing -XX:PermSize.
  - d.) If young gen / minor collections are too lengthy, you may need to set the young generation space size. You can experiment with sizing young generation to see what impact it has on gc time. You can try sizing it larger or smaller to see the result. Changing it will likely impact how often the gc event occurs and how long it takes to perform the gc.

Now, repeat the same exercise using a GUI monitoring tool such as VisualGC or JConsole or VisualVM.

Then, after doing this with a GUI tool, switch to using concurrent collector, swap out -XX:+UseSerialGC in the command line above to using -XX:+UseConcMarkSweepGC. With the concurrent collector, you will probably notice that you will need a larger overall Java heap space. But, you may be able to use a smaller young generation space.

Again, use both a command line tool and a GUI tool. Personally, for GUI tools I like using VisualGC the best.

If you feel adventurous, you could attempt to tune the GC spaces for NetBeans IDE. The java command line args for launching NetBeans is in <NetBeans install dir>/etc/netbeans.conf. If you want to tune GC spaces for launching NetBeans, you will probably want to use a command line tool. You can place -verbose:gc and -XX:+PrintGCDetails in the netbeans.conf file. To launch NetBeans you will need to launch it from the command line at <NetBeans install dir>/bin/ and the script / batch / exe file 'netbeans'. It'll be very hard to monitor NetBeans launch / startup with GUI monitoring tools. But, you could monitor NetBeans with one of the GUI tools and do some simple tasks like open a NetBeans project and compile the opened NetBeans project. One could open one of the NetBeans projects in the JDK/demo directory.

Comment to Instructor: This is kind of a long demo. But, I think it's needed to help students understand how to tune Java heap spaces for an application.

## Module 5 Demo 2: Using the PrintGCStats Script

This demo obtains and interprets GC output produced by the `-verbose:gc` option of the `java` command.

Slide id: 226

### Required Software

- Solaris OS
- JDK 1.5 or higher

### Objective

- Use the `PrintGCStats` script to interpret output produced by the `-verbose:gc` option of the `java` command.

### Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: `java version 1.5.0_00` or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the `bin` subdirectory of the Java SE version, to the OS `PATH` variable.

### Instructions

1. Open a command line window and change directory to the exercise directory for this exercise:  
`cd demos/mod05/sw/PrintGCStats`
2. Use the `ls` command to list the directory.
3. Examine the directory listing to confirm the directory contains the following files:

- `PrintGCStats`

`PrintGCStats` is a shell script which mines the `verbose:gc` logs and summarizes statistics about garbage collection.

- `license.txt`

This file contains the license text.

- `PrintGCStats_manual.pdf`

This file contains the user manual for the `PrintGCStats` script.

- `gc-logs.txt`

The `gc-logs.txt` file GC output captured from a run of the SPECjbb2005

benchmark. It was obtained with `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps` JVM command line arguments on a dual core Opteron (2 logical CPUs).

5. Optionally use a PDF viewer to open and examine the `PrintGCStats_manual.pdf` file. This file contains the documentation for the `PrintGCStats` script.

6. Analyze the `gc-logs.txt` file using the `PrintGCStats` script:

```
PrintGCStats -v ncpu=2 gc-logs.txt
```

Your instructor will assist you with interpreting the results.



## PrintGCStats User Guide

`PrintGCStats` is a shell script which mines the `verbose:gc` logs and summarizes statistics about garbage collection, in particular, the GC pause times (total, averages, maximum and standard deviations) in the young and old generations. It also calculates other important GC parameters like the GC sequential overhead, GC concurrent overhead, data allocation and promotion rates, total GC and application time and so on. In addition to summary statistics, `PrintGCStats` also provides the timeline analysis of GC over the application run time, by sampling the data at user specified intervals.

- **Input:**

The input to this script should be the output from the HotSpot Virtual Machine when run with one or more of the following flags.

- `-verbose:gc`

Produces minimal output, so statistics are limited, but available in all JVMs

- `-XX:+PrintGCTimeStamps`

Enables time-based statistics (for example, allocation rates, intervals), but only available from Java SE 1.4.0 and later.

- `-XX:+PrintGCDetails`

Enables more detailed statistics gathering, but only available from Java SE 1.4.0 and later.

- `-XX:+PrintGCDetails`

```
java -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails  
...
```

- **Usage:**

```
PrintGCStats -v ncpu=<n>[-v interval=<seconds>][-v verbose=1]  
<gc_log_file>
```

- `ncpu`

Number of cpus on the machine where the Java application was run. Used to compute cpu time available and GC *load* factors. No default; must be specified on the command line (defaulting to 1 is too error prone).

- `interval`

Print statistics at the end of each interval to provide timeline analysis; requires output from `-XX:+PrintGCTimeStamps`. Default is 0 (disabled).

- `verbose`

If non-zero, print each item on a separate line in addition to the summary statistics.

## Module 5 Demo 3: Using GCHisto

This demo shows how GCHisto can be used to understand GC statistics gathered from a running application.

Slide id: 235

### Required Software

- Any OS, Windows, Solaris or Linux
- JDK 6.0 or higher
- GChisto.jar from demos/mod05/sw/GCHisto
- gc-logs-1.txt, gc-logs-2.txt, gc-logs-3.txt and gc-logs-4.txt from demo/mod05/sw/GCHisto

### Objective

- Use GCHisto to view and understand GC behavior.

