

Java Performance Tuning and Optimization

Activity Guide

D69518GC10

Edition 1.0

June 2011

D73451

ORACLE

Copyright © 2011, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Authors

Clarence Tauro, Michael Williams

Technical Contributors and Reviewers

Charlie Hunt, Staffan Friberg

This book was published using: **Oracle Tutor**

Table of Contents

Practices for Lesson 1: Introduction.....	1-1
Practices for Lesson 1: Overview.....	1-2
Practices for Lesson 2: Java Virtual Machine and Performance Overview	2-1
Practices for Lesson 2: Overview.....	2-2
Practice 2-1: Log In to Solaris.....	2-3
Practice 2-2: Open Terminal Windows in Solaris	2-4
Practice 2-3: Open a Text File in Solaris	2-5
Practice 2-4: Start NetBeans and Open a Project.....	2-6
Practices for Lesson 3: Monitoring Operating System Performance	3-1
Practices for Lesson 3: Overview.....	3-2
Practice 3-1: Using the prstat Utility	3-3
Practices for Lesson 4: Monitoring the JVM	4-1
Practices for Lesson 4: Overview.....	4-2
Practice 4-0: Setting up VisualVM and VisualGC.....	4-3
Practice 4-1: Examining Garbage Collection Basics	4-5
Practice 4-2: Examining the Permanent Generation	4-6
Practice 4-3: Using -verbose:gc.....	4-7
Practice 4-4: Using -XX:+PrintGCDetails.....	4-9
Practice 4-5: Obtaining Application Stopped Time.....	4-11
Practice 4-6: Using jstat to Monitor GC	4-13
Practice 4-7: Monitoring a Remote Application (Optional)	4-15
Practice 4-8: Using jconsole.....	4-17
Practice 4-9: Examining VisualVM Capabilities.....	4-19
Practice 4-10: Examining VisualGC Capabilities.....	4-21
Practice 4-11: Examining JIT Compilation Activity	4-26
Practices for Lesson 5: Performance Profiling	5-1
Practices for Lesson 5: Overview.....	5-2
Practice 5-1: Application Profiling Using NetBeans Profiler	5-3
Practice 5-2: Profiling root Methods	5-6
Practice 5-3: Exploring Thread State with NetBeans Profiler	5-8
Practice 5-4: Modifying the NetBeans Profiler Session	5-9
Practice 5-5: Attaching the Profiler to Another JVM	5-10
Practice 5-6: Profiling a Web Application with NetBeans Profiler	5-11
Practice 5-7: Profiling an Application by Using Oracle Studio.....	5-13
Practice 5-8: Profiling Heap Memory with jmap and jhat.....	5-17
Practice 5-9: Profiling with NetBeans and Oracle Studio.....	5-19
Practice 5-10: Performing Memory Leak Profiling.....	5-21
Practice 5-11: Using jhat to Detect Memory Leaks	5-25
Practice 5-12: Profiling Memory Leaks with VisualVM.....	5-27
Practice 5-13: Profiling with NetBeans Profiler's HeapWalker	5-29
Practice 5-14: Finding Lock Contention.....	5-32
Practices for Lesson 6: Garbage Collection Schemes.....	6-1
Practices for Lesson 6: Overview.....	6-2
Practice 6-1: Discovering Ergonomic Selections.....	6-3

Practices for Lesson 7: Garbage Collection Tuning	7-1
Practices for Lesson 7: Overview.....	7-2
Practice 7-1: Using JVM Heap Sizing.....	7-3
Practice 7-2: Using the PrintGCStats Script	7-5
Practice 7-3: Using GCHisto.....	7-7
Practices for Lesson 8: Language-Level Concerns and Garbage Collection	8-1
Practices for Lesson 8: Overview.....	8-2
Practices for Lesson 9: Performance Tuning at the Language Level.....	9-1
Practices for Lesson 9: Overview.....	9-3
Practice 9-1: Testing Performance of String/StringBuffer/StringBuilder.....	9-4
Practice 9-2: Performance Testing of Exceptions	9-6
Practice 9-3: Performance Benchmarking of Collection Classes	9-8
Practice 9-4: Benchmarking of Primitives vs. Object Types.....	9-10
Practice 9-5: Benchmarking File Classes	9-12

Preface

Profile

Before you begin this course, you should be able to:

- Develop applications by using the Java programming language
- Implement interfaces and handle Java programming exceptions
- Use object-oriented programming techniques

How This Course Is Organized

This is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Additional Publications

- System release bulletins
- Installation and user's guides
- *Read-me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Practices for Lesson 1: Introduction

Chapter 1

Practices for Lesson 1: Overview

Practice Overview

There are no practices for this lesson.

Practices for Lesson 2: Java Virtual Machine and Performance Overview

Chapter 2

Practices for Lesson 2: Overview

Practice Overview

In these practices, you explore the systems and tools used throughout the course.

Practice 2-1: Log In to Solaris

Overview

In this practice, log in to the Solaris operating system.

Assumptions

Solaris 10 is installed on your system and it is on and functioning.

Tasks

1. At the login screen enter the following information:

User name: *student*

2. Click OK.

Password: *cangetin*

3. Click OK.

Session Type

If for some reason you need to specify the session type under options, choose:

Java Desktop System, Release 3

Root Access

Some of the utilities used in the practices require root system access. To obtain root access, enter the following in a terminal window:

su

When prompted for the password, enter:

oracle

Practice 2-2: Open Terminal Windows in Solaris

Overview

In this practice, open two terminal windows in Solaris.

Assumptions

You are logged in to Solaris and you are running a **Java Desktop System, Release 3** desktop.

Tasks

1. Click the *Launch* button.
2. From the menu, select *Applications → Utilities → Terminal*
A terminal session should start.
3. Repeat steps 1 and 2 to open another terminal window.

Practice 2-3: Open a Text File in Solaris

Overview

In this practice, open a text file using the Nautilus file manager.

Assumptions

You are logged in to Solaris and you are running a **Java Desktop System, Release 3** desktop.

Tasks

1. Open the Nautilus file manager by double-clicking the *Documents* folder on the desktop.
2. Click the *Up* arrow to move to your home directory.
3. Double-click the *labs* directory to open it.
4. Double-click the *read.txt* file.

The file is opened in the *gedit* text editor. Use Nautilus when you need to open a text file in a practice.

5. Close the text file. Keep Nautilus running.

Practice 2-4: Start NetBeans and Open a Project

Overview

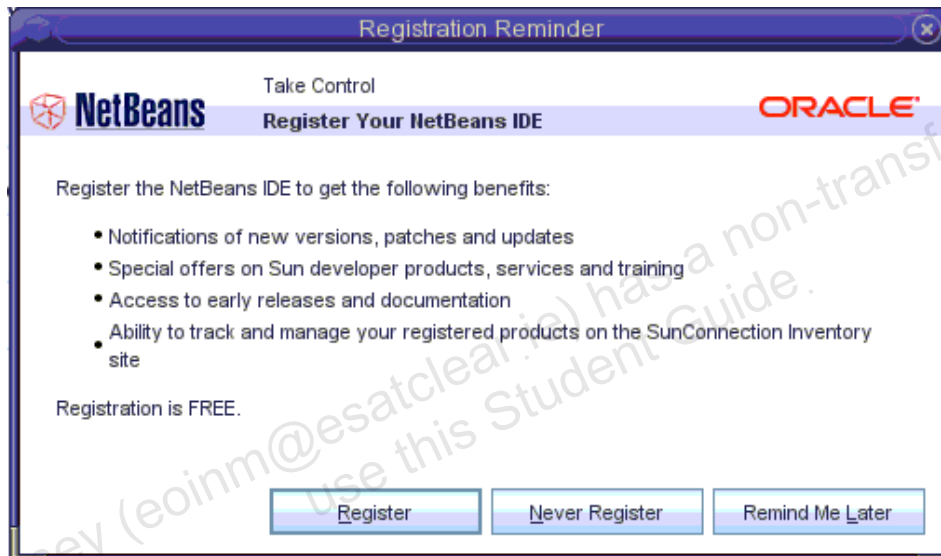
In this practice, launch NetBeans and open a NetBeans project.

Assumptions

NetBeans is installed and functioning correctly. You are logged in to Solaris and you are running a **Java Desktop System, Release 3** desktop.

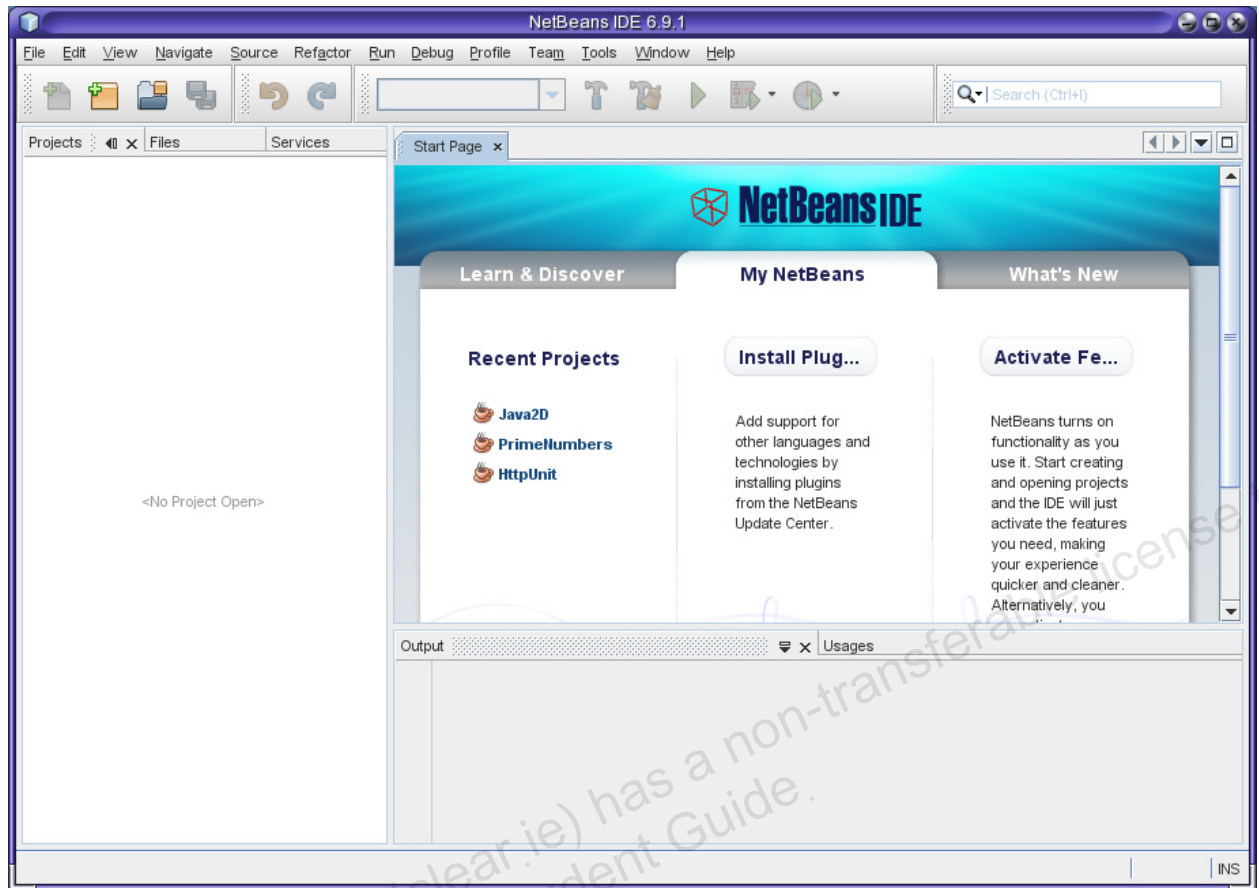
Tasks

1. Open a Terminal window.
2. At the command prompt, enter: `netbeans &`
3. The first time you run NetBeans, you will be prompted to register the product. For example:



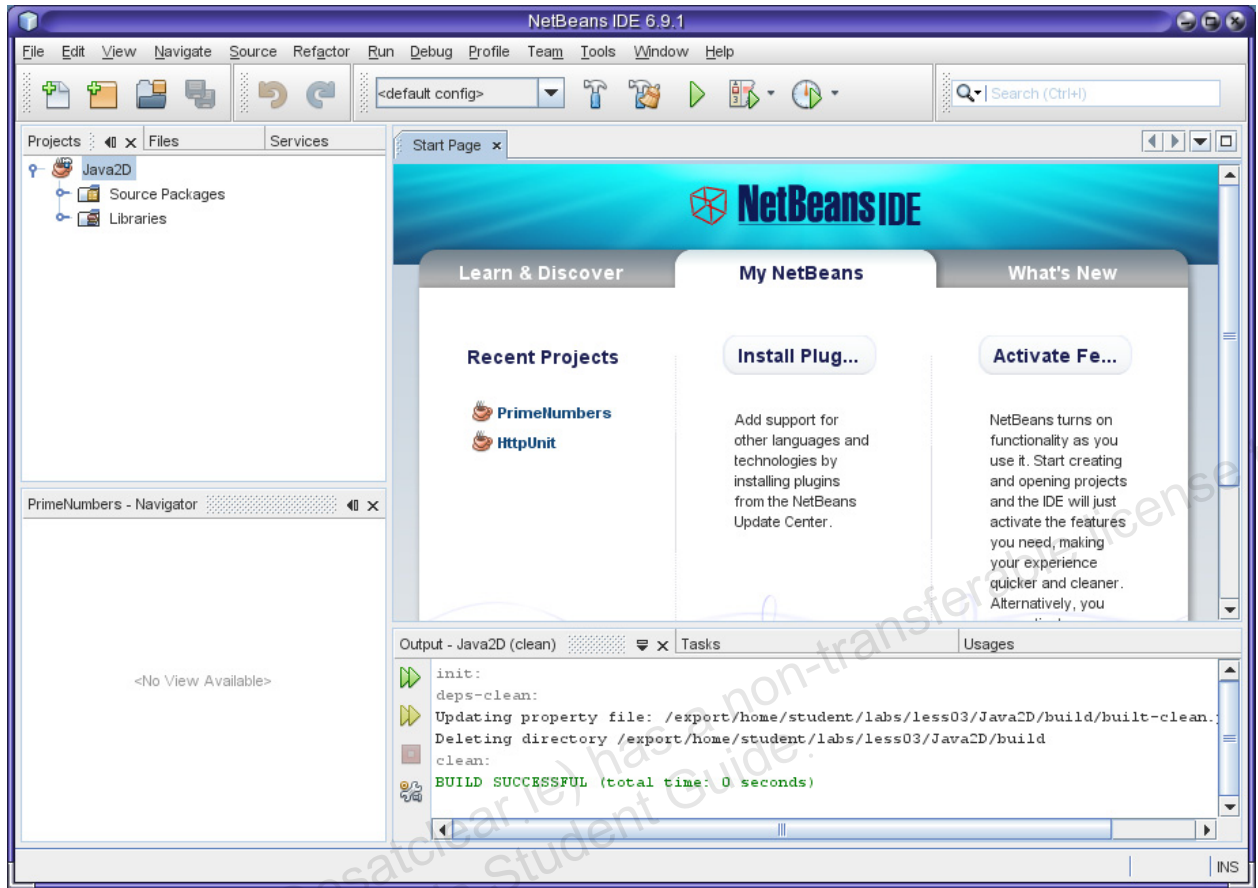
4. Just click *Never Register* and continue.

Note: The first time NetBeans runs, it caches and indexes a lot of information. So the initial load time might be a little slow. Subsequent launches of the application will be much faster. Once launched, NetBeans should look similar to this:



5. Open a sample NetBeans Project, and click *File* → *Open Project*.
6. Navigate to the `labs/less02` directory under your home directory.
7. Select the *Java2D* project. Click *Open Project*.

NetBeans should now look similar to this.



8. To run the project, right-click the project name and select *Run*.
9. Explore the user interface. Open some source files and other elements of the user interface.
10. When you are done, right-click the project name and choose *Close*.
11. Close NetBeans.

Practices for Lesson 3: Monitoring Operating System Performance

Chapter 3

Practices for Lesson 3: Overview

Practice Overview

In this practice, monitor operating system performance by using Solaris monitoring tools.

Practice 3-1: Using the `prstat` Utility

Overview

In this practice, use the `prstat` utility to identify the lightweight process (LWP) ID that has the highest CPU usage. Then map that LWP ID to a Java thread.

Assumptions

You have the Solaris OS with JDK 6 installed, including the `demo` directory for Java.

Tasks

1. Open two (Solaris OS) terminal (or `xterm`) windows.

2. In the first window, run the `Java2Demo`:

```
java -server -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar &
```

3. In the second window, run the command:

```
prstat -Lm
```

The `prstat -Lm` command will report microstate information for each LWP. You should note the following:

- The PID is reported in the left-most column. The LWP ID is reported in the right-most column.
- The LWP IDs are arranged in rows with the highest CPU utilizing LWP ID at the top.

