

Java Performance Tuning and Optimization

Student Guide

D69518GC10
Edition 1.0
June 2011
D73450

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

Introduction	1-1
Introduction.....	1-2
Course Goal	1-3
Course Objectives.....	1-4
Class Introductions	1-5
Audience	1-6
Prerequisites.....	1-7
Course Map.....	1-8
Course Topics	1-9
Course Schedule	1-10
Course Environment	1-11
Additional Resources	1-12
JVM and Performance Overview	2-1
JVM and Performance Overview	2-2
Objectives.....	2-3
JVM Overview	2-4
Java Programming Language.....	2-5
HotSpot JVM: Architecture	2-6
Key HotSpot JVM Components	2-7
What Is Performance?.....	2-8
Memory Footprint.....	2-9
Startup Time.....	2-10
Scalability	2-11
Application Scalability	2-12
Responsiveness	2-13
Throughput.....	2-14
Performance Focus for This Course	2-15
Performance Issues Covered in This Course	2-16
Performance Issues Not Covered in This Course	2-17
Performance Methodology	2-18
Performance Monitoring	2-19
Performance Profiling.....	2-20
Performance Tuning.....	2-21
Typical Development Process	2-22
Application Performance Process	2-23
Summary.....	2-24
Java Performance Resources.....	2-25
Additional Resources	2-26
Monitoring Operating System Performance.....	3-1
Monitoring Operating System Performance.....	3-2
Objectives.....	3-3
Why Are We Monitoring?.....	3-4
Monitoring CPU Usage Overview	3-5
CPU Monitoring Performance Indicators	3-6
Voluntary Context Switching (VCX).....	3-7
Involuntary Context Switching (ICX).....	3-8
Tools For Monitoring CPU Usage	3-9

CPU Usage: vmstat.....	3-10
CPU Usage: mpstat	3-11
CPU Usage: prstat	3-12
CPU Usage: prstat -m	3-13
CPU Usage: prstat -Lm	3-14
CPU Monitoring: Solaris - cpubar.....	3-15
Map Lightweight Processes to Java Threads	3-16
Monitoring Network I/O Overview	3-17
Network I/O: Using tcptop.....	3-18
Network I/O: Using nicstat	3-19
Monitoring Disk I/O Overview	3-20
Disk I/O: iotop	3-21
Monitoring Virtual Memory: Overview.....	3-22
Virtual Memory Tools	3-23
Virtual Memory: vmstat.....	3-24
Virtual Memory: Swapping Example	3-25
Monitoring Virtual Memory: cpubar	3-26
Virtual Memory: Fixing the Swapping Problem	3-27
Monitoring Processes Overview.....	3-28
Process Monitoring Tools	3-29
Processes: prstat-Lm	3-30
Processes: mpstat	3-31
Monitoring the Kernel	3-32
Kernel: vmstat.....	3-33
Kernel: mpstat	3-34
Kernel: prstat-Lm	3-35
Summary.....	3-36
Monitoring the JVM.....	4-1
Monitoring the JVM.....	4-2
Objectives.....	4-3
What to Monitor	4-4
HotSpot GC Basics	4-5
The Young GC Process: Part 1	4-6
The Young GC Process: Part 2	4-7
The Young GC Process: Part 3	4-8
The Young GC Process: Part 4	4-9
The Young GC Process: Part 5	4-10
The Young GC Process: Summary	4-11
Young Generation Recap.....	4-12
Tenured (old) Generation	4-13
Permanent Generation.....	4-14
Tools for Monitoring GC	4-15
Using -verbose:gc.....	4-16
Additional -verbose:gc Print Options	4-17
Using -XX:+PrintGCDetails.....	4-18
Printing Pause Time.....	4-20
Using jps	4-21
Using jstat.....	4-22
Using jconsole	4-23

Using VisualVM	4-24
Using VisualGC	4-25
Using GCHisto	4-26
Monitoring JIT Compilation.....	4-27
Using -XX:+PrintCompilation	4-28
What Is the .hotspot_compiler File?	4-29
Using .hotspot_compiler File	4-30
Focusing on Throughput.....	4-31
Focusing on Responsiveness	4-32
Summary.....	4-33
Performance Profiling.....	5-1
Performance Profiling.....	5-2
Objectives.....	5-3
Tools for Profiling Java Applications	5-4
NetBeans and NetBeans Profiler	5-5
NetBeans	5-6
Oracle Solaris Studio	5-7
jmap and jhat	5-10
Profiling Tips.....	5-11
CPU Profiling: Why and When.....	5-12
CPU Profiling: Strategies.....	5-13
CPU Profiling Entire Application.....	5-14
CPU Profiling a Portion of the Application	5-15
Heap Profiling: Why and When.....	5-16
Heap Profiling: Strategies.....	5-17
Heap Profiling: jmap/jhat	5-19
Heap Profiling: jmap/jhat Strategies	5-20
Memory Leak Profiling: Why and When	5-21
Memory Leaks Profiling Tips: Tools	5-22
Memory Leaks: NetBeans/VisualVM Strategies.....	5-23
Memory Leaks: jmap/jhat Strategies	5-24
Lock-Contention Profiling: Overview	5-25
How to Reduce Lock Contention.....	5-26
Biased Locking	5-27
Inlining Effect	5-28
Identifying Anti-Patterns	5-29
Anti-Patterns in Heap Profile	5-30
Anti-Patterns in Heap Profiles.....	5-31
Anti-Patterns in Method Profiles.....	5-34
Summary.....	5-36
Garbage Collection Schemes	6-1
Garbage Collection Schemes	6-2
Objectives.....	6-3
Garbage Collection Basics	6-4
Normal Deletion	6-6
Deletion with Compacting.....	6-7
Generational Garbage Collection.....	6-8
Why Generational GC?	6-9
Generational Garbage Collection: Major Spaces	6-10