### Preparations

Verify and if required download and install Java SE 6 or later.

1. Verify if you are running Java SE 6 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.6.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

1. Open a command line window and change directory to the exercise directory for this exercise:  
`cd demos/mod05/sw/GCHisto`
2. Use the `ls` command to list the directory.
3. Examine the directory listing to confirm the directory contains the following files:

- GChisto.jar
- gc-logs-1.txt, gc-logs-2.txt, gc-logs-3.txt, gc-logs-4.txt

The gc-logs-4.txt file GC output captured from a run of the SPECjbb2005 benchmark. It was obtained with `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps` JVM command line arguments on a dual core Opteron (2 logical CPUs). The others were captured with only `-verbose:gc` and `-XX:+PrintGCTimeStamps`

Analyze the gc-logs.txt file using GChisto:

```
java -jar GChisto.jar
```

Add gc-logs-4.txt to the GChisto. Go through each of the tabs in GChisto. Each tab is pretty self-explanatory.

The instructor will assist you with interpreting the results.

Now, load the remaining gc logs files; gc-logs-1.txt gc-logs-2.txt gc-logs-3.txt

Now, compare the gc logs of each run to highlight the additional abilities of GChisto. Also point out any interesting observations.

## ***Additional Information on GChisto Panels***

### **GC Pause Stats Pane**

This pane shows some GC statistics (GC number, total time, etc.) for the loaded traces and broken down by GC type. If only one trace is loaded, then this pane will show the stats for that trace. If more than one trace are loaded, this pane will show the stats for all traces (say 1 to N) and, optionally, will also show comparison stats for traces 2 to N as compared against trace 1. This is a nice and intuitive way to compare different runs of the JVM with, say, different GC settings and tuning.

Apart from the table with the stats, this pane has several sub-panes with graphs for different metrics.

### **GC Pause Distribution Pane**

This pane shows the pause time distribution for the loaded traces, with one sub-pane per loaded trace. A set of checkboxes on the left allows you to make individual GC types visible / invisible.

### **GC Timeline Pane**

This pane shows the GC timeline for each trace, with one sub-pane per loaded trace. A set of checkboxes on the left allows you to make individual GC types visible / invisible.

### **Zooming In And Out On The Charts**

In all the charts it is possible to zoom in and out to take a better look at some detail in the chart. Left-clicking on the chart and then dragging the mouse will create a selection rectangle and then the chart will zoom into that rectangle. Right-clicking on the chart will give you a pop-up menu with some options. To totally zoom back out choose “Auto Range,” then “Both Axes” from that menu.

## PrintGCStats User Guide

`PrintGCStats` is a shell script which mines the `verbose:gc` logs and summarizes statistics about garbage collection, in particular, the GC pause times (total, averages, maximum and standard deviations) in the young and old generations. It also calculates other important GC parameters like the GC sequential overhead, GC concurrent overhead, data allocation and promotion rates, total GC and application time and so on. In addition to summary statistics, `PrintGCStats` also provides the timeline analysis of GC over the application run time, by sampling the data at user specified intervals.

- **Input:**

The input to this script should be the output from the HotSpot Virtual Machine when run with one or more of the following flags.

- `-verbose:gc`

Produces minimal output, so statistics are limited, but available in all JVMs

- `-XX:+PrintGCTimeStamps`

Enables time-based statistics (for example, allocation rates, intervals), but only available from Java SE 1.4.0 and later.

- `-XX:+PrintGCDetails`

Enables more detailed statistics gathering, but only available from Java SE 1.4.0 and later.

- `-XX:+PrintGCDetails`

```
java -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails  
...
```

- **Usage:**

```
PrintGCStats -v ncpu=<n>[-v interval=<seconds>][-v verbose=1]  
<gc_log_file>
```

- `ncpu`

Number of cpus on the machine where the Java application was run. Used to compute cpu time available and GC *load* factors. No default; must be specified on the command line (defaulting to 1 is too error prone).

- `interval`

Print statistics at the end of each interval to provide timeline analysis; requires output from `-XX:+PrintGCTimeStamps`. Default is 0 (disabled).

- `verbose`

If non-zero, print each item on a separate line in addition to the summary statistics.

## Module 6 Demo 1: Examining JIT compiler optimizations

This exercise illustrates some of the compiler optimizations made by the JIT compiler. This exercise / demo comes from a chapter of the upcoming book on Java Performance, titled Java Performance.

Slide id: 257

### Required Software

- Any OS
- JDK 6.0 or higher
- JDK 6.0 or higher HotSpot debug JVM

### Objective

- This exercise illustrates some of the compiler optimizations made by the JIT compiler.

### Preparations

Verify and if required download and install Java SE 6 or later.

1. Verify if you are running Java SE 6 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.6.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

### Instructions

Read the following text from the book and run the programs as appropriate.