5. Write down the PID and LWP ID of the busiest Java LWP ID.

6. Quit the `prstat` command by pressing `Q` in the `prstat` terminal window.

7. Use the following `jstack` command to dump information about running threads into a file. For example, if the PID is 957, the command would be.

```
jstack 957 > /tmp/demo1.txt
```

The `jstack` command dumps the thread information for the process identified to standard out. So the thread information is written to a text file.

8. Use the File menu to exit the `Java2Demo` application.

9. Convert the decimal value of the LWP ID to hex. For example, if the decimal value of the LWP ID identified in step 5 is 10, then the corresponding hex value is `0xa`. Make a note of this value.

Note: If you are not comfortable with hex conversions, click the *Launch* button. Then select *Applications->Accessories->Calculator*. Change the view to scientific mode and the calculator will do decimal to hex conversions.

10. Open `/tmp/demo1.txt` in the default text editor to examine the Java thread dump.

11. Search `/tmp/demo1.txt` for `nid=0xa` where “a” is the hex value of the LWP ID calculated in step 9. For example, a search for the string `nid=0xa` will find the Java thread mapping to an LWP ID of 10 in decimal.

12. If time permits, repeat the exercise.

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.

Practices for Lesson 4: Monitoring the JVM

Chapter 4

Practices for Lesson 4: Overview

Practice Overview

In these practices, you explore techniques for monitoring the JVM garbage collector and JIT compiler.

Practice 4-0: Setting up VisualVM and VisualGC

Overview

In this practice, you set up VisualVM for use in the rest of the practices in this lesson.

Assumptions

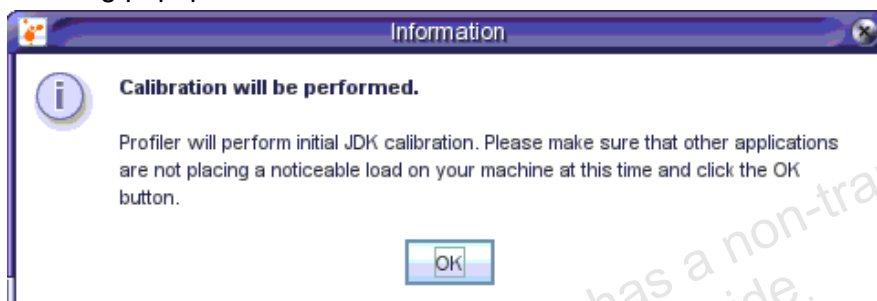
VisualVM is installed along with Java 6 and in your \$PATH.

Tasks

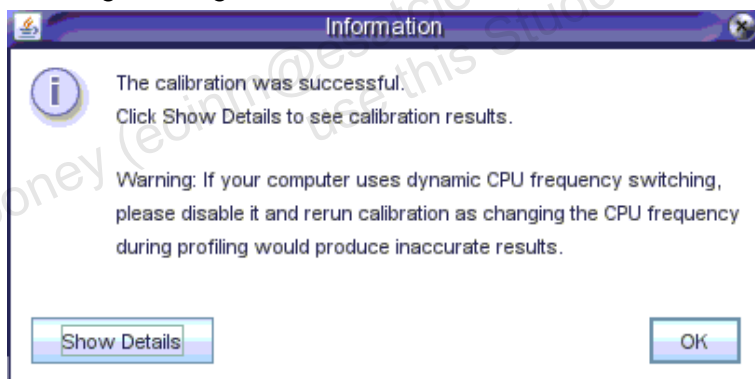
1. Launch VisualVM by entering the following command in a terminal window:

```
jvisualvm
```

Note: The first time you run VisualVM, the application calibrates itself. You will get the following popup:



VisualVM then calibrates itself for your system. Once this task is complete, you get the following message:



Your system has been calibrated and VisualVM will now start.

2. Next, install the VisualGC plugin. Click the *Tools* menu.
3. Select *Plugins*.

Note: VisualVM can be extended by using a plug-in module for any application to be monitored. You should see a list of available plug-ins. A custom plug-in can also be developed and installed by using this mechanism.

Examples include a `jconsole` plugin and a VisualGC plugin that you will install in this practice.

4. Click the *Downloaded* tab.
5. Click the *Add Plugins* button.
6. Navigate to the `$HOME/labs/visualvm-plugins/` directory.

7. Select the `com-sun-tools-visualvm-modules-visualgc.nbm` file and click *OK*. VisualGC now appears in the list of plugins and is checked.
8. Click the *Install* button. The VisualGC plugin installer get launched.
9. Click *Next*.
10. Select the *check box* to accept the license agreement.
11. Click the *Install* button. The VisualGC plugin is now installed.
12. Click *Finish*.
13. Close VisualVM. VisualVM is now ready for the remaining practices.

Practice 4-1: Examining Garbage Collection Basics

Overview

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

Assumptions

Visual VM is installed along with Java 6.

Tasks

1. Launch VisualVM by entering the following command in a terminal window:

```
jvisualvm
```

2. In a command-line window, run the Java2Demo. It is located in the `/usr/java/demo/jfc/Java2D/` directory.

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -  
XX:MaxPermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

3. Examine the VisualVM application GUI and notice that the left panel automatically detects the Java2Demo application and displays its PID. Note this PID.
4. Right-click Java2Demo application from the list of local JVMs and select Open.
5. Click the VisualGC tab (the right panel of VisualVM GUI).
6. Examine the VisualGC application now running in the right panel. In particular, observe the Java heap spaces (left subpanel of the VisualGC panel): Perm, Old, Eden, S0, and S1.
7. Click the Transforms tab on the Java2Demo and observe the Java heap spaces on the VisualGC GUI.

Note: 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 the Transforms tab, you probably observed the 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.

8. Exit and close VisualVM and Java2Demo.

Practice 4-2: Examining the Permanent Generation

Overview

This practice uses VisualVM to examine the permanent generation heap space and learn what it does and how it works.

Assumptions

VisualVM with the VisualGC plug-in and JDK 6 or later are installed.

Tasks

1. Launch VisualVM:

```
jvisualvm
```
2. In a different command-line window, run the Java2Demo.

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -  
XX:MaxPermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```
3. Right-click the Java2Demo application from the list of local JVMs and select Open.
4. Click the VisualGC tab (the right panel of the VisualVM GUI).
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. Select the Java2Demo and slowly click each of the tabs.
7. As you click 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 or raising more than others. This is expected.

Explanation

Clicking each tab causes the Java2Demo classes required to meet the requested functionality to load. As classes load into the permanent generation, notice its use or occupancy grow. If the permanent generation runs out of space, it tries to unload unused or unreferenced classes by invoking a full garbage collection. If that fails to create enough space, the JVM throws 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.

Optional: Try shrinking the `PermSize` and `MaxPermSize` parameters enough to cause an out-of-memory error.

8. When finished, exit and close VisualVM and Java2Demo.

Practice 4-3: Using `-verbose:gc`

Overview

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

Assumptions

JDK 6 or later is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -client -verbose:gc -XX:+PrintGCTimeStamps -Xmx12m -Xms3m  
-Xmn1m -XX:MaxPermSize=20m -XX:PermSize=20m -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Observe the command-line window for output after launching the applications.
3. Click a few of the Java2Demo tabs.
4. Use the GC output interpretation notes below to interpret the output.

GC Output Interpretation Notes

GC output contains lines such as this:

```
1.035: [GC 2898K->2407K(3388K), 0.0050878 secs]
```

Here is another example:

```
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 running for 1.035 seconds when the garbage collection event occurred.

Inside the brackets [] 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 generation space) being utilized at the time the GC event occurred (that is, 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 difference between the number to the left and right of the `->`, you 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 that 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.

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 that Java heap space sizing will improve the performance of the application.

Here is 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, and then repeating itself several times. Tuning Java heap spaces would likely help the performance of this application.

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 that 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 resized to a larger value.

It is possible that 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 resize permanent generation space. If it cannot resize to a larger needed space, it will throw an `OutOfMemoryError`.

It is also important to note that permanent generation space resizing/growing requires a full GC event. Therefore, it is important to monitor permanent generation space usage.

5. When finished, exit the Java2Demo program.

Practice 4-4: Using -XX:+PrintGCDetails

Overview

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

Assumptions

Java 6 or later is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -client -XX:+PrintGCDetails -Xmx12m -Xms3m -Xmn1m -  
XX:MaxPermSize=20m -XX:PermSize=20m -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Observe the command-line window for output after launching the applications.
3. 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 that the full GC was provoked by an explicit Java source code call to the `System.gc` method

The output portion `DefNew: 64575K->959K(64576K), 0.0457646 secs` indicates that 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 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.

The output portion `[Times: user=0.05, sys=0.02, real=0.03]` shows the user, system, and real CPU (usr + sys) times (used in the GC event), respectively.

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 permanent generation space statistics. When permanent generation is resized, 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 that the full GC was provoked by an explicit Java source code call to the `System.gc` method

Full GC events can include permanent generation space statistics. This happens when the permanent generation is resized.

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:

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

Notice that the young generation space was not collected; it stayed full at 8128K bytes. This activity took about 50 microseconds (0.0000505 seconds).

Note 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 sec). Again, you can see the amount of CPU time consumed after the “Times” label.

The following list contains output that includes the following labels:

- DefNew: Indicates a young generation collection. The output will also show how much was collected and how long it took.
- Tenured: Indicates an old generation collection. The output will also show how much was collected and how long it took.
- Perm: 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 watch for the following patterns:

- Frequent collections and/or increasing old generation heap sizes
- Full GC events that include the "(System)" label indicate that the application is explicitly calling `System.gc()`. In general, this is not an advisable practice.
- Permanent generation being resized could mean explicit sizing of permanent generation might be useful.
- Many or lengthy full GC events indicate 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 (and minimize the frequency of GC events in general) while maintaining acceptable pause times.

4. When finished, exit the Java2Demo program.

Practice 4-5: Obtaining Application Stopped Time

Overview

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

Assumptions

Java 6 or later is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -client -XX:+PrintGCDetails -  
XX:+PrintGCApplicationStoppedTime -XX:+UseSerialGC -Xmx12m -  
Xms3m -Xmn1m -XX:MaxPermSize=20m -XX:PermSize=20m -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. 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 that 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 that 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.

3. Shut down the Java2Demo.
4. Restart the Java2Demo using the following command line:

```
java -client -XX:+PrintGCApplicationStoppedTime -  
XX:+UseConcMarkSweepGC -XX:+PrintGCApplicationConcurrentTime -  
Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m -XX:PermSize=20m -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

5. Observe the command-line window for output after launching the applications.
6. You should observe output similar to the following:

```
Total time for which application threads were stopped: 0.0839802  
seconds Application time: 0.7187232 seconds
```

This output shows that:

- 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 can sum all the values for "stopped" and "concurrent" time to obtain an overall gc overhead for an application.

7. When finished, exit the Java2Demo program.

Practice 4-6: Using jstat to Monitor GC

Overview

This practice demonstrates the use of the `jstat` utility to monitor GC.

Assumptions

Java 6 or later is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m -
XX:PermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Click the Transforms tab on the Java2Demo GUI.

3. Obtain the local virtual machine ID (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 (2874 in this example) 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.

4. 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
```

5. 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 that 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 third and fourth sample, 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 fourth, fifth, and sixth reporting intervals. These increases are likely the result of the classloading activity.

6. When finished, exit the Java2Demo program.

Practice 4-7: Monitoring a Remote Application (Optional)

Overview

To monitor remote HotSpot VMs, the `jstatd` daemon is required to be installed and configured on the system where the remote HotSpot VM is running. The `jstatd` daemon launches a Java RMI server application that monitors the creation and termination of HotSpot VMs and provides an interface to enable 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 the 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 that monitored JVMs are not compromised. The policy file used by `jstatd` must conform to Java's policy specification.

Note: This practice requires two machines. Therefore, if you do not have two machines or another student you can work with, feel free to skip this practice.

Assumptions

Java 6 or later is installed. A remote machine and a monitoring machine are available for testing.

Tasks for Server/Hosting Machine

Perform the following steps on the remote machine.

1. From the Solaris menu select *Launch* → *Accessories* → *Text Editor*. The Gnome text editor is launched.
2. Create a new file buffer, save the file in your home directory (`/export/home/student`), and name it `jstatd.policy`.
3. Enter the following policy information into the file:

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

Note: The example policy file allows `jstatd` to run without any security exceptions. This policy is less liberal than granting all permissions to all codebases, but it 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 provides. However, if security concerns cannot be addressed with a policy file, the safest approach is not to run `jstatd` and instead use the monitoring tools locally rather than connecting remotely.

Additional details about how to configure `jstatd` can be found on the following page:

<http://download.oracle.com/javase/6/docs/technotes/tools/share/jstatd.html>

4. Save the file.
5. Exit the text editor.
6. 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
```

7. Run the Java2Demo with the following command-line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m -  
XX:PermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

8. Click the Transforms tab on the Java2Demo GUI.
9. Obtain the LVMID of the Java2Demo process by using the following command on another command window.

```
jps -l
```

Tasks for the Monitoring Machine

Perform the following steps on the monitoring machine:

10. 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@remotehost 250
```

11. Observe the command-line window for the output of the `jstat` utility.
12. When finished, exit the Java2Demo program.

Practice 4-8: Using jconsole

Overview

This practice introduces the `jconsole` utility and demonstrates its monitoring capabilities. `jconsole` is a JMX (Java Management Extensions)-compliant GUI tool that can connect to a running JVM.

Assumptions

Java 6 or later is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -Dcom.sun.management.jmxremote -jar
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Start `jconsole` by entering the following command in a terminal window.

```
jconsole
```

When `jconsole` is launched it automatically discovers running Java applications and presents a “connect to” dialog box.

3. Use the Connect button in the dialog box to navigate to the New Connection dialog box and select the Java2Demo application.
4. When `jconsole` connects (to an application), it loads a set of six tabs. Use the Memory tab to observe heap memory spaces.
5. When finished, exit the Java2Demo program and `jconsole`.

Optional

The next set of steps demonstrates how `jconsole` can remotely monitor another JVM. In addition, you can connect multiple machines running `jconsole` to a single remote JVM. To perform this lab you must have machines that connect to each other over a network. Please consult with your instructor on which machines to connect to for your particular class.

6. To determine the IP address of your machine in Solaris, first open a terminal window.
7. Use the `su` command to become a superuser.
8. Enter `ifconfig -a`.
9. This command should indicate the IP address for your machine. Share this address with your fellow students or with your instructor as directed by your instructor.
10. Run the Java2Demo with the following command-line switches:

```
java -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false -jar
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

There are three new switches that merit explanation.

`-Dcom.sun.management.jmxremote.port=8888` – This switch sets the connection port for the `jconsole` client. For this lab everyone will connect to port 8888.

`-Dcom.sun.management.jmxremote.authenticate=false` - This switch turns off authentication. Typically, you should create a password file and define roles and passwords in the file to allow connections to a remote machine. We have skipped these steps in the interest of time.

-Dcom.sun.management.jmxremote.ssl=false - This switch turns off SSL. In a production environment you would want to require passwords and use SSL. Once again, we are disabling this feature in the interest of saving time.

11. Start `jconsole` by entering the following command in a terminal window:
`jconsole`
12. In the connection dialog, select *Remote Process*.
13. Enter the IP address and port number of a machine you wish to connect to.
For example: 192.168.1.2:8888
14. Click *Connect*.
15. This should connect you to the remote machine and begin a monitoring session.
16. From the `jconsole` menu, select *Connection* → *New Connection*. You can now select another machine to begin a new monitoring session.
17. Repeat these steps and connect to multiple machines as directed by your instructor.
18. When finished, exit the Java2Demo program and `jconsole`.

JConsole Documentation:

<http://download.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html>

Practice 4-9: Examining VisualVM Capabilities

Overview

This practice demonstrates the monitoring capabilities of the VisualVM utility.

Assumptions

VisualVM and JDK 6 or later are installed.