Features of Young Generational Space	6-11
Features of Old Generation Space.....	6-12
Generational Garbage Collection.....	6-13
Garbage Collection Notations.....	6-14
Generational Garbage Collection: Young Collection	6-15
GC Performance Metric.....	6-17
Choices of Garbage Collecting Algorithms	6-18
Serial versus Parallel	6-19
Stop the World Versus Concurrent.....	6-20
Compacting Versus Non-Compacting Versus Copying	6-21
Types of GC Collectors	6-22
Serial Collector	6-23
Serial Collector on Young Generation: Before	6-24
Serial Collector on Young Generation: After	6-25
Serial Collector on Old Generation.....	6-26
Parallel Collector: Throughput Matters!	6-27
Parallel Collector on Young Generation	6-28
Parallel Compacting Collector.....	6-29
Concurrent Mark-Sweep (CMS) Collector	6-31
CMS Collector Phases	6-32
CMS Collector on Old Generation.....	6-33
Garbage Collectors: Comparisons	6-35
Ergonomics: What It Does	6-36
Ergonomics	6-38
Summary.....	6-39
Garbage Collection Tuning.....	7-1
Garbage Collection Tuning.....	7-2
Objectives.....	7-3
Garbage Collectors: Recap	7-4
Garbage Collection: Myth	7-5
Garbage Collectors: Sizing Java Heap Spaces	7-6
Garbage Collectors: Sizing Spaces.....	7-8
Garbage Collectors: The Choices	7-9
Garbage Collectors: Serial Collector	7-11
Garbage Collectors: Throughput Collector	7-14
Garbage Collectors: Concurrent Collector	7-19
Serial Collector Versus Parallel Collector	7-28
Parallel Collector Versus CMS Collector	7-29
Garbage Collectors: Permanent Generation.....	7-30
GC Output: Using Serial Collector	7-32
GC Output: Using Throughput Collector.....	7-34
GC Output: Using Concurrent Collector	7-36
Concurrent Collector: Losing the Race.....	7-38
Garbage Collectors: PrintGCStats	7-39
Garbage Collectors: GCHisto	7-42
Summary.....	7-43
Language-Level Concerns and Garbage Collection	8-1
Language-Level Concerns and Garbage Collection.....	8-2
Objectives.....	8-3

Object Allocation: Best Practices	8-4
Object Allocation: Working with Large Objects	8-5
Garbage Collectors: Explicit GC	8-6
Data Structure Sizing	8-7
Garbage Collectors: Reference Objects	8-8
Reference Objects, Illustration (1/2)	8-9
Reference Objects: Illustration (2/2)	8-10
Reference Objects: Soft Reference	8-11
Reference Objects: Weak Reference	8-12
Reference Objects: Phantom Reference	8-13
Garbage Collectors: Soft References	8-14
Memory Leaks	8-15
Garbage Collectors: Finalizers	8-17
Finalizers Versus Destructors	8-18
Summary	8-19
Performance Tuning at the Language Level	9-1
Performance Tuning at the Language Level	9-2
Objectives	9-3
Strings: An Introduction	9-4
Compile Time Versus Runtime Strings	9-5
How JVM Works with Strings	9-6
Optimization by Interning Strings	9-7
String Versus StringBuffer Versus StringBuilder	9-8
StringBuffer Versus StringBuilder (in Nanoseconds)	9-9
Execution of "+" Operator in String (in Milliseconds)	9-10
Exception Handling and Performance	9-11
Exception Handling and Performance (in Nanoseconds)	9-12
Primitives Versus Objects	9-13
Primitives Versus Objects [Benchmark Results]	9-14
Prefer Reusing Objects	9-15
Thread Synchronization	9-16
Collections	9-18
Performance of Java Collections	9-20
Benchmark Results	9-21
Copying Entire Array: Use System.arraycopy	9-22
Tuning Java I/O Performance	9-23
Speeding Up I/O	9-24
Speeding Up I/O: Approach 1	9-25
Speeding Up I/O: Approach 2	9-26
Speeding Up I/O: Approach 3	9-27
Speeding Up I/O: Performance Benchmarking	9-28
Buffering	9-29
Reading/Writing Text Files	9-30
Formatting Costs	9-32
Formatting Costs: Use MessageFormat Class	9-34
Formatting Costs: Benchmarking	9-35
Random Access	9-36
Compression	9-37
Tokenization	9-38