#### Optimizing Away Dead Code

Modern JVMs have the ability to identify code which is never called through the form of static analysis, through run time observation and through a form of lightweight profiling. Since micro-benchmarks rarely produce significant output it is often the case some portions of micro-benchmark code can be identified as dead code by a JVM's dynamic compiler. In extreme cases the code of interest being measured could be completely optimized away without the creator or executor of the micro-benchmark knowing. The following micro-benchmark, which attempts to measure the time it takes to calculate the 25<sup>th</sup> Fibonacci<sup>1</sup> number is an example where a modern JVM's dynamic compiler can find dead code and eliminate it.

```
public class DeadCode1 {
```

---

1 Fibonacci, also known as Leonardo di Pisa, in his thirteenth century book *Liber abaci* posed the question; two young rabbits, one of each sex, is placed on an island. A pair of rabbits do not reproduce until they are two months old. After they are two months old, each pair of rabbits produces another pair each month. What is an equation that models the number of pairs of rabbits on the island after n months, assuming no rabbits die?

```

final private static long NANOS_PER_MS = 1000000L;
final private static int NUMBER = 25;

// Non-recursive Fibonacci calculator
private static int calcFibonacci(int n) {
    int result = 1;
    int prev = -1;
    int sum = 0;
    for (int i = 0; i <= n; i++) {
        sum = prev + result;
        prev = result;
        result = sum;
    }
    return result;
}

private static void doTest(long iterations) {
    long startTime = System.nanoTime();
    for (long i = 0; i < iterations; i++)
        calcFibonacci(NUMBER);
    long elapsedTime = System.nanoTime() - startTime;
    System.out.println("    Elapsed nanoseconds -> " +
        elapsedTime);
    float millis = elapsedTime / NANOS_PER_MS;
    float itrPerMs = 0;
    if (millis != 0)
        itrPerMs = iterations/millis;
    System.out.println("    Iterations per ms ---> " +
        itrPerMs);
}

public static void main(String[] args) {
    System.out.println("Warming up ...");
    doTest(1000000L);
    System.out.println("Warmup done.");
    System.out.println("Starting measurement interval ...");
    doTest(900000000L);
    System.out.println("Measurement interval done.");
    System.out.println("Test completed.");
}
}

```

Notice in this example there is a warm-up period of one million iterations and a measurement interval of 900 million iterations. However, in `doTest()`, the call to method `calcFibonacci(int n)` can be identified as dead code and subsequently optimized into a no-op and eliminated. A no-op is defined as an operation or sequence of operations which has no effect on the state or output of a program. A dynamic compiler could potentially see that no data computed in `calcFibonacci()` escapes that method and may eliminate it. In other words, the dynamic compiler can determine that `calcFibonacci()` is a no-op and can eliminate the call to it as a performance optimization. The above micro-benchmark executed with a JDK 6 HotSpot Server JVM on a 2 GHz AMD Turion running Solaris 10 produced the following output.

```
Warming up ...
    Elapsed nanoseconds -> 282928153
    Iterations per ms -> 3546.0
Warmup done.
Starting measurement interval ...
    Elapsed nanoseconds -> 287452697
    Iterations per ms -> 313588.0
Measurement interval done.
Test completed.
```

Comparing the iterations per millisecond during the warm-up period and measurement interval suggests the HotSpot Server dynamic compiler has managed to increase the performance of calculating the 25<sup>th</sup> Fibonacci number by almost 9000%. A speed up of 9000% does not make sense. This is pretty strong evidence there is something wrong with the implementation of this micro-benchmark.

If this benchmark is updated with the following modifications the reported iterations per millisecond change drastically:

1. Modify the doTest() method to store the returned result of the called method, calcFibonacci(int n).
2. Print the stored result returned from the called calcFibonacci(int n) method after the elapsed time has been calculated in the doTest() method.

An updated implementation is shown below.

```
public class DeadCode2 {

    final private static long NANOS_PER_MS = 1000000L;
    final private static int NUMBER = 25;

    private static int calcFibonacci(int n) {
        int result = 1;
        int prev = -1;
        int sum = 0;
        for (int i = 0; i <= n; i++) {
            sum = prev + result;
            prev = result;
            result = sum;
        }
        return result;
    }

    private static void doTest(long iterations) {
        int answer = 0;
        long startTime = System.nanoTime();
        for (long i = 0; i < iterations; i++)
            answer = calcFibonacci(NUMBER);
        long elapsedTime = System.nanoTime() - startTime;
        System.out.println("    Answer -> " + answer);
        System.out.println("    Elapsed nanoseconds -> " +
                            elapsedTime);
        float millis = elapsedTime / NANOS_PER_MS;
        float itrPerMs = 0;
        if (millis != 0)
            itrPerMs = iterations/millis;
    }
}
```

```

        System.out.println("    Iterations per ms ---> " +
                           itrPerMs);
    }

    public static void main(String[] args) {
        System.out.println("Warming up ...");
        doTest(1000000L);
        System.out.println("Warmup done.");
        System.out.println("Starting measurement interval ...");
        doTest(900000000L);
        System.out.println("Measurement interval done.");
        System.out.println("Test completed.");
    }
}

```

Executing this modified version produces the following output.

```

Warming up ...
  Answer -> 75025
  Elapsed nanoseconds -> 28212633
  Iterations per ms -> 35714.0
Warmup done.
Starting measurement interval ...
  Answer -> 75025
  Elapsed nanoseconds -> 1655116813
  Iterations per ms -> 54380.0
Measurement interval done.
Test completed.