Tasks

1. In a command-line window, run the Java2Demo.

```
java -client -Xmx16m -Xms3m -Xmn1m -XX:MaxPermSize=20m -  
XX:PermSize=20m -Dcom.sun.management.jmxremote -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Launch VisualVM:

```
jvisualvm
```

3. Examine the Start Page tab. This tab appears the first time you launch VisualVM. It contains links to a set of user guides.
4. Examine the Applications panel, which contains three nodes: Local, Remote, and Snapshot.

Note: If for some reason the Applications panel does not appear by default, select Windows → Reset Windows. This resets the interface and restores the Applications panel.

Local: The Local node shows auto-discovered Java processes that 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 requires either a specific JMX connection or the `jstatd` daemon to be running on the remote machine. If the `jstatd` daemon is running on the remote machine, Java processes that can be monitored are automatically discovered and displayed under the remote hosts.

Snapshot: The Snapshot node enables profiler snapshots to be saved and reopened later.

5. Right-click the Java2Demo.

You will see that there are options to capture a thread dump, heap dump, or profile. There are also options to take an application snapshot, enable a heap dump on an out-of-memory error, and 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.

Notice an overall panel for the Java2Demo application open on the right containing four subpanels. The number of subpanels displayed for a monitored application varies depending on the version of the JVM running the application and on whether the application is remote or local (see the feature matrix on <https://visualvm.dev.java.net>).

When Java2Demo is run as a local application on a Java 6 JVM, you should see six subpanels: Overview, Monitor, Threads, Sampler, Profiler, and VisualGC.

7. Open and examine each (VisualVM) subpanel of the Java2Demo application.

Overview tab: Provides overview information about the running application and JVM.

Monitor tab: Provides graphs showing summary information about the JVM. The information includes:

- CPU utilization

- Heap memory utilization
- Number of classes
- Thread count

Threads tab: Provides detailed information about the Java threads in the application

Sampler tab: This tab looks and acts very similar to the profiling tab. However, there is a big difference. The Sampler tab gets its information by polling application thread dumps and memory histograms. Therefore, no instrumentation is added to the application that has a minimal impact on performance. This tab is a very quick and nonintrusive way of getting CPU and memory information.

Profiler tab: Provides the ability to do either CPU profiling or heap (memory) profiling. By default, the profiler is not running. When you select to CPU or Memory profile, VisualVM adds instrumentation code to the application which affects the performance of the application a small amount or a great deal depending upon the number of methods profiled. The subject is explored in more detail in the Profiling lesson.

Any saved profiled snapshots will be placed under the Snapshots node in the left panel.

8. Exit the Java2Demo and the VisualVM application.

Practice 4-10: Examining VisualGC Capabilities

Overview

This practice demonstrates the monitoring capabilities of the VisualGC plug-in.

Assumptions

VisualVM and the VisualGC plug-in are installed. Java 6 or later is installed.

Tasks

1. Run the Java2Demo:

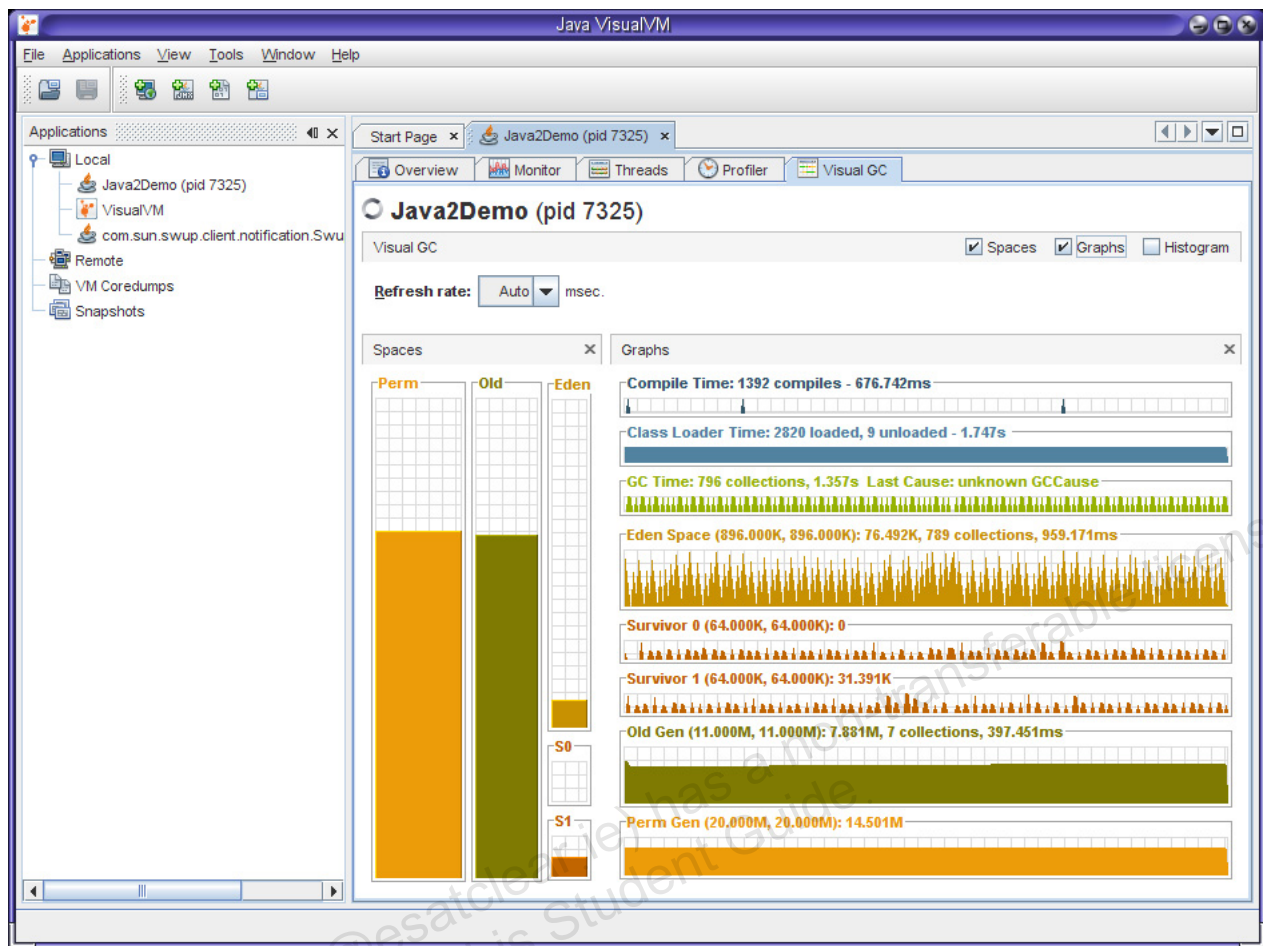
```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m -  
XX:PermSize=20m -Dcom.sun.management.jmxremote -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Click the Transforms tab on the Java2Demo application to increase object allocation. This increases the frequency of full GC events.
3. Launch VisualVM:

```
jvisualvm
```
4. Right-click the Java2Demo in the Applications panel and select Open.
5. Observe the VisualGC panel.
6. (Optional) Try changing values in the Java2Demo application to see how it affects the graphs in VisualVM. If you click the Transforms tab in the application, on the lower right part of the application you can change the number of shapes, strings, or images displayed in the preview area to the left. Change these values and observe the impact in VisualGC.

Overview of the VisualGC Display

VisualGC displays two or three sections in the right panel depending on the garbage collector being used by the JVM. When the throughput collector is used, VisualGC displays only two sections: the VisualGC window and the Graph window. Otherwise, when the serial collector or concurrent collector is being used, the Spaces, Graphs, and Histogram sections are also displayed. The following snapshot shows the Spaces and Graphs sections but hides the histogram section.



Spaces: The Spaces section provides a graphical view of the garbage collection spaces. This panel is divided into three vertical sections, one for each of the spaces: perm (Permanent) space, old (or tenured) space, and the young generation section. The young generation section consists 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. Each space bar-graph indicates the current utilization of the space relative to its maximum capacity.

Young Generation: The relationship between the sizes of the spaces in the young generation portion of the VisualGC frame is 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 the 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 survivor spaces may not be identical and the space in the 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 updates accordingly.

Graphs: This section plots performance statistics as a function of time for a historical view. This section displays garbage collection statistics along with dynamic (JIT) compiler and class loader

statistics. 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.

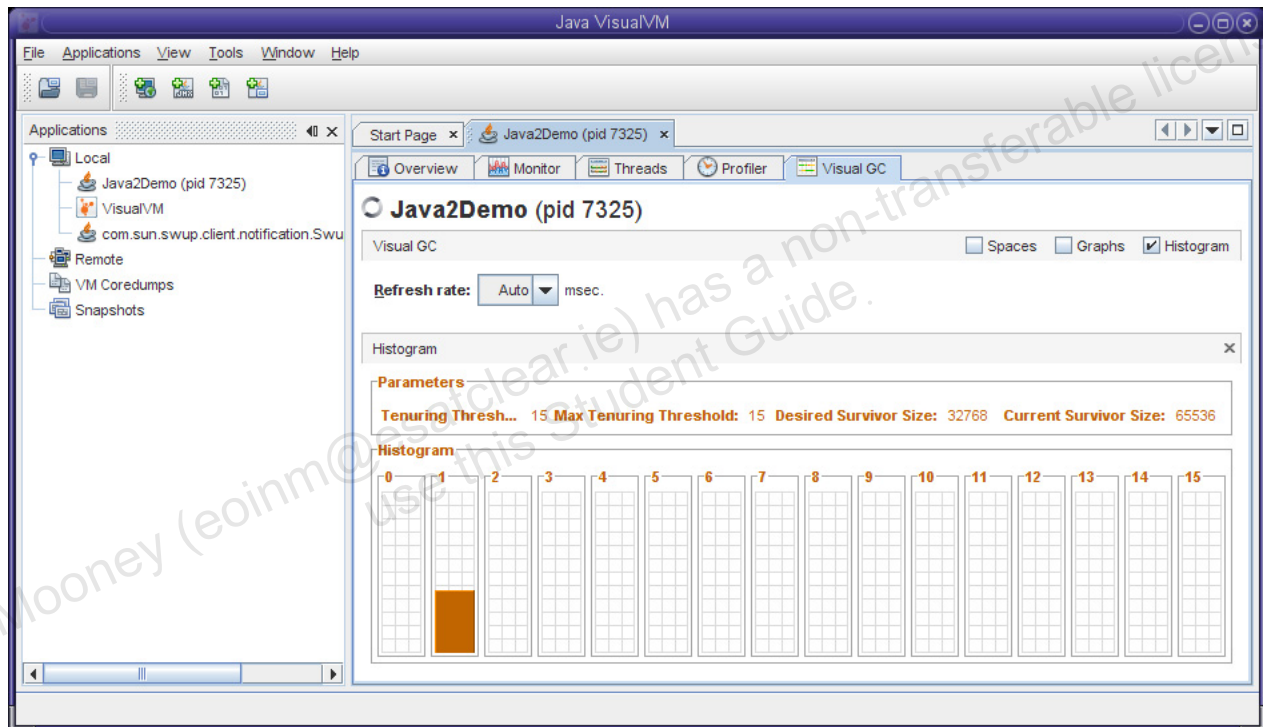
The Graphs section has the following displays:

- **Compile Time:** This graph plots the amount of time spent compiling Java bytecode into native code. The height of this display is not scaled to any particular value. A nonzero value in this graph indicates that compilation 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 number of compilation tasks and the accumulated compilation time since the start of the application.
- **Class Loader Time:** This graph plots the amount of time spent in class loading and unloading activities. The height of this display is not scaled to a particular value. A nonzero value in this graph indicates that class loading 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 number of classes loaded and unloaded and the accumulated class loading time since the start of the application.
- **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 nonzero 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 the 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 parentheses) 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:** Display the utilization of the two survivor spaces over time. The survivor spaces are the remaining two spaces in 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 the space shrinks and grows over time. The title bar displays the name of the space and its maximum and current capacity (in parentheses) 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 parentheses) 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 parentheses) 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 any of these spaces in the Graph window and selecting the Show Reserved Space check box. The default is to show committed memory utilization and can be switched back by right-clicking and deselecting the Show Reserved Space check box. In reserved mode, the data is scaled according to the maximum capacity of the space; in committed mode, the data is scaled according to the current capacity of the space. In reserved mode, the background grid is colored dark gray to represent that portion of the reserved memory that is uncommitted memory and colored green to represent the portion that is committed.

Histogram: The Survivor Age Histogram section, shown below, is displayed below the Spaces and Graphs sections.



Parameters: This panel displays the size of the survivor spaces and the parameters that control the promotion behavior of the young generation. After each young generation collection, objects that survive the collection but are not promoted remain in the survivor spaces. An object's age is incremented each time it survives a young generation collection until the object's age reaches the maximum age as defined by the `TenuringThreshold` variable, which varies between 1 and `MaxTenuringThreshold - 1`, depending on the utilization of the survivor space. If the survivor space overflows, the oldest objects are promoted to the old generation until the utilization of the space does not exceed `DesiredSurvivorSize`. A

`MaxTenuringThreshold` value of 0 results in objects always being promoted to the old generation, and a value of 32 prevents objects from being promoted unless the survivor space fills up.

Histogram: This panel displays a snapshot of the age distribution of objects in the active survivor space after the last young generation collection. The display comprises 32 identically sized regions, one for each possible object age. Each region represents 100% of the active

survivor space and is filled with a colored area that indicates the percentage of the survivor space occupied by objects of the given age.

When the target JVM is started with the parallel young generation collector (-XX:+UseParallelGC), the Survivor Age Histogram panel is not displayed. The parallel young generation collector does not maintain a survivor age histogram because it applies a different policy for maintaining objects in the survivor spaces.

7. When finished, exit the VisualVM program.

Practice 4-11: Examining JIT Compilation Activity

Overview

This practice uses various tools to examine JIT compilation.

Assumptions

VisualVM and the VisualGC plug-in are installed. JDK 6 or later is installed.

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. Therefore, it is useful to observe dynamic compiler behavior. Monitoring dynamic compiler activity is also useful when you want to identify methods that are being optimized or in some cases deoptimized and reoptimized. A method can be deoptimized and reoptimized when the dynamic compiler has made some initial assumptions in an optimization that later turned out to be incorrect. To address this scenario, the HotSpot dynamic compiler discards the previous optimization and reoptimizes the method based on the new information it has obtained.

Tasks

1. Run the Java2Demo.

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m -XX:PermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```
2. Start `jconsole` by entering the following command in a command-line window:

```
jconsole
```

When `jconsole` is launched, it automatically discovers running Java applications and presents a “connect to” dialog box.
3. Connect to the Java2Demo application.
4. Click 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.
5. Examine the attributes accessible through MBean instances on the MBeans tab, under the `java.lang.Compilation.Attributes` node.
6. Exit `jconsole`.
7. Launch VisualVM

```
jvisualvm
```
8. Open the Java2Demo list item.
9. Click the VisualGC tab and select the Java2Demo as the application to monitor.
10. When you watch the Compile Time panel on the right, you observe a spike or blip associated with each time that a compilation activity takes place.
11. Go to the Java2Demo GUI and slowly click each tab. Observe what occurs in the Compile Time panel of VisualGC. As optimizations are performed by the JVM's JIT compiler, you observe blips or spikes in VisualGC's Compile Time panel. The Compile Time panel also displays the number of compilations and the time spent doing the optimizations.
12. Quit the VisualVM utility.

13. Run the `jps` command (with the `-l` option) from a command-line window to obtain the LVMID of the executing Java2Demo application:

```
jps -l
```

14. Use the LVMID obtained in the previous step to run the `jstat` command as follows:

```
jstat -printcompilation <LVMID> 500
```

The `jstat` utility prints the last JIT compiled method in the 500 millisecond duration window. This information is of limited interest because it basically indicates merely that the JIT compiler did something.

15. Quit `jstat` and shut down the Java2Demo application.
16. Restart the Java2Demo with the `-XX:+PrintCompilation` flag:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=20m  
-XX:PermSize=20m -XX:+PrintCompilation -jar  
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

17. The `-XX:+PrintCompilation` option prints output for every JIT compiled method.
18. When finished, exit the Java2Demo program.

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.

Practices for Lesson 5: Performance Profiling

Chapter 5

Practices for Lesson 5: Overview

Practices Overview

In these practices, you use a number of profiling tools to profile Java applications.

Practice 5-1: Application Profiling Using NetBeans Profiler

Overview

In this practice, you profile an application by using NetBeans Profiler.

Assumptions

NetBeans is installed along with the NetBeans profiler.

Tasks

1. Open a terminal window. Start NetBeans by entering: `netbeans &`
2. Open the “Java2D” project located in `$home/labs/less05`.
3. Compile and run the project.
 - a. Experiment with the application a bit. Click the various tabs.
 - b. Run the project once to help cache the application for profiling.
4. Stop the application.

Note: Profiling an entire application is a problem for most profilers. It is an intrusive operation and it complicates the analysis with a large amount of additional data to scrutinize. NetBeans Profiler attempts to solve this problem by allowing you to profile only a part of your application.
5. To start, profile the entire application. Right-click the Java2D project and select **Profile**.
6. Click **CPU**.
7. Select the **Entire Application** option to profile the entire application.