```

Now, the difference between the reported iterations per millisecond between warm-up and measurement interval is about 150%. Observing a speed up of 150% is more believable than the 9000% observed in the earlier version of the micro-benchmark. But, if this version of the micro-benchmark is executed with the `-XX:+PrintCompilation HotSpot` JVM command line switch, there appears to be some compilation activity occurring during the measurement interval.

Executing the modified micro-benchmark version above with the addition of the HotSpot JVM command line switch `-XX:+PrintCompilation` shows the following output.

```

Warming up ...
  1      DeadCode2::calcFibonacci (31 bytes)
  1%     DeadCode2::doTest @ 9 (125 bytes)
  1%     made not entrant DeadCode2::doTest @ 9 (125 bytes)
  Answer -> 75025
  Elapsed nanoseconds -> 38829269
  Iterations per ms -> 26315.0
Warmup done.
Starting measurement interval ...
  2      DeadCode2::doTest (125 bytes)
  2%     DeadCode2::doTest @ 9 (125 bytes)
  Answer -> 75025
  Elapsed nanoseconds -> 1650085855
  Iterations per ms -> 54545.0
Measurement interval done.
Test completed.

```



The way to eliminate the compilation activity reported during a measurement interval is to add a second warm-up period. A second warm-up period can also help show whether the benchmark is indeed fully warmed by comparing the second warm-up period's iterations per millisecond to that which is reported by the measurement interval. Although there is compilation activity occurring during the second warm-up interval, if the second warm-up period is very long relative to the compilation activity time, then the iterations per millisecond should be very close to the measurement interval iterations per millisecond.

Below is a modified version of the micro-benchmark which includes a second warm-up period of the same length as the measurement interval.

```
public class DeadCode3 {

    final private static long NANOS_PER_MS = 1000000L;
    final private static int NUMBER = 25;

    private static int calcFibonacci(int n) {
        int result = 1;
        int prev = -1;
        int sum = 0;
        for (int i = 0; i <= n; i++) {
            sum = prev + result;
            prev = result;
            result = sum;
        }
        return result;
    }

    private static void doTest(long iterations) {
        int answer = 0;
        long startTime = System.nanoTime();
        for (long i = 0; i < iterations; i++)
            answer = calcFibonacci(NUMBER);
        long elapsedTime = System.nanoTime() - startTime;
        System.out.println("    Answer -> " + answer);
        System.out.println("    Elapsed nanoseconds -> " +
                           elapsedTime);
        float millis = elapsedTime / NANOS_PER_MS;
        float itrsPerMs = 0;
        if (millis != 0)
            itrsPerMs = iterations/millis;
        System.out.println("    Iterations per ms ---> " +
                           itrsPerMs);
    }

    public static void main(String[] args) {
        System.out.println("Warming up ...");
        doTest(1000000L);
        System.out.println("1st warmup done.");
        System.out.println("Starting 2nd warmup ...");
        doTest(900000000L);
        System.out.println("2nd warmup done.");
        System.out.println("Starting measurement interval ...");
        doTest(900000000L);
        System.out.println("Measurement interval done.");
    }
}
```

```

        System.out.println("Test completed.");
    }
}

```

Below is the output from executing the modified version with the -XX:+PrintCompilation HotSpot command line switch.

Warming up ...

```

1      DeadCode3::calcFibonacci (31 bytes)
1%     DeadCode3::doTest @ 9 (124 bytes)
1%     made not entrant  DeadCode3::doTest @ 9 (124 bytes)
      Answer -> 75025
      Elapsed nanoseconds -> 40455272
      Iterations per ms -> 25000.0

```

1st warmup done.

Starting 2nd warmup ...

```

2      DeadCode3::doTest (124 bytes)
2%     DeadCode3::doTest @ 9 (124 bytes)
      Answer -> 75025
      Elapsed nanoseconds -> 1926823821
      Iterations per ms -> 46728.0

```

2nd warmup done.

Starting measurement interval ...

```

      Answer -> 75025
      Elapsed nanoseconds -> 1898913343
      Iterations per ms -> 47418.0

```

Measurement interval done.

Test completed.

By adding the second warm-up period, compilation activity no longer occurs in the measurement interval. In addition, comparing the iterations per millisecond in the second warm-up period to that of the measurement interval, the cost of the compilation activity during the second warm-up period is very small, less than 700 iterations per millisecond, or about a 1.5% difference.

To reduce the chances of code in micro-benchmarks from being identified as being dead code or drastically simplified, the following programming practices should be integrated:

- Make the computation non-trivial.
- Print computation results immediately outside of the measurement interval or store the computed results for printing outside the measurement interval

To make the computation non-trivial pass in arguments to the methods being measured and return a computation result from methods being measured. Additionally, vary the number of iterations used in the measurement interval within the benchmark or alternatively in difference runs of the benchmark. Then, compare the iterations per second values to if see they are remaining consistent as the number of iterations is varied while also tracking the behavior of the dynamic compiler with the HotSpot JVM command line switch -XX:+PrintCompilation.