Note: You can choose to profile the entire application or part of it. You can also use filters to exclude parts of the application. For example, a commonly used filter is “Exclude Java core classes.”
8. For a Filter, select **Profile All Classes**.

Note: This shows the impact of profiling all methods of all classes. This is everything in the application itself and all the libraries that it uses.
9. Click the **Run** button. A couple of dialog boxes pass by for the profiler. Eventually the application's progress meter appears.

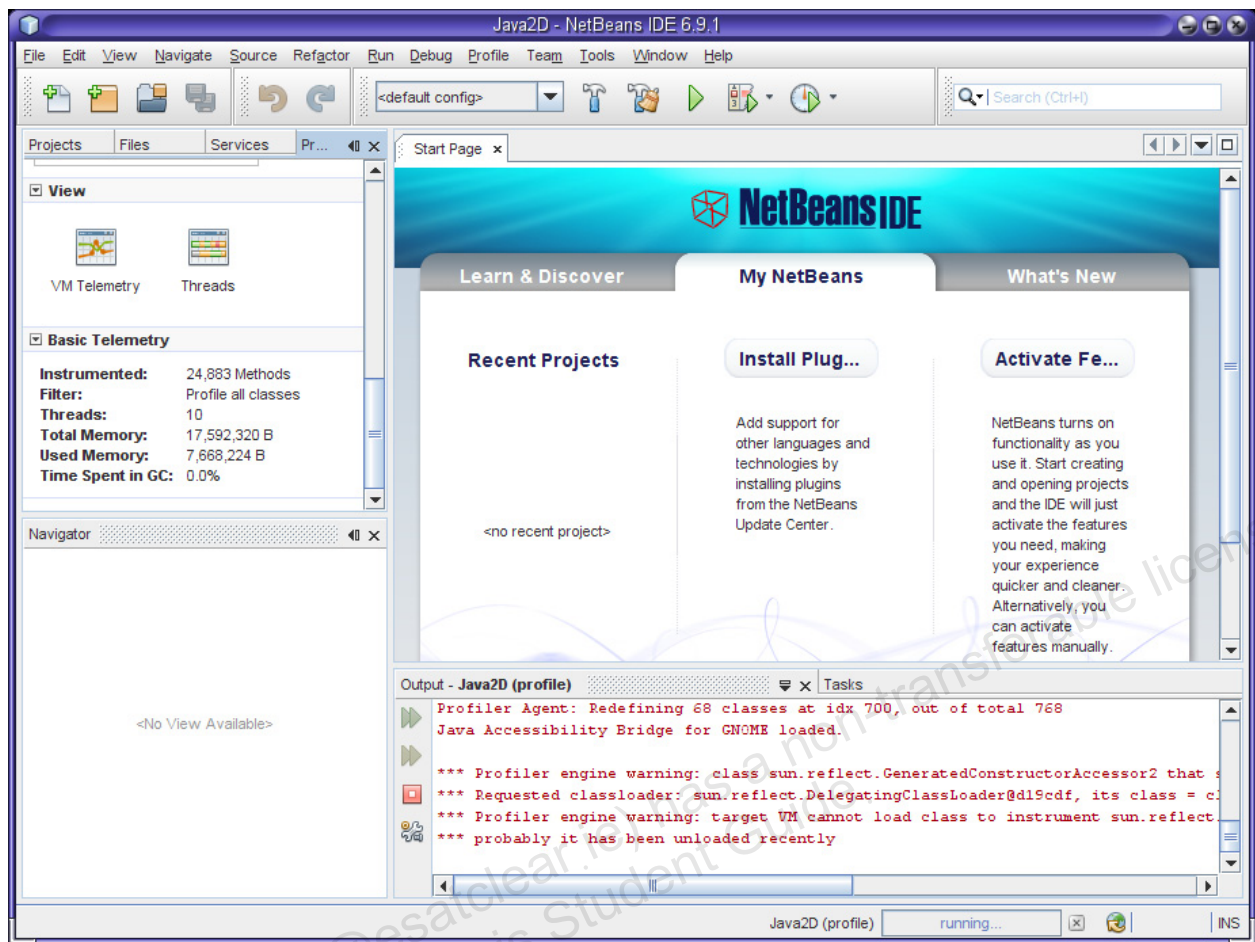
Note: Notice the difference in how long the wait is for the progress meter to complete.
10. Click a few of the tabs again.
11. Click the **Colors** tab last. The animation will usually lag a bit initially and then smoothe out.

Note: Notice the impact on performance. The application took much longer to start and did not respond to mouse clicks as quickly.

The frames per second (FPS) 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, point this out.
12. Switch back to the IDE. Scroll the Profile window until the **Basic Telemetry** information is visible.

Note: 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 10 times larger or more.

Basic Telemetry



13. Under the Profiling Results section of the **Profiler Tab**, click the **Live Results**.

14. Scroll through the section to show how many rows it has.

Note: The other problem with profiling everything is that you have to peer at too much information to find the cause of the performance problem.

NetBeans Live Results Methods

The screenshot shows the NetBeans IDE 6.9.1 interface. The 'Live Profiling Results' window is open, displaying a table of Hot Spots. The table has four columns: Method, Self time [%], Self time, and Invocations. The methods listed include various rendering and utility methods from the sun.java2d package and other Java classes. The bottom pane shows the Output window with messages from the Profiler Agent and warnings about class loading.

Hot Spots - Method	Self time [%]	Self time	Invocations
sun.java2d.pipe.DuctusShapeRenderer. renderTiles ...		40858 ms (5.1%)	207530
sun.java2d.pipe.DuctusRenderer. createShapeRaster ...		37690 ms (4.7%)	201865
sun.java2d.pipe.AlphaColorPipe. renderPathTile (O...		23370 ms (2.9%)	863602
sun.awt.SunToolkit. isInstanceOf (Class, String)		20618 ms (2.6%)	692218
java.util.Arrays. binarySearch0 (Object[], int, int, Obj...		19205 ms (2.4%)	387708
sun.dc.pr.PathFiller. writeAlpha (byte[], int, int, int)		17370 ms (2.2%)	837935
sun.java2d.loops.GraphicsPrimitiveMgr. \$2.compare ...		17348 ms (2.2%)	3518094
java.lang.Math. getExponent (double)		12702 ms (1.6%)	1209879
java.lang.StrictMath. floor (double)		12238 ms (1.5%)	1206843
java.lang.StrictMath. floorOrCeil (double, double, do...		11962 ms (1.5%)	1209879
java.lang.Math. floor (double)		11843 ms (1.5%)	1206843
java.lang.Math. min (int, int)		10465 ms (1.3%)	2180653
sun.awt.SunToolkit. awtLock ()		10215 ms (1.3%)	908886
sun.java2d.pipe.AlphaColorPipe. fillParallelogram (...)		10173 ms (1.3%)	10341

Output - Java2D (profile) Tasks

```

Profiler Agent: Redefining 68 classes at idx 700, out of total 768
Java Accessibility Bridge for GNOME loaded.

*** Profiler engine warning: class sun.reflect.GeneratedConstructorAccessor2 that s
*** Requested classloader: sun.reflect.DelegatingClassLoader@d19cdf, its class = c
*** Profiler engine warning: target VM cannot load class to instrument sun.reflect
*** probably it has been unloaded recently
  
```

15. In the Profile tab, click the **Stop** button.
16. In the Projects tab, right-click the Java2D project and select **Profile**.
17. Profile the entire application, but this time use the “Exclude Java core classes” filter. Click **Run**.

Note: This is a more common scenario, applying some kind of filter.
18. In the Profile window, click the **Live Results** button.
19. Compare the output to what was shown before. Notice that the core Java classes are not shown.

Note: This filter capability helps in two ways. There are fewer classes to scrutinize and the profiling is less intrusive on the application.
20. In the Profile tab, click the **Stop** button.

Practice 5-2: Profiling `root` Methods

Overview

The last practice demonstrated that some profiling tools force you to define the filters of the package and/or class names for what should or should not be instrumented by the profiler. The NetBeans Profiler supports filters, but it also has a more powerful feature: `root` methods.

Assumptions

Continue to work with NetBeans and the same project from the last practice.

Tasks

1. In the Projects window, expand the source package for: `java2d.demos.Colors`
2. Open the source file `Rotator3D.java` in the NetBeans editor.
3. In the editor window, right-click the `render()` method and select **Profiling > Add as Profiling Root Method**.
4. Add the `render()` method to the **Analyze Performance** configuration.
5. Do the same thing for the `step()` method.
6. In the Projects tab, right-click the Java2D project and select **Profile**.
7. In the Profile Java2D/Analyze Performance dialog box, click **Part of Application**.
8. Select **Profile only project classes** as the filter.
9. Click the **Run** button. Startup should be much faster.

Note: When only a part of the application is profiled, startup is much faster, closer to the performance with no profiling.

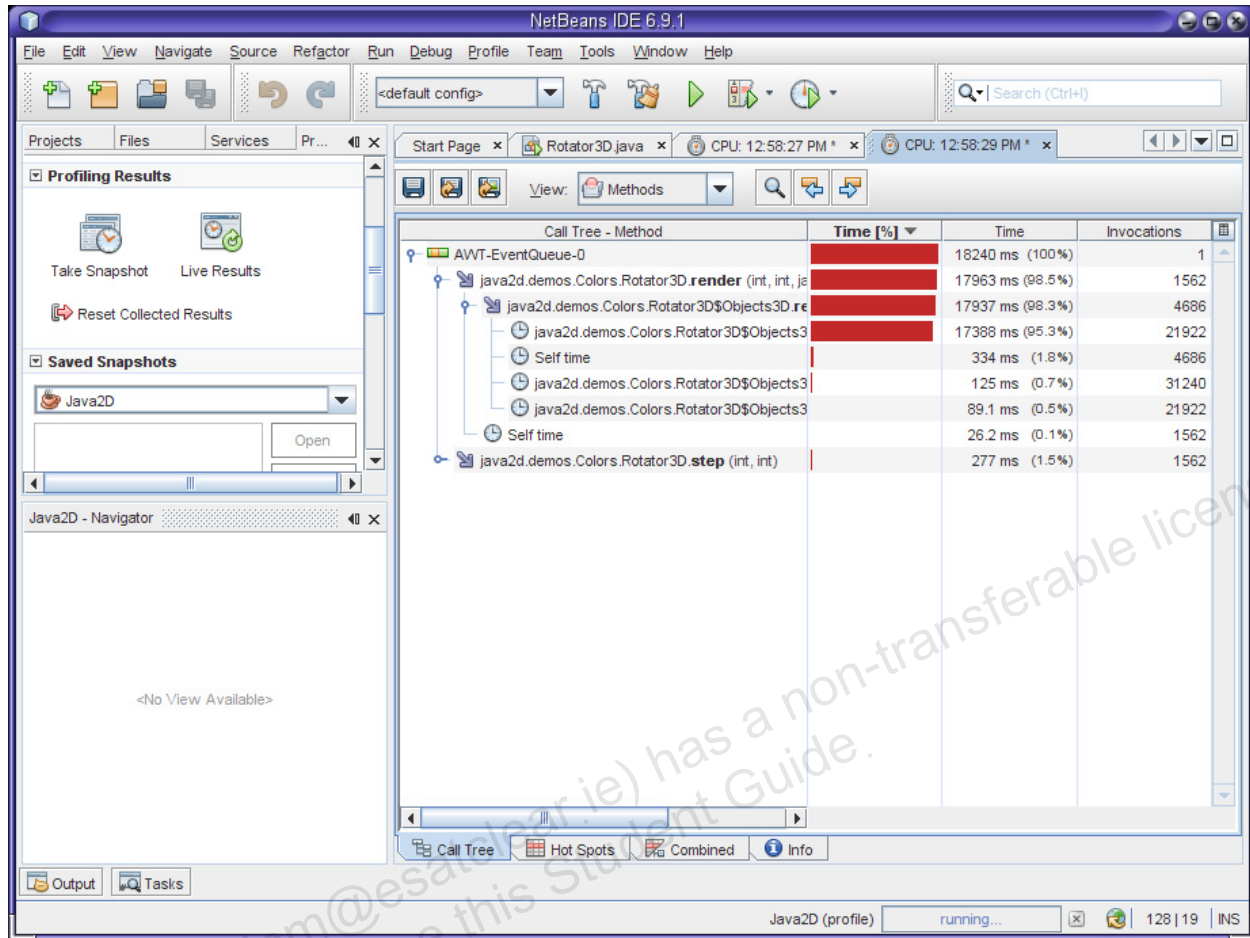
10. Click some tabs.
11. Click the **Colors** tab last.
12. Switch back to the IDE.
13. Scroll the Profile window until the **Basic Telemetry** information is visible.

Note: Notice that the number of methods being profiled is much smaller. This is because the profiler examined the application, starting at the `root` methods. It instrumented those root methods and then examined the methods that they call and instrumented them. Then it examined the methods that those methods call, and so on. The profiler does this until it has figured out the call graph that originates at the `root` methods. Those methods are the only ones to be instrumented. Everything else runs at full speed.

14. In the Profile tab, click the **Take Snapshot** button.

Note: With less instrumentation, there is less data to peer at. The `render()` method is taking up much more time than `step()`.

NetBeans Root Method Profiling



15. Stop the profiling session.
16. Go back to the editor window where the `Rotator3D.java` source file is displayed.
17. Right-click the `render()` method and select **Profiling → Insert Profiling Point**.
18. Select each one and review the action it takes in the Description dialog box.
Note: There are use cases where each of these options may be useful.
19. Click *Cancel* to close the dialog.

Practice 5-3: Exploring Thread State with NetBeans Profiler

Overview

Examine the state of threads by using NetBeans Profiler.

Assumptions

Continue to work with NetBeans and the same project from the last practice.

Tasks

1. Restart the Profiling session from the main menu by selecting **Profile → Rerun Profiling Session**.
2. Go to the Profile tab in the upper-left panel.
3. Scroll down to find the **View** subpanel.
4. Click the **Threads** icon.
5. Enable threads monitoring on the right panel that is displayed to enable collection of thread information.
Note: The thread state is easily observed with the state information color-coded.
6. Select and double-click a thread, such as the `AWT-EventQueue` thread.
Note: You can drill into a specific thread to see a breakdown of how much time it has spent in each state.
7. Click the **Details** tab in the thread window to switch to the text view.
Note: Notice the state changes for a thread.

Practice 5-4: Modifying the NetBeans Profiler Session

Overview

With NetBeans Profiler, you can modify your profiling session without disturbing normal operations.

Assumptions

Continue to work with NetBeans and the same project from the last practice.

Tasks

1. From the menu, select **Profile → Modify Profiling Session**.

2. Click the **Monitor** button on the left and then click **OK**.

Note: NetBeans Profiler can change the profiler settings as directed, without having to restart the application. ***By switching to a monitor-only mode, there is no instrumentation at all now. The entire application is running at full speed.***

3. From the menu, select **Profile > Modify Profiling**. Notice that by clicking on any of the three buttons on the left, you can modify the profiling session.

4. To show heap profiling, click the **Memory** button and then click **OK**.

Note: In addition to reporting CPU performance, the profiler reports detailed monitoring of your application's memory usage. This is especially helpful for tracking down memory leaks. It also helps identify areas of excess object allocations. Excess object allocations can be an issue when frequent garbage collection events occur or when trying to reduce the frequency of garbage collections.

The profile settings can be fine-tuned by defining a custom configuration, filters, and so on.

5. Scroll down to the View section of the Profiling tab and click **VM Telemetry**.

Note: You see a real-time graph of the Used Heap.

6. Select **Profile → Stop Profiling Session**.

Practice 5-5: Attaching the Profiler to Another JVM

Overview

Review the steps to attach NetBeans Profiler to another JVM.

Assumptions

Continue to work with NetBeans and the same project from the last practice.

Tasks

1. From the menu, select **Profile → Attach Profiler**. The Attach Profiler task dialog box opens.

Note: NetBeans Profiler also supports attaching to any JVM. It shows you the JVM command-line flags that are needed and helps edit a startup script for you to use.

2. Select **CPU** and **Entire Application**.
3. Select **Exclude Java core classes**.
4. Click the **Attach** button.
5. For Target Type, select **Application**.
6. Take the defaults for the other settings and select **Next**.
7. Click **Next**.

Note: You can specify my application type, server, JDK version, and so on. The IDE will offer to edit the application's startup script or show the changes that need to be made.

8. Click the **Cancel** button.
9. Go back and explore some other options and see how the command-line options have changed.

Note: If you had actually started a JVM with these settings, the JVM would see the command-line flags and wait before proceeding. This is the same way that debuggers attach to a JVM. Once the JVM is waiting, click the **Attach** button and start profiling.

10. This completes our work with the Java2D project. Go ahead and close the project.

Practice 5-6 Profiling a Web Application with NetBeans Profiler

Overview

Profile a web application by using NetBeans Profiler.