## Inlining

The HotSpot JVM Client and Server dynamic compilers also has the ability to inline methods. This means that the target method at a call site is expanded into the calling method. This is done by the dynamic compiler to reduce the call overhead in calling methods and results in faster execution. In addition, the inlined code may provide further optimization opportunities in that the combined code may be simplified or eliminated in ways not possible without inlining. Inlining can also produce surprising observations in micro-benchmarks. This section presents an example which illustrates what can happen to a micro-benchmark as a result of

inlining optimization decisions made by the HotSpot Client dynamic compiler.

Consider the following micro-benchmark which attempts to measure the performance of `String.equals(String s)` when both `String` objects are the same `String` object.

```
public class SimpleExample {

    final private static long ITERATIONS = 5000000000L;
    final private static long WARMUP = 100000000L;
    final private static long NANOS_PER_MS = 1000L * 1000L;

    private static boolean equalsTest(String s) {
        boolean b = s.equals(s);
        return b;
    }

    private static long doTest(long n) {
        long start = System.nanoTime();
        for (long i = 0; i < n; i++) {
            equalsTest("ABC");
        }
        long end = System.nanoTime();
        return end - start;
    }

    private static void printStats(long n, long nanos) {
        float itrPerMs = 0;
        float millis = nanos/NANOS_PER_MS;
        if (millis != 0) {
            itrPerMs = n/(nanos/NANOS_PER_MS);
        }
        System.out.println("    Elapsed time in ms -> " + millis);
        System.out.println("    Iterations / ms ----> " + itrPerMs);
    }

    public static void main(String[] args) {
        System.out.println("Warming up ...");

        long nanos = doTest(WARMUP);
        System.out.println("1st warm up done.");
        printStats(WARMUP, nanos);

        System.out.println("Starting 2nd warmup ...");
        nanos = doTest(WARMUP);
        System.out.println("2nd warm up done.");
        printStats(WARMUP, nanos);

        System.out.println("Starting measurement interval ...");
        nanos = doTest(ITERATIONS);
        System.out.println("Measurement interval done.");
        System.out.println("Test complete.");
        printStats(ITERATIONS, nanos);
    }
}
```

Based on the information presented in the *Optimizing Away Dead Code* section in this chapter, the `SimpleExample.equalsTest("ABC")` method call from within the `for / loop` in method `doTest()` could be optimized away into dead code since the result of the call to method `SimpleExample.equalsTest(String s)` never escapes the `SimpleExample.doTest(long n)` method. However, executing the above micro-benchmark with a JDK 6 HotSpot Client JVM suggests this is not the case. The output below is from a JDK 6 HotSpot Client JVM executing the micro-benchmark above using the `-XX:+PrintCompilation` command line switch.

```
Warming up ...
1      java.lang.String::hashCode (60 bytes)
2      java.lang.String::charAt (33 bytes)
3      java.lang.String::equals (88 bytes)
4      SimpleExample::equalsTest (8 bytes)
1%     SimpleExample::doTest @ 7 (39 bytes)
1st warm up done.
Elapsed time in ms -> 96
Iterations / ms ----> 104166
5      java.lang.String::indexOf (151 bytes)
Starting 2nd warmup ...
6      SimpleExample::doTest (39 bytes)
2nd warm up done.
    Elapsed time in ms -> 95
    Iterations / ms ----> 105263
Starting measurement interval ...
Measurement interval done.
Test complete.
    Elapsed time in ms -> 42870
    Iterations / ms ----> 116631
```

The output above shows there is very small improvement in the iterations per milli-second between the first warm-up period and the second warm-up period or measurement interval. It is less than 20%. The output also shows dynamic compiler optimizations were performed for `String.equals()`, `SimpleExample.equalsTest()` and `SimpleExample.doTest()`. Although the dynamic compiler has performed optimizations for those methods, it may not have optimized any part of the micro-benchmark into dead code. To gain a further understanding of what is happening with a micro-benchmark and the dynamic compiler, a HotSpot debug JVM can be used. A HotSpot debug JVM has additional instrumentation enabled within it so more can be learned about what the JVM is doing as it is executing a program.

**HotSpot debug JVMs with support for JDK 6 and later can be downloaded from the OpenJDK open source project on java.net at, <https://openjdk.dev.java.net>.**

With a HotSpot debug JVM additional information about the optimizations and decisions made by the dynamic compiler can be observed. For example, with a HotSpot debug JVM a micro-benchmark executed with the `-XX:+PrintInlining` command line switch will report which methods have been inlined. Executing this micro-benchmark with a HotSpot debug JVM using the `-XX:+PrintInlining` command line switch, as shown below, indicates `String.equals(String s)` is not inlined since the `String.equals(String s)` method is too large.

```
- @ 2    java.lang.String::equals (88 bytes)  callee is too large
    @ 16  SimpleExample::equalsTest (8 bytes)
- @ 2    java.lang.String::equals (88 bytes)  callee is too
large
```