Assumptions

NetBeans is installed with a functioning GlassFish application server.

Tasks

Start the GlassFish Application Server

The sample application used in the practice will be deployed to the GlassFish Application Server.

1. From the NetBeans main screen, click the **Services** tab.
2. Expand the **Servers** folder.
3. Right-click the **GlassFish** icon. Select **Start**.
This should start the GlassFish server.

Configure HTTP Monitor

Next configure the GlassFish Server to run the HTTP Monitor Tool.

4. Right-click the **GlassFish** icon. Select **Properties**.
5. Select the **Enable HTTP Monitor** option.
6. Click **OK** to dismiss the warning message.
7. Click **Close** to close the Properties dialog.

Create a Sample Java EE Web Project

Now create a sample Java Web project.

8. Select **File > New Project** from the main menu.
9. Expand the **Samples** category and select **Java Web**.
10. Select the **Lean Service ECB Component with Ajax** project and click **Next**.
11. Change the project name to **LeanService**, and then click **Finish**.
The **LeanService** project is now saved in the `$HOME/NetBeansProject` directory.
12. Right-click the **LeanService** project and select **Deploy**. It should take a few seconds for the application to deploy to GlassFish. You are now ready to profile the application.
13. Right-click the **LeanService** project and select **Profile**.
14. Click **OK** in the dialog that asks which platform to run on. The default of Java 1.6 is selected.
15. Click **OK** in the dialog box that asks for permission to instrument the project.
16. The Profile LeanService dialog box is displayed.
17. Click the **CPU** button on the left if it is not already selected.
18. Select the **Entire Application** option.
19. Choose to filter to profile only project classes.

Note: It is acceptable if "Use defined Profiling Points" is enabled. There are points defined at the moment. But, they could be defined by loading the Project files in the NetBeans editor.

20. When you run the application in the next step, a deployment error will be generated. Ignore the deployment error that is generated. It is caused because there is no registration account associated with this server. You must have Internet access to create a registration account.
21. Click the **Run** button.
Note: It may take about 30–60 seconds to start and deploy the application to the GlassFish application server. Once it is deployed, the web browser will automatically display the deployed Tomcat Servlet Examples.
22. If your browser is not automatically opened, switch to the browser.
23. Open `http://localhost:8080/LeanService/` and verify that the application is running.
24. Enter some test data into the form displayed on the page. Examine the data generated in Output and HTTP Server Monitor tabs.
Note: This was basically the same as running the application. The IDE started the server, started the browser, and so on. The only difference now is that the application is being profiled. 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.
Note: 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.
25. Click **Live Results** in the Profiler window in the upper left or take a snapshot. Note how the information displayed is very similar to earlier projects.
26. Select **Profile > Stop Profiling Session**.
27. Close the **LeanService** Project.
28. Close **NetBeans**.

Practice 5-7: Profiling an Application by Using Oracle Studio

Overview

In this practice, you profile an application by using Oracle Studio.

Assumptions

Oracle Studio is installed and is in the path of the system you are using.

Tasks

1. Open a command-line window.
2. Change directory to `$home/labs/less05/studio01`.
3. Use the `javac` command to compile the following Java source files.

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

Note: If you receive the following warning, you can ignore it:

```
note:./Queue.java uses unchecked or unsafe operation
Note: Recompile with -Xlint:unchecked for details
```

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

```
java BatchProcessor
```

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

5. Run the Sun Studio Collector utility by using the following command.

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

This command invokes the Sun Studio Collector to capture a running profile of the Java program. It may take a few seconds for the application to start and produce its normal text output.

Note: The utility creates a directory in the current directory called `test.er.1`. If the directory is not created, make sure that you have write permissions to the directory.

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

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

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

Note: The `test.1.er` is the output directory created by the Collector. The Analyzer might produce a warning dialog box when it loads the profile data file. This is expected in this demo and can be ignored.

By default, Analyzer displays *exclusive* and *inclusive* 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 the time for any methods and any child methods in the call tree.

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

7. Add additional columns by selecting **View → Set Data Presentation**.
8. Select percentage(%) for both **Exclusive** and **Inclusive** user CPU time. Click **Apply** and then click **Close** to close the dialog.

Note: 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 the overall percentage of time. System CPU data is useful for tracking down methods consuming a large amount of system or kernel CPU. If high system or kernel CPU is not a concern for an application under profile, there is probably not much to be gained from looking at that data. The same is true for User Lock data.

9. Note the **View Mode** combo box at the top center of the user interface.

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 because 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 bytecode is viewable for Java methods. Likewise, Java bytecodes for a method that is the most expensive can be viewed too.

Machine mode shows method and function names from the JVM along with JIT-compiled method names and native method names. Some of the JVM method or function names represent transition code between interpreted Java code, JIT-compiled code, and native code. The Dissassembly tab in machine mode shows generated machine code, not Java bytecode seen in User and Expert modes. Machine mode also shows operating system lock primitives in method lists.

10. Select **View → Filter Data**.

Note: 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.

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.

11. Click on **test.1.er/_x1.er** from the list of experiments.
12. Under Samples, change the range from all to **16-75**.
13. Click **Apply** and then **Close**. Review the changes in the output.
14. Go back to **View → Filter** and reset the sample to **all**.

Note: Likewise, if you knew there were thread IDs you were not interested in, you could exclude them, too. Similarly you could exclude LWPs and CPUs. However, filtering on a range of samples is probably the most valuable use of this feature.

15. Close the **Filter** dialog box.
16. Click the Function tab.

17. Select a function in the Functions tab and then click the **Callers – Callees** tab. The method that calls the function you selected is shown in the top half of the window. The method you selected is in the bottom half.
Note: To view the source code, the *search path* must be configured in the **View > Set Data Presentation** dialog box under the **Search Path** tab.
18. Click the **Disassembly** tab.
 You can see the Java bytecode for the Java method selected. For each Java bytecode, it will show how much time is spent on a Java bytecode. The most expensive will be colored green.
19. Click the **Functions** tab. Look for the following entries:
JVM-System: Represents internal JVM methods
No Java callstack recorded: Represents methods the Collector/Analyzer is not able to determine. They are most likely JVM related.
20. Add **System CPU** as a metric and order it in **exclusive** descending order. Observe the change to the display.
21. Remove the **System CPU** results. . (This application has very little system CPU activity.)
22. Click on the **View Mode** drop down box and select **Expert**.
Note: Notice that Analyzer has resolved the information from the JVM-System 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.
23. Select the `ReadThread.doWorkOnItem` method and click the **Disassembly** tab.
Note: If the source for the `ReadThread` class is not available, the Java bytecode for the `ReadThread.doWorkOnItem` method is displayed. Studio will also show which Java bytecode is consuming the most time.
24. Click on the **Functions** tab.
25. Select a Java method name such as `java.lang.Thread.sleep`, where the Java source is available, and click the **Disassembly** tab.
26. Find the class declaration for `java.lang.Thread` and a green bar to the right of the scroll bar.
27. Click 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 bytecode associated with the time consumed. Click
28. Click on the **Functions** tab.
29. Select an internal JVM method, such as `java.util.Vector.size()`, and click the **Disassembly** tab. 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.
30. Click the **Functions** tab.
31. Click on the **View Mode** drop down box and select **Machine**.
Note: In this mode, any Java method selected will show the JIT generated assembly code for the Java method. You might not see the assembly on the first try of a Java method. You might have to switch back and forth between a couple Java methods before the assembly appears.
32. Select the `ReadThread.doWorkOnItem` method and click the **Disassembly** tab.

Note: 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 bar indicate the assembly instructions where the maximum time is spent.

33. Click the **Functions** tab.
34. Click the **View Mode** drop-down box and select **User**.
35. From the main menu select **View → Set Data Presentation**.
36. Deselect any System CPU value previously selected.
37. Deselect the User CPU Time that is inclusive.
38. Select **User Lock Time** under exclusive. Click OK to return to the Functions tab.
39. Sort the **User Lock Time** in **descending** order.
40. Notice that the `java.lang.Object.wait` method is at the top.
41. Select the `java.lang.Object.wait` method and click the **Callers - Callees** tab. You will see who calls the `java.lang.Object.wait` method.
42. Continue to double-click the method name in 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 the `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 have a locking-contention problem.
43. Close the **Analyzer** application.

Practice 5-8: Profiling Heap Memory with jmap and jhat

Overview

Profile an application by using the bundled jmap and jhat utilities.

Assumptions

A version of the JDK is installed.

Tasks

1. Run the Java2Demo with the following command-line switches:

```
java -client -Xmx12m -Xms3m -Xmn1m -XX:MaxPermSize=15m -
XX:PermSize=15m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. Click the **Transforms** tab of the Java2Demo GUI.

3. Obtain the LVMID of the Java2Demo process using the following command in another command window:

```
jps -l
```

Note: 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
```

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.

4. 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 by using VisualVM (performed later in this practice).

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

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

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

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

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

7. Scroll to the bottom of the web page to view general queries that you can perform on the heap dump.

8. Click the **Show heap histogram** query link under **Other Queries** section.

Note: This query identifies the biggest allocator of memory in the application.

9. Click the **Back** button on the browser.

10. Click the **Execute Object Query Language** link. This provides a form for querying application objects.

Note: A powerful capability that jhat supports is queries against the binary heap. It provides the ability to ask specific questions about the captured heap dump.

11. Enter `select s from java.awt.geom.AffineTransform s` in the form.
12. Click **Execute**. This should list all the instances of this class.
13. Try well-known classes like `java.lang.String` and look at the results. Enter `select s from java.lang.String s` in the form.
14. Close the web browser.
15. Quit jhat by executing a ^C in the window where it was started.
16. Start Java VisualVM: `jvisualvm`.
17. Inside VisualVM, use **File → Load** and filter by **Heap Dumps** to find the `heapdump.hprof` file. When the heap dump loads, you will see a summary screen that provides some high-level information.
18. Click the **Classes** button to display a heap profile view per-class in a traditional heap profiler manner.
19. Right-click any class and select **Show Instances View**. The Show Instances View enables you to “walk” the heap to see who has references to the selected object.
20. Close the VisualVM utility.

Practice 5-9: Profiling with NetBeans and Oracle Studio

Overview

This practice demonstrates the use of the complementary profiling capabilities of NetBeans Profiler and Oracle Studio Collector Analyzer.

Assumptions

NetBeans and Oracle Studio are installed on your system.

Tasks

1. Start **NetBeans**.
2. Select **Profile > Attach Profiler**.
3. Click the **CPU** icon on the left.
4. Select **Entire Application** on the right.
5. Change the filter to **Exclude Java core classes**.
6. Click **Attach**.
7. Select **Application** as the target type of profiling.
8. Select **Attach Method** of **local** and **Attach invocation** of **Dynamic**.
9. Select **Java SE 6.0** as the Java platform to run your application to complete the wizard interaction.
10. When the Select Process dialog box is displayed in NetBeans IDE, use a command-line window and start the Java2Demo application with the following command-line options.

```
java -server -Xmx32m -Xms24m -Xmn10m -XX:+UseSerialGC
-jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```
11. Click the **Refresh** button in the Select Process dialog box.
12. Select the running Java2Demo. Click **OK**.
13. Click 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.
14. Click the **Transforms** tab of the Java2Demo application.

Note: You can now observe the Java level methods that are consuming the most time in the application in NetBeans IDE.

The NetBeans Profiler should show that much time is spent in the `java2d.AnimatingSurface.run` method.
15. Open a command-line window and change directory to `exercises/less05/working`.

```
cd $home/labs/less05/working
```
16. Run Java2Demo by using the following command line.

```
collect -j on java -server -Xmx32m -Xms24m -Xmn10m -
XX:+UseSerialGC -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

Note: 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.1.er`.

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

17. After the Java2Demo is started, click the **Transforms** tab and let the Java2Demo run for about 15 or 20 seconds.
18. Quit the Java2Demo program.
19. Use the Sun Studio Analyzer to open the `test.1.er` 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.

NetBeans Profiler showed the percentage of its time was spent in the `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 the `java2d.AnimatingSource.run` method.

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

If you considered that the task underway was to identify performance-tuning opportunities with the Java2Demo, the view in Oracle 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 box to Expert or Machine shows the JVM methods where the time is being spent. This is great for JVM developers, but not of 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 classes of applications that are more difficult to set up, configure, and profile with traditional Java profilers such as JProbe and Optimizeit or with NetBeans Profiler.

20. When finished, exit analyzer.

Practice 5-10: Performing Memory Leak Profiling

Overview

Perform memory leak profiling with NetBeans Profiler.

Background

This is based on an actual usage of NetBeans Profiler as part of the development and testing of a production application. Andrés González of Spain used 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 multiday tests of his application (so it sounds as if he was using it as a convenient way to do long-term stress testing). He discovered something a bit 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 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](#) 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 webpages 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 scrutinize all those exceptions in the output, 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, it can host the servlet for you, in the same JVM as your test application. So the sample application consists of:

- A servlet that returns JavaScript that has an error
- A `main()` method that repeatedly requests a page from that servlet

This practice demonstrates some key features of NetBeans Profiler.

- NetBeans Profiler has a powerful capability to help you track down memory leaks.
- Using instrumentation, you can watch allocations on the heap happen in real time.
- The profiler provides statistical values that 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.
- With its tight integration into the developer workflow, 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.

Tasks

1. Start NetBeans if it is not already started.
2. Open the HttpUnit project in the `$home/labs/less05/httpunit` directory.
3. Right-click the **HttpUnit** project and select **Profile**.
4. Select **Memory** from the list of tasks to the left of the dialog box.
5. Select **Record both object creation and garbage collection**.
6. Select **10** for the value in “Track every object allocation.”
7. Select **Record stack trace for allocations**. Make sure that the **Use defined Profiling Points** option is *not* selected.
8. Click the **Run** button.
Note: 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.
9. Click the **Telemetry Overview** icon in the Profile window under the Controls panel heading (the icon looks like a graph). This opens the Telemetry Overview window at the bottom of the NetBeans IDE.
10. Observe that the purple graph on the left shows heap usage. The value is trending upward over time, but very slowly. It may take more than 30 seconds for this pattern to emerge.
Optional: You can click the **VM Telemetry** button under View if you want to see a larger version of the heap graph.
11. Click the **Live Results** icon in the Profile window.
Note: 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*.
12. Click the **Generations** column to sort the display by Generations.
Note: 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.

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. If, for example, an object is created at the beginning of an application and the garbage collector has run 466 times, 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.

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, 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, more and more objects are created at different points in time and therefore with different ages.

The increasing generation count for the `String` class (and its little friend `char` array :)) indicates Strings are being created repeatedly.
13. Right-click the entry for “String” and select **Take Snapshot** and **Show Allocation Stack Traces**.

14. Select **Profile → Stop Profiling Session**.

Note: 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.

So we know that `StringBuilder.toString()` is allocating strings that appear to be candidate memory leaks. So what? How do you tie that back to our application's usage of `HttpUnit`?

15. Click the icon next to the entry for `StringBuilder.toString()` to expand it.

Note: 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.handleScriptException()`.

16. Expand the entry for `handleScriptException()`. Because there is only one way it is being called, the profiler continues and expands the entire stack trace. You will see a straight line back to the `main()` method in the test application.

17. Right-click **Main.main** (for it to be visible, you will have to increase the width of the Method Name column).

18. Select **Go To Source**.

Note: One of the advantages of an integrated profiler is easy access to the source code. Here is the sample application calling `HttpUnit`'s `getResponse()` method on line 46, which ends up making the call that results in the memory leak.

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

19. Go back to the top of stack traces and you will find that `JavaScript.handleScriptException()` is the method that contains the memory leak.

20. Traverse to the `JavaScript.handleScriptException()` method and right-click **Go to Source**.

21. Examine the method and notice that a String is allocated with

```
final String errorMessage = badScript + " failed: " + e
on line 196. But, that is not the memory leak. The memory occurs at line 204:
    _errorMessages.add( errorMessage );
```

22. Hold down the Control key and move the cursor over `_errorMessages`. You will see a tool tip that says `_errorMessages` is an `ArrayList`. Alternatively, you can see it in the Navigator window on the left.

23. Right-click `_errorMessages` and select **Find Usages**. Any usage of the `_errorMessages` object in the application is displayed in a Usages tab in the output window.

Notice that there are calls to add elements, `clear()` elements, and return an array of elements.

24. Double-click the `_errorMessages.clear()` to see where it is called.
`_errorMessages.clear()` is called from `JavaScript.clearErrorMessages()`.
25. Now, select the `JavaScript.clearErrorMessages()` method in **Find Usages**. It is called in only one place by `JavaScriptEngineFactory.clearErrorMessages()`.
26. Select `JavaScriptEngineFactory.clearErrorMessages()` in **Find Usages**. This is called by `HttpUnitOptions.clearScriptErrorMessages()`.
27. Select `HttpUnitOptions.clearScriptErrorMessages()` in **Find Usages**. You see that nobody calls `HttpUnitOptions.clearScriptErrorMessages()`.
28. You 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`.
29. Close the project and exit `NetBeans`.

Practice 5-11: Using jhat to Detect Memory Leaks

Overview

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

Tasks

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

```
cd $home/labs/less05/prime
```

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

```
java -Xmx12m -XX:+HeapDumpOnOutOfMemoryError -jar  
PrimeNumbers.jar
```

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 be displayed fairly quickly. Click the button again with 1000000, and the result should display even faster, as this program caches the answers it previously calculated.
5. Click the **Calculate Prime Numbers** button again. The application should hang or die with a `java.lang.OutOfMemoryError: Java heap space` error, visible in the window where you started the `PrimeNumbers` program.
6. Kill the `PrimeNumbers` program if it did not die.
7. Use the window where you started the `PrimeNumbers` program to list the directory.
8. Look for a file with an `.hprof` extension. This is the heap dump that was generated by JVM when it encountered `OutOfMemoryError`. Note the name of the file.
9. Start `jhat` by using the name of the `.hprof` file as an argument. Here is an example:

```
jhat java_pid2944.hprof
```
10. When `jhat` reports the server is ready, use a browser to connect to the URL:
<http://localhost:7000>
11. In the browser, scroll to the bottom of the screen and select **Show Heap Histogram**.
On the resulting page you should notice there are a very large number of bytes allocated to `int` arrays, denoted by `[I`. You will see some 2000+ instances of `int` arrays that have allocated over 8,000,000+ bytes. This pattern should draw your attention enough to investigate further. For example, if you divide 8,000,000 by 2000, you should get an average of 4000 bytes per instance. In other words, each `int` array averaging 4000+ bytes requires further investigation.
12. Click the back button on the browser window and click the **Execute Object Query Language** (OQL) query.
13. Enter the following query in 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 4000.

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

Notice that each one of these int arrays is a reference 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 appears to hold what is clearly a large set of prime numbers. Remember we were running a program that calculated prime numbers.

15. The next step is to identify who has references to the `HashMap.Entry` holding the integer arrays.
16. Perform the following actions on both links. If you continue to click **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 **References to this object** to find what holds the reference to 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 see how it is used.

17. Open the **PrimeNumbers** project in NetBeans.
18. Open the **PrimeNumbers.java** file in NetBeans.
19. Look at how `PrimeNumbers.completeResults_` is used. You will find it is declared in `PrimeNumbers` and you will find values are *put* in it in the `PrimeCalculator.construct` method. But, you should find no other usage. The fact that `completeResults_` is a `HashMap` of `int[]` keys and `int[]` values, and only `put` methods are invoked, suggests that the memory leak has been found.
20. You can close the browser window and exit the `jhat` application.

Practice 5-12: Profiling Memory Leaks with VisualVM

Overview

In this practice, you profile an application with VisualVM.

Assumptions

Completion of the previous practice

Tasks

1. Start VisualVM: `jvisualvm`.
2. Select **File** → **Load** and load the same `.hprof` file that was generated when the `PrimeNumber` GUI experienced an `OutOfMemoryError`. You should have made a note of the file name in the practice titled “Using `jhat` to Detect Memory Leaks.”
3. This will load the heap dump in VisualVM. On the right panel you should see a **Summary** tab.
4. Click the **Classes** button on the right and order by **Size** in descending order.
Note: Notice the same `int[]` pattern discovered in the previous practice, again with some 2000+ instances and 8,000,000+ bytes allocated.
5. Select the `int[]` row, and then right-click and select **Show Instances View**. In the Instances view, the left panel is ordered by instances allocating the most bytes.
6. Select the top instance. 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`.
7. Select and right-click the `int[]` or either the `HashMap.Entry.key` or `HashMap.Entry.value` and select **Show Nearest GC Root**. You will find the previously discovered `PrimeNumbers.completeResults_`.
8. For a second use case, leave VisualVM running and restart the `PrimeNumbers` application again, this time without the `-XX:+HeapDumpOnOutOfMemoryError` command-line switch.
9. Notice in the left panel that VisualVM automatically discovered the `PrimeNumbers` GUI Java application.
10. Select and right-click the `PrimeNumbers` GUI Java application icon in the left panel of VisualVM and select **Enable Heap Dump on OOME**. The `PrimeNumbers` application will create a heap dump when it experiences an `OutOfMemoryError`.
11. Again, enter 1000000 in the `PrimeNumbers` GUI. Repeat this step to generate an `OutOfMemoryError`. Note where the heap file is written to in the output window that the `PrimeNumbers` application was started in.
12. Kill the `PrimeNumbers` GUI if it is hung.
13. Load the heap dump file in VisualVM as you did in step 2, but this time you will be loading a new heap dump file, as identified in step 11.
14. Repeat steps 3–7 and you will find the same results.
Note: The two uses cases demonstrate the following:
 - Having a heap dump generated by an explicit `-XX:+HeapDumpOnMemoryError`
 - Enabling a heap dump on a memory error on a running application

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

15. Exit VisualVM when you are done.

Practice 5-13: Profiling with NetBeans Profiler's HeapWalker

Overview

The HeapWalker provides a complete picture of the objects on the heap **and** the references between the objects. It is especially useful for analyzing binary heap dump files produced when an `OutOfMemoryError` occurs.

The Find Nearest GC root feature can help track down memory leaks by showing the reference that prevents an object from being garbage collected. This feature is similar to VisualVM.

Assumptions

Use the PrimeNumbers project from the practice titled "Profiling Memory Leaks with VisualVM."

Tasks

1. Start NetBeans.
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. Stop the Profiler.
5. Right-click **PrimeNumbers** and select **Properties**; then click **Run**. Verify that VM Options is set to `-Xmx12m`.
6. If the number is set correctly, click **Cancel**.
7. Right-click the **PrimeNumber** project and select **Profile**.
8. Click the **Monitor** button on the left side of the dialog box and then click the **Run** button.
9. In the application's "Enter a number" field, enter 1000000 (that is, a one followed by six zeroes) and then click the **Calculate Prime Numbers** button.
10. The result is displayed as 999983. The result should be displayed fairly quickly.
11. Click the **Calculate Prime Numbers** button again. The application dies with a `java.lang.OutOfMemoryError: Java heap space` alert, which is visible in the Output window of the IDE.
12. Switch to the IDE window.
13. A dialog box is displayed asking if you want to open the heap dump that was generated in the HeapWalker. Click **Yes**.
Note: An `OutOfMemoryError` was thrown, so the profiler requested a standard binary heap dump snapshot from the JVM. You can open it in the profiler's HeapWalker to take a look at what is on the heap.
14. The HeapWalker opens with a **Summary** view.
Note: You can see the summary here: size, operating system, and the JVM system properties.
15. Click the **Classes** button and then click the **Size** column to sort by size.
Note: 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.
16. Click the first line, which is for `int` array, to highlight it.
Note: It looks like most of the heap is taken up by `int` arrays. Take a look at the object instances for `int` array.
17. Right-click the `int` array entry and select **Show in Instances View**.

Note: 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. Take a look inside.

18. Click the first `int` array instance (the one that is 4 million bytes).

Note: With an instance selected, you can see two things over on the right: its fields and a list of any objects that reference this particular instance.

19. Expand the **<items 0-499>** entry.

Note: Because this is an array, the list of fields is actually a list of array indexes, in groups of 500.

20. Scroll down through the list.

Note: This array holds **all** the prime numbers less than the requested value, with placeholder entries for integers that are not prime.

21. Close the **Fields** panel.

Note: You 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.

22. Right-click the entry for `this` in the References panel and click **Show nearest GC Root**.

Note: Garbage Collection roots are the objects that are never removed from the heap. They are the starting point for JVM's garbage collector. Any object that is reachable from a GC root cannot be removed from the heap.

23. The display expands so that the entry for `primeNumbers` is shown.

Note: 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, because 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_`.

24. Right-click `completeResults_` and select **Go To Source**.

Note: An advantage of an integrated tool is easy access to the source code. 😊

25. The `PrimeNumbers.java` file opens up; the `completeResults_` variable is on line 21.

26. Right-click the variable and select **Find Usages**.

Note: Interesting. This is a `Map`—let's see where it is used.

27. When the Usages window opens with the results, double-click the only result.

Note: Well, well, well. The complete list of candidate prime numbers is being put into this `Map`, but as you can see from the Usages results, **it is never removed**. So, you do not need the array anymore, but it cannot be garbage collected, that is a memory leak. One is easily found with the profiler's HeapWalker.

28. Select **Profile > Load Heap Dump**.

Note: As you saw earlier, the profiler's HeapWalker can also be used on any binary heap dump file produced by one of Oracle's JVMs. There are a variety of ways for 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 saw earlier, you can then open that file with this feature.

29. When finished, close NetBeans.

Practice 5-14: Finding Lock Contention

Overview

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

Assumptions

The Oracle Studio Collector is installed and working.

Tasks

1. Open a command-line window and change to the `studio01` directory.

```
cd $home/labs/less05/studio01
```
2. Use the `javac` command to compile the following Java source files if they are not already compiled.

```
javac BatchProcessor.java
javac ReadThread.java
javac ReadThread.java
javac WriteThread.java
```
3. Run the `BatchProcessor` program to ensure that 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 by using the following command.

```
collect -j on java -server -Xmx64m -Xms64m -Xmn24m -
XX:+UseSerialGC 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.1.er`. If the file is not created, check the directory to ensure that you have write permissions.
5. Let the `BatchProcessor` program run to completion.
Note: This program tends to work best on a system with two logical processors (that is, single processor dual-core Intel). It is hard to mimic lock contention on large multicore systems.
6. When the `BatchProcessor` application completes, run the Oracle Studio Analyzer program to read the results:

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

Note: The name of the file `test.1.er` may be different depending on the locale and version of Oracle Studio in use. For, example, it may be `analyser.1.er` rather than `test.1.er`.
7. In Analyzer, select **View > Set Data Presentation** from the main menu, then select the **User Lock Exclusive** and **Inclusive Time** check boxes and click **Apply**.
8. Sort the data by **Exclusive User Lock** in descending order.
9. Locate and select the `java.lang.Object.wait` method. Then, select the **Callers - Callees** tab.

10. On the Callers - Callees view, notice that the `Object.wait(long)` method is called by the `Object.wait` method.
11. Click the upper `Object.wait` method. Click the **Call Tree** tab to identify who calls it. Notice that it is called by the `Queue.dequeue` method. Also notice that if you click the **Functions** tab, the `Queue.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 the `java.lang.Object.wait` method, it is likely that it is a synchronized method or has a synchronized block in it.
12. Click the `Queue.dequeue` method to identify who is calling it. You should see that the `ReadThread.getWorkItemFromQueue` method calls the `Queue.dequeue` method.
13. Examine the source code for the `Queue.dequeue` method located in the `Queue.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 `Queue.dequeue` method versus time spent in the synchronized method `Queue.dequeue` method.
14. At this point, you would have identified that there is a lock on the `Queue.dequeue` method that is being contended for and the `Object.wait` method that is being called in the `WorkQueue.dequeue` method is associated with the traditional Thread wait/notify concept.
15. When finished, exit analyzer.

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.

Practices for Lesson 6: Garbage Collection Schemes

Chapter 6

Practices for Lesson 6: Overview

Practices Overview

In these practices, you will explore JVM ergonomics.

Practice 6-1: Discovering Ergonomic Selections

Overview

This practice demonstrates the actions that you can take to discover what default ergonomics choices you have.

Assumptions

JDK 6 is installed.

Tasks

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

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

2. If the output returned includes `-client`, ergonomics says the HotSpot server JVM is not the ergonomically selected HotSpot JVM on this machine due to machine resources the HotSpot JVM identified. However, that selection can be overridden explicitly by specifying the `-server` HotSpot command-line option.
3. In the case where HotSpot ergonomically identified the system as a “server class” machine, it will print output somewhat similar to that shown below, which was gathered from a two-CPU socket, quad-core Intel system with 8 GB of RAM. As you can see below, the HotSpot JVM ergonomically identified the system as a “server class machine” and selected the HotSpot Server JVM, along with values for sizing the maximum Java heap size, parallel garbage collector, and how many parallel garbage collector threads to use.

```
EDRSR6P1:~ $ java -XX:+PrintCommandLineFlags -version
-XX:MaxHeapSize=1073741824 -XX:ParallelGCThreads=8
-XX:+PrintCommandLineFlags -XX:+UseParallelGC
java version "1.6.0_23"
Java(TM) SE Runtime Environment (build 1.6.0_23-b05)
Java HotSpot(TM) Server VM (build 19.0-b09, mixed mode)
```

4. 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
5. If the output returned did not include `-client`, use the output to identify the selections made by ergonomics.

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.

Practices for Lesson 7: Garbage Collection Tuning

Chapter 7

Practices for Lesson 7: Overview

Practices Overview

In these practices, you will learn the process of sizing of GC heap spaces and using various GC monitoring tools.

Practice 7-1: Using JVM Heap Sizing

Overview

In this practice, run the Java2Demo and try to tune its GC spaces. Tune the heap for the serial collector, concurrent collector, and throughput collector.

The goal is to size the GC heap spaces to avoid a Full GC and keep the young generation and minor GC events rather short also. For a GUI application, responsiveness is important, so keep pause times to a minimum. If this were a batch-processing engine, pause times would not be an issue. Instead the focus would be raw throughput (least amount of time spent in GC). These are two different goals and require a slightly different strategy when sizing heap spaces. It is important to know what your goals and pause time requirements are for an application before starting the tuning practice.

In tuning the Java2Demo, the goal is to minimize pause times to under 10 milliseconds and avoid all full GC events.

Assumptions

JDK 6 or later is installed.

Tasks

1. Start the Java2Demo application with the following command line:

Note: The Java2Demo does explicit GC calls by using `System.gc()`. To keep the Java2Demo from responding to those explicit calls to `System.gc()`, add the `-XX:+DisableExplicitGC` command-line switch. Also explicitly use the Serial collector `-XX:+UseSerialGC`.

```
java -client -XX:+DisableExplicitGC -XX:+UseSerialGC -verbose:gc
-XX:+PrintGCDetails -Xmx12m -Xms3m -Xmn1m -jar
/usr/java/demo/jfc/Java2D/Java2Demo.jar
```

2. By adding `-verbose:gc, -XX:+PrintGCDetails`, detailed information will be displayed in the terminal window.
3. If you see full GCs occurring as a result of a growing Java heap space, you need first to determine a maximum Java heap size.
4. Click across all of the tabs, which will force the loading of almost all the needed class files.
5. Once you have 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`.
6. Observe some Java heap space expansion on full GC events. If you set `-Xmx` and `-Xms` to the same value, that will prevent the young generation and old generation from being resized. Remember, resizing Java spaces requires a full GC.
7. Also remember that permanent-generation space may be resized. So, use a tool to monitor permanent-generation Java heap space and then adjust or size permanent-generation space. Remember that you have to set both `-XX:MaxPermSize` and `-XX:PermSize`. You may be able to get by just sizing `-XX:PermSize`.

Note: If the young generation or minor collections are too lengthy, you may need to set the young-generation space size. You can experiment with sizing the 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.

8. Repeat the preceding steps by using VisualVM.

```
java -client -XX:+DisableExplicitGC -XX:+UseSerialGC -Xmx12m -Xms3m -Xmn1m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

9. Next, switch to using the concurrent collector, and swap out `-XX:+UseSerialGC` in the previous command line to `-XX:+UseConcMarkSweepGC`.

```
java -client -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -Xmx12m -Xms3m -Xmn1m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
```

Note: With the concurrent collector, you need a larger overall Java heap space. But, you may be able to use a smaller young-generation space.

10. When finished, close all applications used in the practice.

Practice 7-2: Using the PrintGCStats Script

Overview

Obtain and interpret GC output produced by the `-verbose:gc` option of the Java command.

Assumptions

JDK 6 or later is installed.

Tasks

1. Open a command-line window and change directory to the practices directory for this practice:

```
cd /export/home/student/labs/less07/PrintGCStats
```

The `gc-logs.txt` file GC output was 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).

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

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

3. Review the detailed documentation for `PrintGCStats` below.

PrintGCStats User Guide

`PrintGCStats` is a shell script that 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 runtime, 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 1.6 and later

Usage

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

ncpu

The number of CPUs on the machine where the Java application is running. Used to compute CPU time available and GC load factors. No default; must be specified at 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`. The default is 0 (disabled).

verbose

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

Practice 7-3: Using GCHisto

Overview

Use `GCHisto` to understand the GC statistics gathered from a running application.