The output suggests that increasing the inlining size could result in the `String.equals(Object o)` being inlined since the output says 88 bytes of byte code from `String.equals(Object o)` is too large to be inlined. The output below shows the effect of increasing the inlining size to 100 bytes of byte code using the HotSpot

command line switch `-XX:MaxInlineSize=100` to inline the `String.equals(Object o)` method. The micro-benchmark reported results change rather dramatically as illustrated below.

```
Warming up ...
1      java.lang.String::hashCode (60 bytes)
2      java.lang.String::charAt (33 bytes)
3      java.lang.String::equals (88 bytes)
4      SimpleExample::equalsTest (8 bytes)
1%     SimpleExample::doTest @ 7 (39 bytes)
1st warm up done.
Elapsed time in ms -> 21
Iterations / ms ----> 476190
5      SimpleExample::doTest (39 bytes)
2nd warm up done.
6      java.lang.String::indexOf (151 bytes)
Elapsed time in ms -> 18
Iterations / ms ----> 555555
Test complete.
Elapsed time in ms -> 8768
Iterations / ms ----> 570255
```

The output again reports the same methods are dynamically compiled. But, comparing the elapsed time reported to execute the warm-up periods and measurement interval to the time reported to execute the warm-up periods and measurement interval when `-XX:MaxInlineSize=100` is not specified is quite different. For example, previously 96, 95 and 42870 milliseconds were reported respectively. But, after specifying `-XX:MaxInlineSize=100`, reported elapsed times dropped to 21, 18 and 8,768 milliseconds respectively. What happened? Can inlining possibly increase the performance of a method that much? The HotSpot debug JVM can be used to confirm the `String.equals(Object o)` method is inlined when `-XX:MaxInlineSize=100` is added by using `-XX:+PrintInlining` switch and executing the micro-benchmark. The output below confirms the `String.equals(Object o)` method is indeed inlined which was not inlined in a previous run when not explicitly setting `-XX:MaxInlineSize=100`.

```
@ 2    java.lang.String::equals (88 bytes)
@ 16   SimpleExample::equalsTest (8 bytes)
@ 2    java.lang.String::equals (88 bytes)
```

Based on the reported elapsed time and iterations per millisecond, the micro-benchmark is now starting to look a like a candidate for a modern JVM dynamic compiler identifying dead code and eliminating it as an optimization, similar to what was discussed in the *Optimizing Away Dead Code* section in this chapter. By setting the inlined size so that `String.equals(Object o)` was inlined resulted in the dynamic compiler identifying dead code in the micro-benchmark and optimizing it away. Capturing a profile of both configurations, one using `-XX:MaxInlineSize=100` and one without, using a profiler such as Sun's Collector / Analyzer where the generated assembly code can be viewed, can show that the HotSpot Client dynamic compiler will indeed identify dead code and optimize away the call to `String.equals(Object o)`. For tips on how to capture a profile and view assembly language instructions generated by the HotSpot dynamic compilers with Sun Collector / Analyzer, see chapters *Profiling* and *Profiling Tips & Tricks*. The generated assembly code can also be viewed using a HotSpot debug JVM and adding the HotSpot `-XX:+PrintOptoAssembly` command line switch. The generated assembly code is printed as standard output by default and can be saved to a file by redirecting standard output to a file.

If the same micro-benchmark is executed in a HotSpot Server JVM, the reported performance will be similar to that seen by the HotSpot Client JVM when explicitly setting `-XX:MaxInlineSize=100`. The HotSpot Server JVM's dynamic compiler in its default configuration will inline and identify dead code in this micro-benchmark since it uses more aggressive inlining policies. This does not imply you should always use the HotSpot Server JVM or go through an exercise of collecting inlining data from a debug HotSpot JVM and then trying to set `-XX:MaxInlineSize` to an optimal size for your application. Both the HotSpot Server JVM

and HotSpot Client JVM have undergone rigorous performance testing of a large number of benchmarks and workloads. It is only after careful analysis of the performance data has the default inlining size been chosen.

Obviously this micro-benchmark presented in this section would require some modifications to more accurately measure the performance of a `String.equals(Object o)` operation where the object being compared is the same `String` such as those modifications described earlier in the ***Optimizing Away Dead Code*** section. But, that was not the intent of this section. Rather the purpose of this section was to illustrate the potential effects inlining can have on the observed results of a micro-benchmark. Using a HotSpot debug JVM to observe inlining decisions made by HotSpot can help avoid some of the pitfalls which may occur as a result of inlining decisions made by the HotSpot dynamic compiler. Additionally, capturing profiles with Sun Collector / Analyzer and viewing the generated assembly code can help identify whether the dynamic compiler is identifying dead code and optimizing away that dead code.

When developing a micro-benchmark to model the behavior of a larger benchmark or application, be aware of the potential impact inlining can have on the results of a micro-benchmark implementation. If a micro-benchmark does not sufficiently model a target operation of interest where methods in the micro-benchmark may or may not get inlined in a similar way as they would in a target application can lead to some misleading conclusions.

## De-optimization

Dynamic compilers are widely known for their ability to perform optimizations. But, there are situations when dynamic compilers will perform de-optimizations. For example, once a Java application starts running, methods become hot, the dynamic compiler will make optimization decisions based what it has learned from the executing program. There are cases where the optimization decisions made may turn out to be incorrect. When the dynamic compiler notices it has made an incorrect decision in a previous optimization, the dynamic compiler performs a de-optimization. A dynamic compiler de-optimization often times will, a short time later, be followed by a re-optimization once a number of execution times threshold has been reached. The mere fact that de-optimizations may occur suggests that such an event could happen while executing a benchmark or micro-benchmark. Not recognizing that a de-optimization event has occurred may result in an incorrect performance conclusion. In this section an example is provided in which initial optimizations made by a HotSpot Server dynamic compiler are followed by de-optimizations and re-optimizations.