Assumptions

Java 6 or later is installed

Tasks

1. Open a command-line window and change directory to the practices directory for this practice:

```
cd /export/home/student/labs/less07/GCHisto
```

2. Analyze the `gc-logs.txt` file by using `GCHisto`.

```
java -jar GCHisto.jar
```

3. Add `gc-logs-4.txt` to the `GCHisto`. Click each of the tabs in `GCHisto`.

GC Pause Stats Pane

This pane shows some GC statistics (GC number, total time, and so on) for the loaded traces and broken down by GC type. If only one trace is loaded, this pane will show the stats for that trace.

GC Pause Distribution Pane

This pane shows the pause-time distribution for the loaded traces, with one subpane per loaded trace. A set of check boxes on the left enable you to make individual GC types visible or invisible.

GC Timeline Pane

This pane shows the GC timeline for each trace, with one subpane per loaded trace. A set of check boxes on the left enable you to make individual GC types visible or invisible.

Zooming In and Out of the Charts

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

4. Load the remaining gc log files: `gc-logs-1.txt` `gc-logs-2.txt` `gc-logs-3.txt`
5. Compare the gc logs of each run to highlight the additional abilities of `GCHisto`.

GC Pause Stats Pane

If multiple traces 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 to 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 subpanes with graphs for different metrics.

Optional Practice

6. Remove the gc log file currently loaded in `GCHisto`. Do this by using the **Trace Management** tab.
7. Add the three gc log files from `/export/home/labs/less07/GCHisto/moreLogs`;

jbb-ps-1g.log, jbb-po-1g.log, jbb-cms-1500m.log.

These are three gc logs from the SPECjbb2005 benchmark. One is using the young-generation parallel collector (jbb-ps-1g.log). One uses the parallel old collector, which enabled both young-generation and old-generation parallel collection (jbb-po-1g.log). The other is the CMS collector. The parallel collection logs use a 1-GB Java heap, and the CMS logs use a 1.5-GB Java heap.

8. Explore and compare the behavior of these three GC configurations and determine which performs the best.

Select the best configuration for each of these responsiveness goals:

- All pause times must be less than one second
 - Ninety-five percent of all pause times must be less than one second
 - If pause times are not a concern
9. When you are done, close the GCHisto application.

Practices for Lesson 8: Language-Level Concerns and Garbage Collection

Chapter 8

Practices for Lesson 8: Overview

Practices Overview

There are no practices for this lesson.

Practices for Lesson 9: Performance Tuning at the Language Level

Chapter 9

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.

Practices for Lesson 9: Overview

Practices Overview

In these practices, you will work on certain best practices that you need to keep in mind while writing a Java application.

Practice 9-1: Testing Performance of String/StringBuffer/StringBuilder

Overview

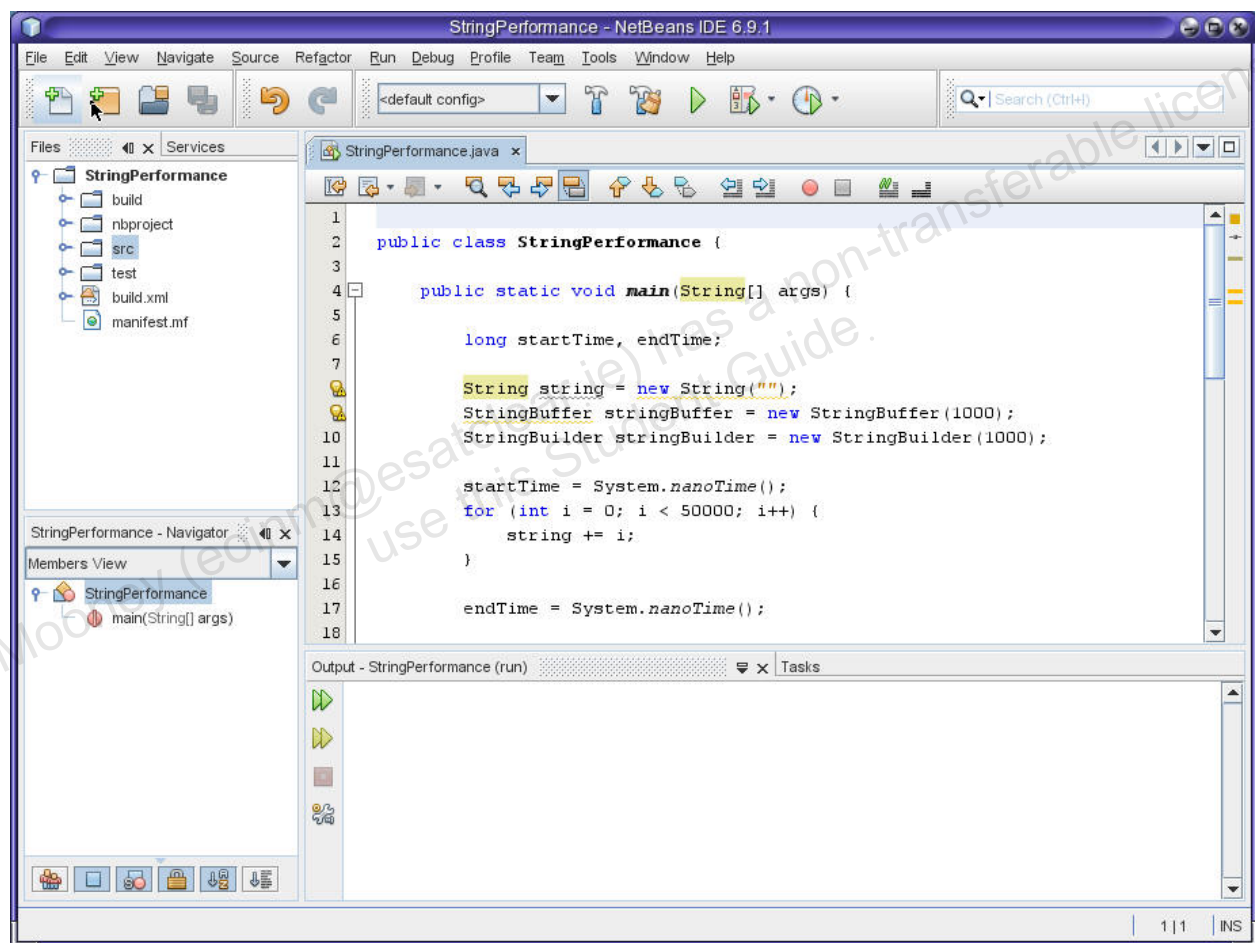
In this practice you will be able to test the performance of String/StringBuffer/StringBuilder.

Assumptions

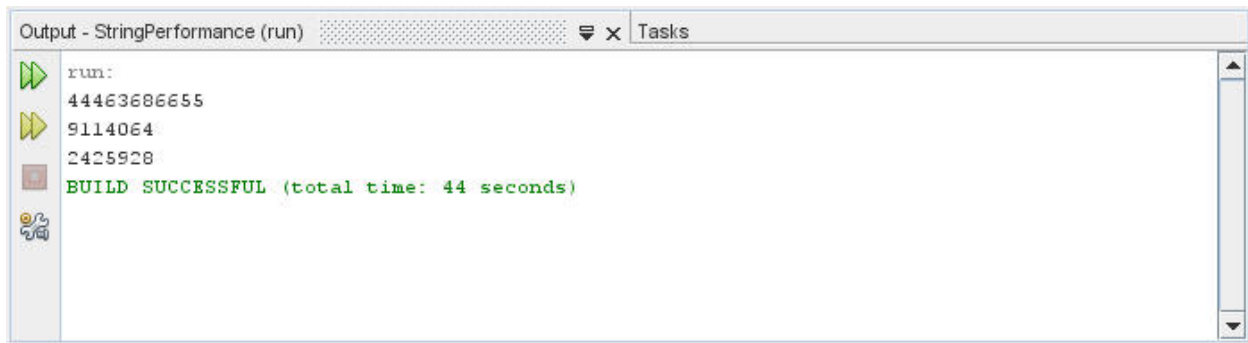
NetBeans is installed

Tasks

1. Start NetBeans.
2. Open the "StringPerformance" project located in \$home/labs/less09. Your screen should look like the one below now.



3. Compile and run the project.
 - a. Experiment with the program a bit. Change the iteration steps in the for loop.
 - b. Check the output.



Note: The output might not be the same, but the values will have a significant difference, enough to prove that concatenation in a `String` happens much slower then compared to `StringBuffer` or `StringBuilder`.

4. Repeat the experiment again to test different methods of `String` and `StringBuffer`.
 - a. `.concat()` of `String` vs. `.append()` of `StringBuffer` at lines 14, 23 and 32
 - b. `.substring()` in `String` and `.substring()` in `StringBuffer` at lines 14, 23 and 32

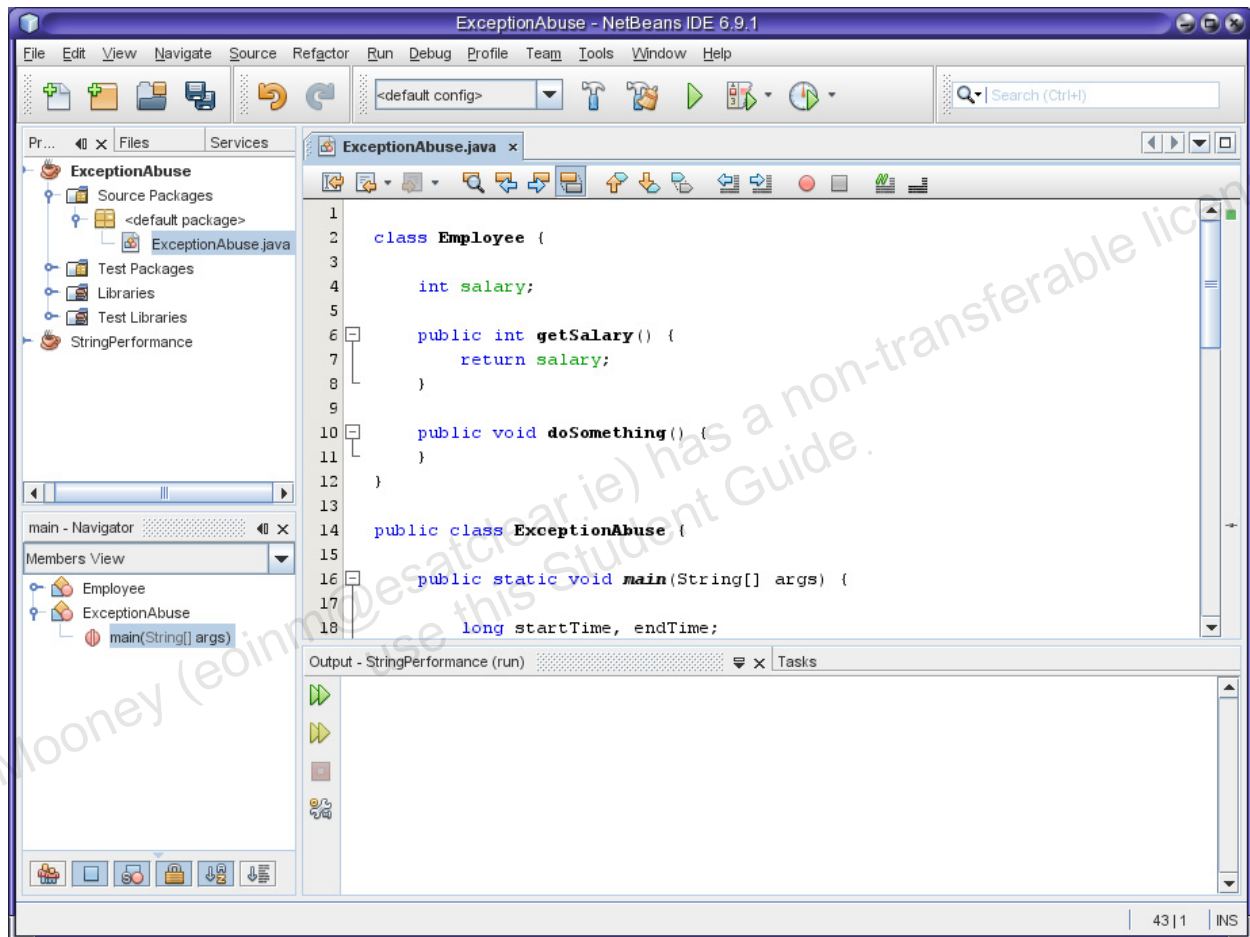
Practice 9-2: Performance Testing of Exceptions

Overview

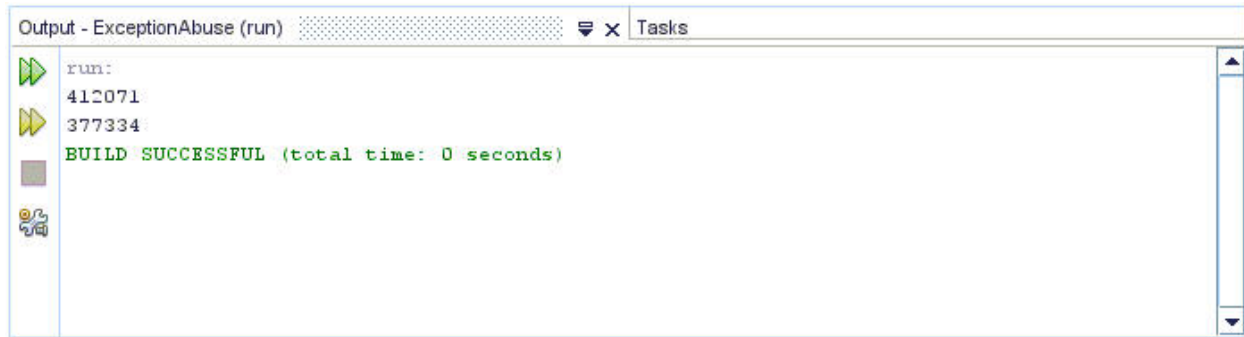
In this practice you will be able to understand why conditional logic should be preferred over exception handling.

Tasks

1. Open the “ExceptionAbuse” project located in `$home/labs/less09`. Your screen should look like the one below now:



2. Compile and run the project.
 - a. Experiment with the program a bit. Change the iteration steps in the `for` loop.
 - b. Check the output.



Note: The output might not be the same, but the values will have a significant difference, enough to prove that conditional logic is better than using exception handling mechanism.

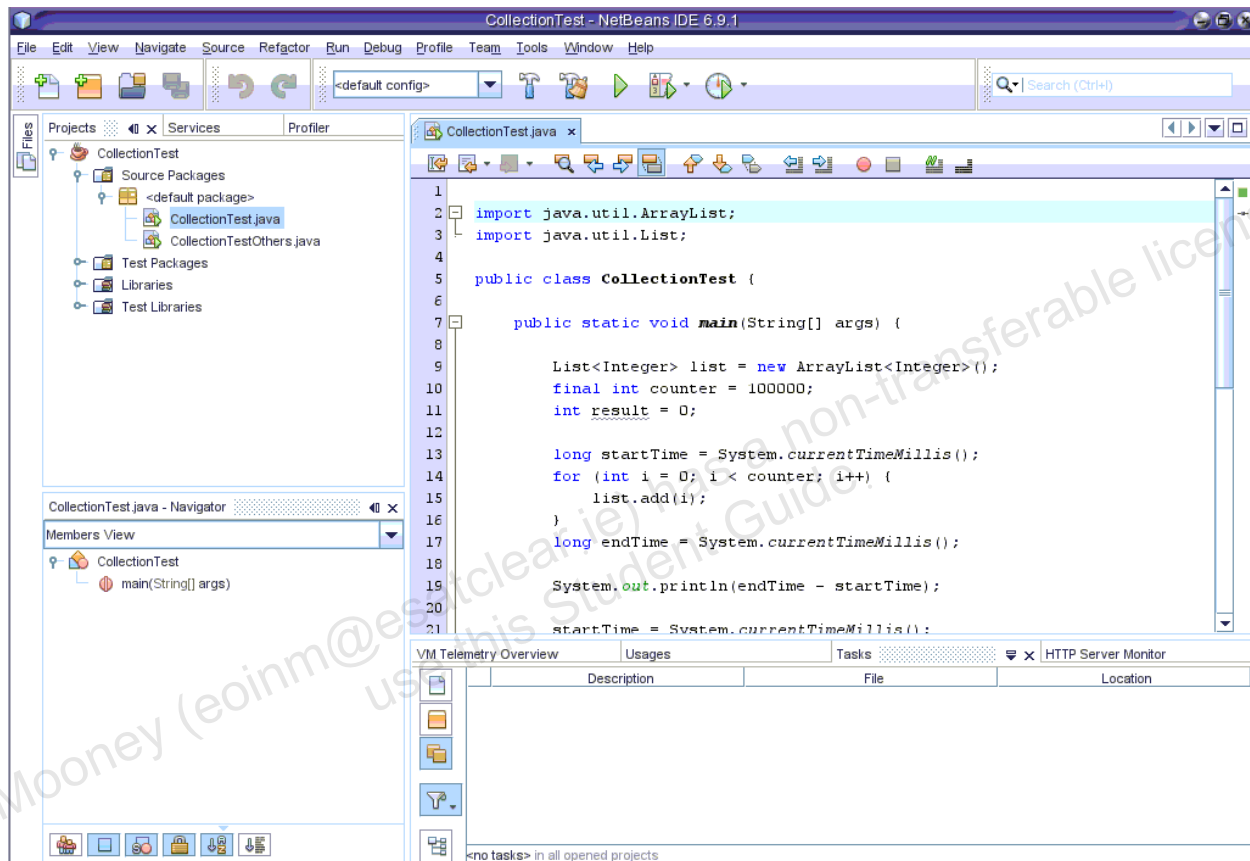
Practice 9-3: Performance Benchmarking of Collection Classes

Overview

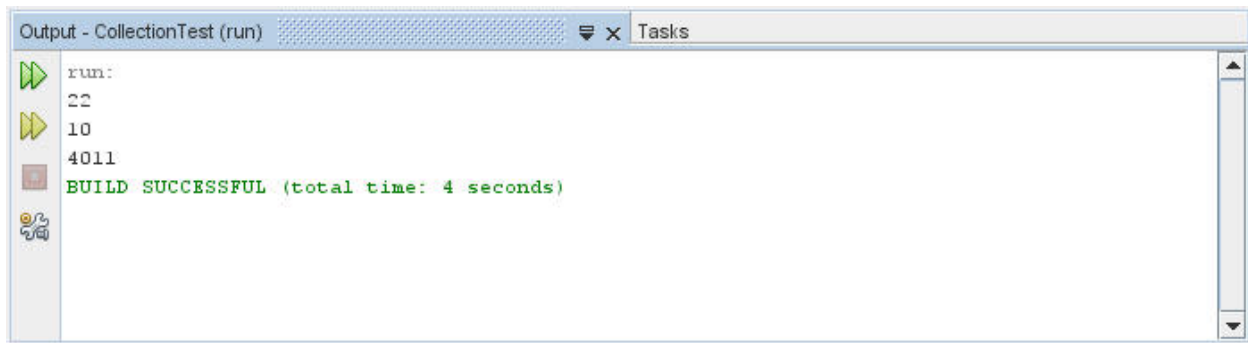
Examine the runtime of various classes in the collection framework.

Tasks

1. Open the “CollectionTest” project located in `$home/labs/less09`. Your screen should look like the one below now:



2. Compile and run the project.
 - a. Experiment with the program a bit. Change the iteration steps in the `for` loop.
 - b. Check the output.



```
run:
22
10
4011
BUILD SUCCESSFUL (total time: 4 seconds)
```

Note: The output might not be the same, but the values will have a significant difference.

3. Repeat the experiment again to test different collection classes:

- a. Vector
- b. Map
- c. LinkedList
- d. HashSet
- e. HashMap

Note: You can follow the example given in `$home/labs/less10/CollectionTestOthers` as a reference.

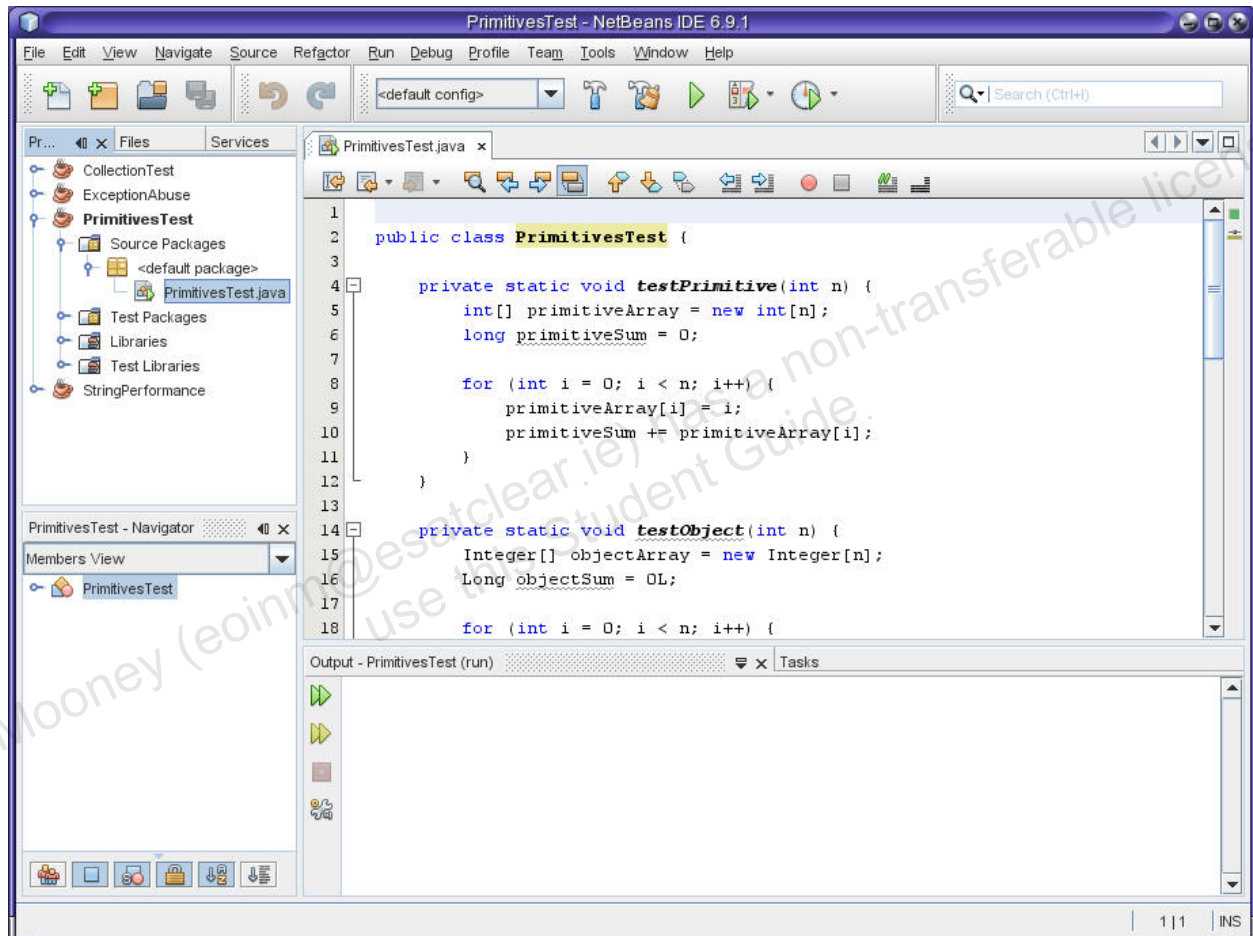
Practice 9-4: Benchmarking of Primitives vs. Object Types

Overview

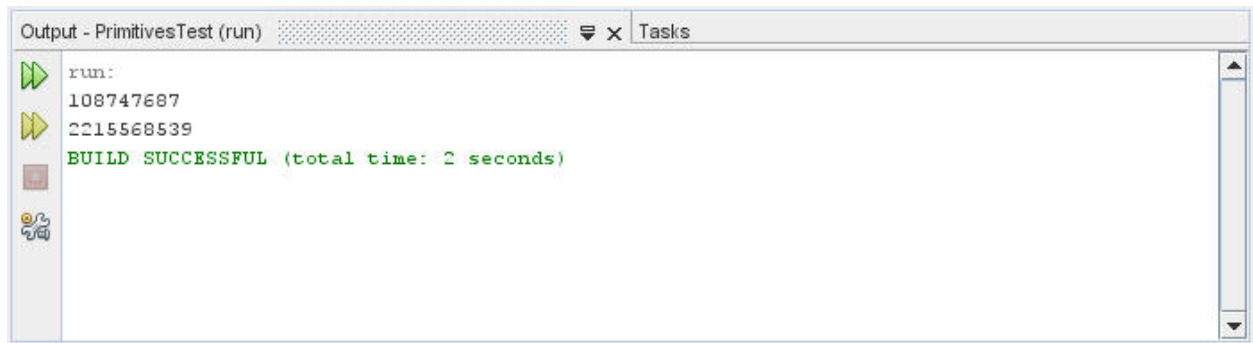
In this practice you will be able to understand why primitive types have to be preferred over object types. Also you will be able to understand that unnecessary usage of auto-boxing/auto-unboxing have to be avoided.

Tasks

1. Open the “PrimitivesTest” project located in `$home/labs/less09`. Your screen should look like the one below now:



2. Compile and run the project.
 - a. Experiment with the program a bit. Iterate the experiment for 10 times and notice the time taken.
 - b. Check the output.



```
run:
108747687
2215568539
BUILD SUCCESSFUL (total time: 2 seconds)
```

Note: The output might not be the same, but the values will have a significant difference, enough to prove that operations on the wrapper classes take more time than performing operations on primitives.

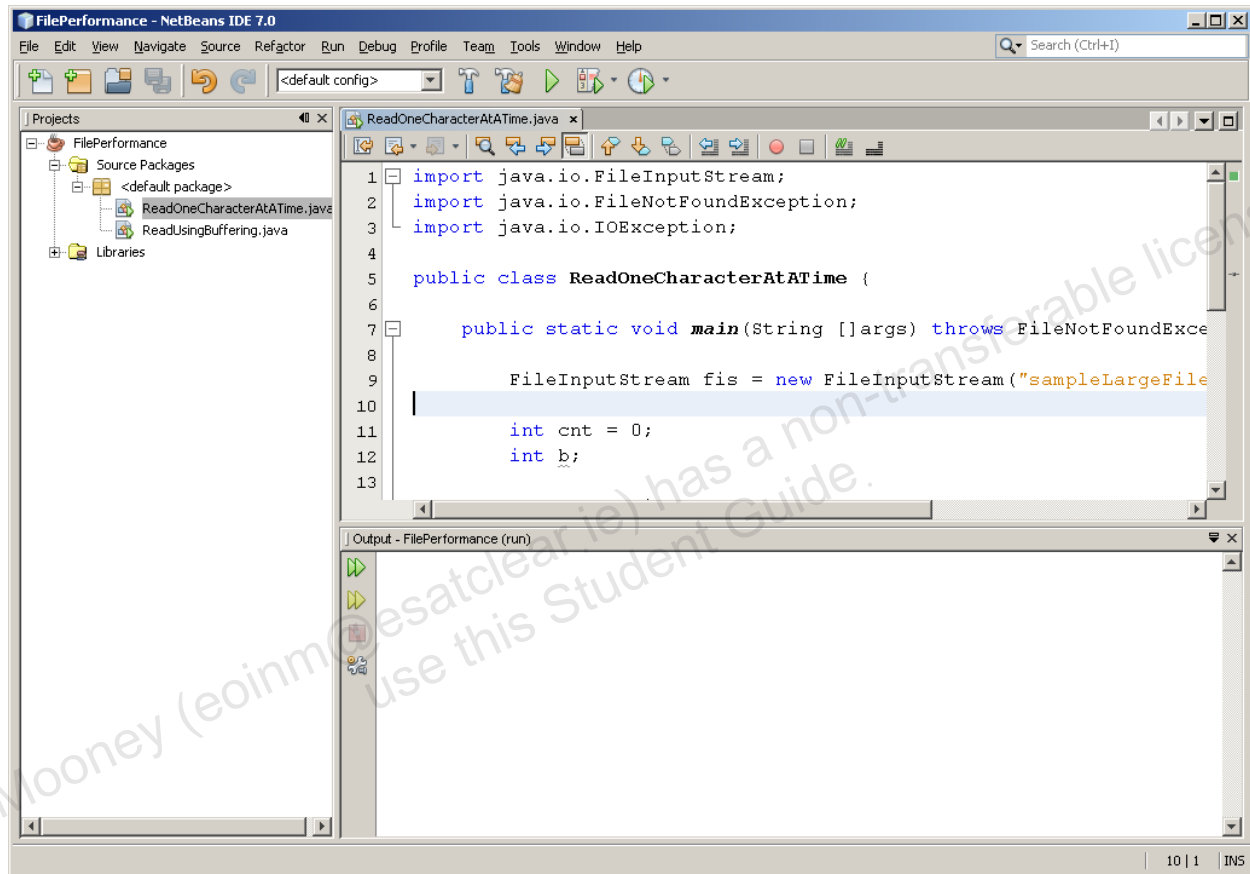
Practice 9-5: Benchmarking File Classes

Overview

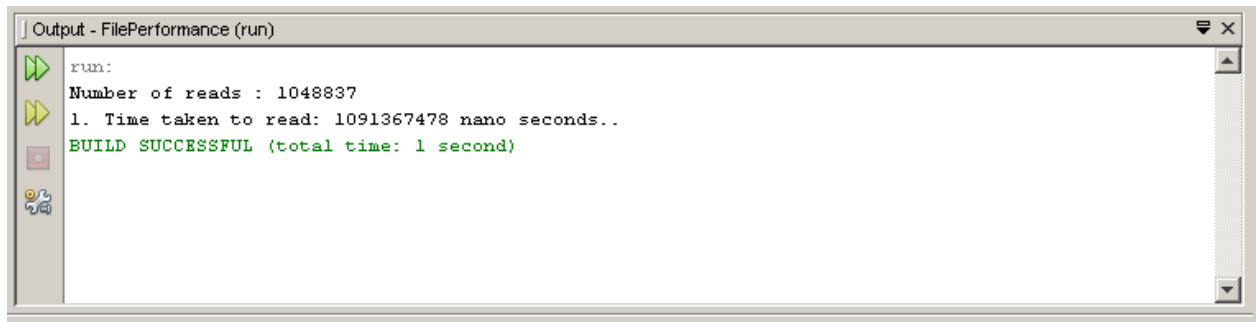
In this practice you will be able to understand why file buffering is very important.

Tasks

1. Open the “FilePerformance” project located in `$home/labs/less09`. Your screen should look like the one below now:



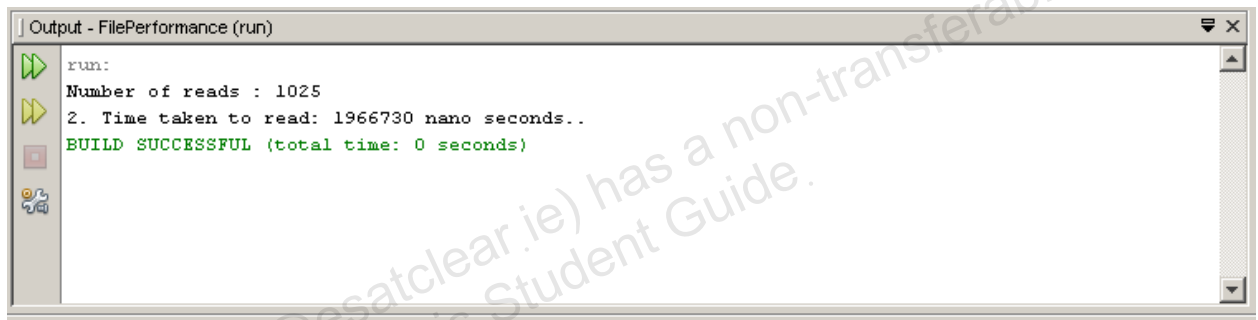
2. There are two .java programs, “ReadOneCharacterAtATime.java” and “ReadUsingBuffering.java”. ReadOneCharacterAtATime is using the FileInputStream to read the file, one character at a time, while ReadUsingBuffering is using a buffering mechanism and is reading 1024 characters at one time.
3. Open “ReadOneCharacterAtATime.java”.
 - a. Compile and run the file.
 - b. Experiment with the program a bit. Notice the time taken and the number of reads.
 - c. Check the output.



```
run:
Number of reads : 1048837
1. Time taken to read: 1091367478 nano seconds..
BUILD SUCCESSFUL (total time: 1 second)
```

Note: The output might not be the same. Just take a note of the values.

4. Open "ReadUsingBuffering.java"
 - a. Compile and run the file.
 - b. Experiment with the program a bit. Notice the time taken and the number of reads.
 - c. Check the output.



```
run:
Number of reads : 1025
2. Time taken to read: 1966730 nano seconds..
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: The output might not be the same. Just take a note of the values.

5. Note that the time difference taken and the number of I/O reads by the two programs to read the same file are significant enough to prove that buffering while reading a file is always better.

Eoin Mooney (eoinm@esatclear.ie) has a non-transferable license to use this Student Guide.