Below is a declaration for a `Shape` interface, which has one method called `area()`. Below the `Shape` interface are class declarations for `Square`, `Rectangle` and `RightTriangle` which implement the `Shape` interface.

```
// Shape interface
public interface Shape {
    public double area();
}

// Square class
public class Square implements Shape {

    final private double side;

    public Square(double side) {
        this.side = side;
    }

    private Square(){side = 0;}

    public double area() {
        return side * side;
    }
}
```

```
// Rectangle class
public class Rectangle implements Shape {

    final private double length, width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    private Rectangle(){length = width = 0;}

    public double area() {
        return length * width;
    }
}

// RightTriangle class
public class RightTriangle implements Shape {

    final private double base, height;

    public RightTriangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    private RightTriangle(){base = height = 0;}

    public double area() {
        return .5 * base * height;
    }
}
```

Consider the following micro-benchmark implementation which has the objective to compare the time it takes to calculate the area for each of the Shapes; Square, Rectangle and RightTriangle.

```
public class Area {
    final static long ITERATIONS = 5000000000L;
    final static long NANOS_PER_MS = (1000L * 1000L);
    final static StringBuilder sb = new StringBuilder();

    private static void printStats(String s, long n,
                                   long elapsedTime){
        float millis = elapsedTime / NANOS_PER_MS;
        float rate = 0;
        if (millis != 0) {
            rate = n / millis;
        }
        System.out.println(s + ": Elapsed time in ms -> " +
millis);
        System.out.println(s + ": Iterations per ms --> " + rate);
    }
}
```

```

private static long doTest(String str, Shape s, long n) {
    double area = 0;
    long start = System.nanoTime();
    for (long i = 0; i < n; i++) {
        area = s.area();
    }
    long elapsedTime = System.nanoTime() - start;
    sb.append(str).append(area);
    System.out.println(sb.toString());
    sb.setLength(0);
    return elapsedTime;
}

public static void main(String[] args) {
    String areaStr = "    Area: ";
    Shape s = new Square(25.33);
    Shape r = new Rectangle(20.75, 30.25);
    Shape rt = new RightTriangle(20.50, 30.25);

    System.out.println("Warming up ...");
    long elapsedTime = doTest(areaStr, s, ITERATIONS);
    printStats("    Square", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, r, ITERATIONS);
    printStats("    Rectangle", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, rt, ITERATIONS);
    printStats("    Right Triangle", ITERATIONS, elapsedTime);
    System.out.println("1st warmup done.");

    System.out.println("Starting 2nd warmup ...");
    elapsedTime = doTest(areaStr, s, ITERATIONS);
    printStats("    Square", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, r, ITERATIONS);
    printStats("    Rectangle", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, rt, ITERATIONS);
    printStats("    Right Triangle", ITERATIONS, elapsedTime);
    System.out.println("2nd warmup done.");

    System.out.println("Starting measurement intervals ...");
    elapsedTime = doTest(areaStr, s, ITERATIONS);
    printStats("    Square", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, r, ITERATIONS);
    printStats("    Rectangle", ITERATIONS, elapsedTime);
    elapsedTime = doTest(areaStr, rt, ITERATIONS);
    printStats("    Right Triangle", ITERATIONS, elapsedTime);
    System.out.println("Measurement intervals done.");
}
}

```

This implementation uses two warm-up periods along with a measurement interval for calculating the area of a Square, Rectangle and RightTriangle. Both elapsed time in milliseconds and the number of iterations per millisecond are reported in each interval. As shown in the output below, executing this micro-benchmark with JDK 6 HotSpot Server JVM produces some surprising results. When comparing the elapsed time and iterations per millisecond between the first warm-up period and second warm-up period, the performance of



calculating a Square's area has decreased by about 29%. Additionally, comparing the elapsed time and iterations per millisecond between the first warm-up period and the measurement interval also shows a decrease of about 29%.

```
Warming up ...
  Area: 641.6089
  Square: Elapsed time in ms -> 11196
  Square: Iterations per ms --> 446588
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17602
  Rectangle: Iterations per ms --> 284058
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33894
  Right Triangle: Iterations per ms --> 147518
1st warmup done.
Starting 2nd warmup ...
  Area: 641.6089
  Square: Elapsed time in ms -> 15766
  Square: Iterations per ms --> 317138
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17679
  Rectangle: Iterations per ms --> 282821
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33339
  Right Triangle: Iterations per ms --> 149974
2nd warmup done.
Starting measurement intervals ...
  Area: 641.6089
  Square: Elapsed time in ms -> 15750
  Square: Iterations per ms --> 317460
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17595
  Rectangle: Iterations per ms --> 284171
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33477
  Right Triangle: Iterations per ms --> 149356
Measurement intervals done.
```

The observed decrease in performance is the result of the dynamic compiler making some initial optimization decisions which later turned out to be incorrect. For example, there are optimizations a modern dynamic compiler can make when it observes only one class implementing an interface. When executing this micro-benchmark, the dynamic compiler first performed an aggressive optimization thinking only Square implemented the Shape interface. As the micro-benchmark began calculating the area for the rectangle, the dynamic compiler had to undo the aggressive optimization it performed previously when calculating the Square's area. As a result, a de-optimization occurred and a subsequent re-optimization was performed. Being able to identify whether de-optimizations are occurring can be identified using the `-XX:+PrintCompilation` command line switch. When `-XX:+PrintCompilation` output contains the text “made not entrant”, it is an indication a previous compilation optimization is being discarded and the method will execute in the interpreter until it is executed enough times to trigger a re-compilation.

The output shown below is from executing the Area micro-benchmark with a JDK 6 HotSpot Server JVM with the `-XX:+PrintCompilation` and command line switches.

```
Warming up ...
1      com.sun.example.Square::area (10 bytes)
```

```

1%      Area::doTest @ 11 (78 bytes)
  Area: 641.6089
  Square: Elapsed time in ms -> 11196
  Square: Iterations per ms --> 446588
2      Area::doTest (78 bytes)
1% made not entrant Area::doTest @ 11 (78 bytes)
2%      Area::doTest @ 11 (78 bytes)
3      com.sun.example.Rectangle::area (10 bytes)
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17602
  Rectangle: Iterations per ms --> 284058
2 made not entrant Area::doTest (78 bytes)
4      com.sun.example.RightTriangle::area (14 bytes)
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33894
  Right Triangle: Iterations per ms --> 147518
1st warmup done.
Starting 2nd warmup ...
  Area: 641.6089
  Square: Elapsed time in ms -> 15766
  Square: Iterations per ms --> 317138
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17679
  Rectangle: Iterations per ms --> 282821
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33339
  Right Triangle: Iterations per ms --> 149974
2nd warmup done.
Starting measurement intervals ...
  Area: 641.6089
  Square: Elapsed time in ms -> 15750
  Square: Iterations per ms --> 317460
  Area: 627.6875
  Rectangle: Elapsed time in ms -> 17595
  Rectangle: Iterations per ms --> 284171
  Area: 310.0625
  Right Triangle: Elapsed time in ms -> 33477
  Right Triangle: Iterations per ms --> 149356
Measurement intervals done.

```

There are several de-optimizations which occurred during the execution of the Area micro-benchmark. The output above indicates the de-optimizations are an artifact of the three virtual call sites for Shape.area() of which Square, Rectangle and RightTriangle implement coupled with the way this micro-benchmark is written. Although de-optimizations are possible with virtual call sites, what is shown in this example is not intended to suggest to software developers to avoid writing software which utilizes interfaces and multiple classes implementing that interface. The intent is to illustrate a pitfall which can occur with the creation of micro-benchmarks and show it is difficult to know what the dynamic compiler is doing as it attempts to improve the performance of an application.

**Software developers should concentrate their efforts on good software architecture, design and implementation and not worry about trying to out smart a modern dynamic compiler. If a change to a software architecture, design or implementation is needed to overcome some artifact of a dynamic compiler, it should be considered a bug or deficiency**

**of the dynamic compiler.**

An improvement to the implementation of this micro-benchmark would be to call each `area()` method of `Square`, `Rectangle` and `RightTriangle` one right after the other in the warm up intervals rather than attempting to warm up each Shape's `area()` individually. This would allow the dynamic compiler to see all three implementations of the `Shape.area()` interface prior to making aggressive optimization decisions.

# Module 7 Demo 1: Discovering Ergonomic Selections

This exercise demonstrates the actions you can take to tell what defaults ergonomics chooses.

Slide id: 266

## Required Software

- Any OS
- JDK 1.5 or higher

## Objective

- Use the `P-XX:+PrintCommandLineFlags` command line option to tell the defaults chosen by ergonomics.

## Preparations

Verify and if required download and install Java SE 5 or later.

1. Verify if you are running Java SE 5 or later. To check enter the following command on a terminal window:

```
java -version
```

If the response is: java version 1.5.0\_00 or later you can bypass the next step and proceed to the Instructions subsection.

2. Download and install the latest version of Java SE from <http://java.sun.com/javase/downloads/index.jsp>.

Add the bin subdirectory of the Java SE version, to the OS PATH variable.

## Instructions

1. Open a command line window and enter the following command:

```
java -XX:+PrintCommandLineFlags -version
```

2. If the output returned includes `-client`, then ergonomics says the **HotSpot server JVM** is not able to be run on this machine due to resource constraints.

In that case use the following output collected from a server class machine (a two socket, quad core Intel system with 8GB of RAM) to identify the choices made by ergonomics:

```
./java -XX:+PrintCommandLineFlags -version
-XX:MaxHeapSize=1073741824 -XX:ParallelGCThreads=8
-XX:+PrintCommandLineFlags -XX:+UseParallelGC
java version "1.6.0_06-p"
Java(TM) SE Runtime Environment (build 1.6.0_06-p-b02)
Java HotSpot(TM) Server VM (build 13.0-b02, mixed mode)
```

You should identify the selections made by ergonomics for the following:

- The maximum heap size
- The garbage collector used

- The number of threads allocated for GC
3. If the output returned did not include `-client`, then use the output to identify the selections made by ergonomics.