Serialization.....	9-39
Summary.....	9-40
Appendix A: Monitoring Linux Performance.....	10-1
Monitoring Linux Performance.....	10-2
Objectives.....	10-3
Tools For Monitoring Linux	10-4
CPU Usage: Linux - vmstat	10-5
CPU Usage: Linux mpstat	10-6
CPU Usage: pidstat.....	10-7
Linux – Gnome System Monitor.....	10-8
Monitoring Network I/O: Using nicstat	10-9
Monitoring Disk I/O: pidstat Example	10-10
Kernel Monitoring Using: pidstat	10-11
Summary.....	10-12
Appendix B	11-1
Appendix B.....	11-2
Objectives.....	11-3
Java HotSpot VM Options	11-4
Categories of Java HotSpot VM Options.....	11-5
Useful -XX Options	11-6

Preface

Profile

Before You Begin This Course

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 lecture 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*
- Typographic Conventions
- Typographic Conventions for Words within Regular Text

Typographic Conventions

The following two lists explain Oracle University typographical conventions for words that appear within regular text or within code samples.

1. Typographic Conventions for words within regular text

Convention	Object or Term	Example
Courier new,	User input; commands; column, table, and schema names; functions; PL/SQL objects; paths	Use the SELECT command to view information stored in the LAST_NAME column of the EMPLOYEES table. Enter 300. Log in as scott
Initial cap	Triggers; user interface object names, such as button names	Assign a When-Validate-Item trigger to the ORD block. Click the Cancel button.
Italic	Titles of courses and manuals; emphasized words or phrases; placeholders or variables	For more information on the subject see <i>Oracle SQL Reference Manual</i> Do <i>not</i> save changes to the database. Enter <i>hostname</i> , where <i>hostname</i> is the host on which the password is to be changed
Quotation marks	Lesson or module title referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

2. Typographic Conventions for words within code samples

Convention	Object or term	Example
Uppercase	Commands, functions	SELECT employee_id FROM employees
Lowercase italic	Syntax variables	CREATE ROLE <i>role</i>
Initial cap	Forms triggers	Form module: ORD Trigger level: S_ITEM.QUANTITY item Trigger name: When-Validate-Item . . .
Lowercase	Column names, table names Filenames, PL/SQL objects	. . . OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer')) . . . SELECT last_name FROM employees;
Bold	Text that must be entered by a user	CREATE USER scott IDENTIFIED BY tiger ;

3. Typographic Conventions for Oracle Application Navigation Paths

This course uses simplified navigation paths, such as the following example, to direct you through Oracle Applications.

(N) Invoice > Entry > Invoice Batches Summary (M) Query > Find (B) Approve

This simplified path translates to the following:

1. (N) From the Navigator window, select **Invoice** then **Entry** then **Invoice Batches Summary**.
2. (M) From the menu, select **Query** then **Find**.
3. (B) Click the **Approve** button.

Notations:

- (N) = Navigator
- (M) = Menu
- (T) = Tab
- (B) = Button
- (I) = Icon
- (H) = Hyperlink
- (ST) = Sub Tab

4. Typographic Conventions for Oracle Application Help System Paths

This course uses a “navigation path” convention to represent actions you perform to find pertinent information in the Oracle Applications Help System.

The following help navigation path, for example—

(Help) General Ledger > Journals > Enter Journals

—represents the following sequence of actions:

1. In the navigation frame of the help system window, expand the General Ledger entry.
2. Under the General Ledger entry, expand Journals.
3. Under Journals, select Enter Journals.
4. Review the Enter Journals topic that appears in the document frame of the help system window.

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

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

Introduction

Chapter 1

Introduction

1

Introduction

ORACLE

Course Goal

Course Goal

The main goal of this course is to teach students how to monitor, profile, and tune the performance of Java applications.

ORACLE

Course Objectives

Course Objectives

At the completion of this course, you should be able to:

- Describe basic principles of performance
- Monitor operating system performance on Solaris, Linux, and Windows
- Monitor performance at the JVM and application level
- Profile the performance of a Java application
- Describe various garbage collection schemes
- Tune garbage collection in a Java application
- Apply basic performance tuning principles to a Java application
- Tune the performance of a Java application at the language level
- Apply best practices for performance testing

ORACLE

Class Introductions

Class Introductions

Briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending

ORACLE

Class Introductions

Take a moment to share the above information with the class and instructor.

Audience

Audience

The target audience includes:

- Java SE developers
- Java EE developers
- Java technical architects

ORACLE

Audience

The target audience for this course is listed in the slide.

Prerequisites

Prerequisites

To successfully complete this course, you must know how to:

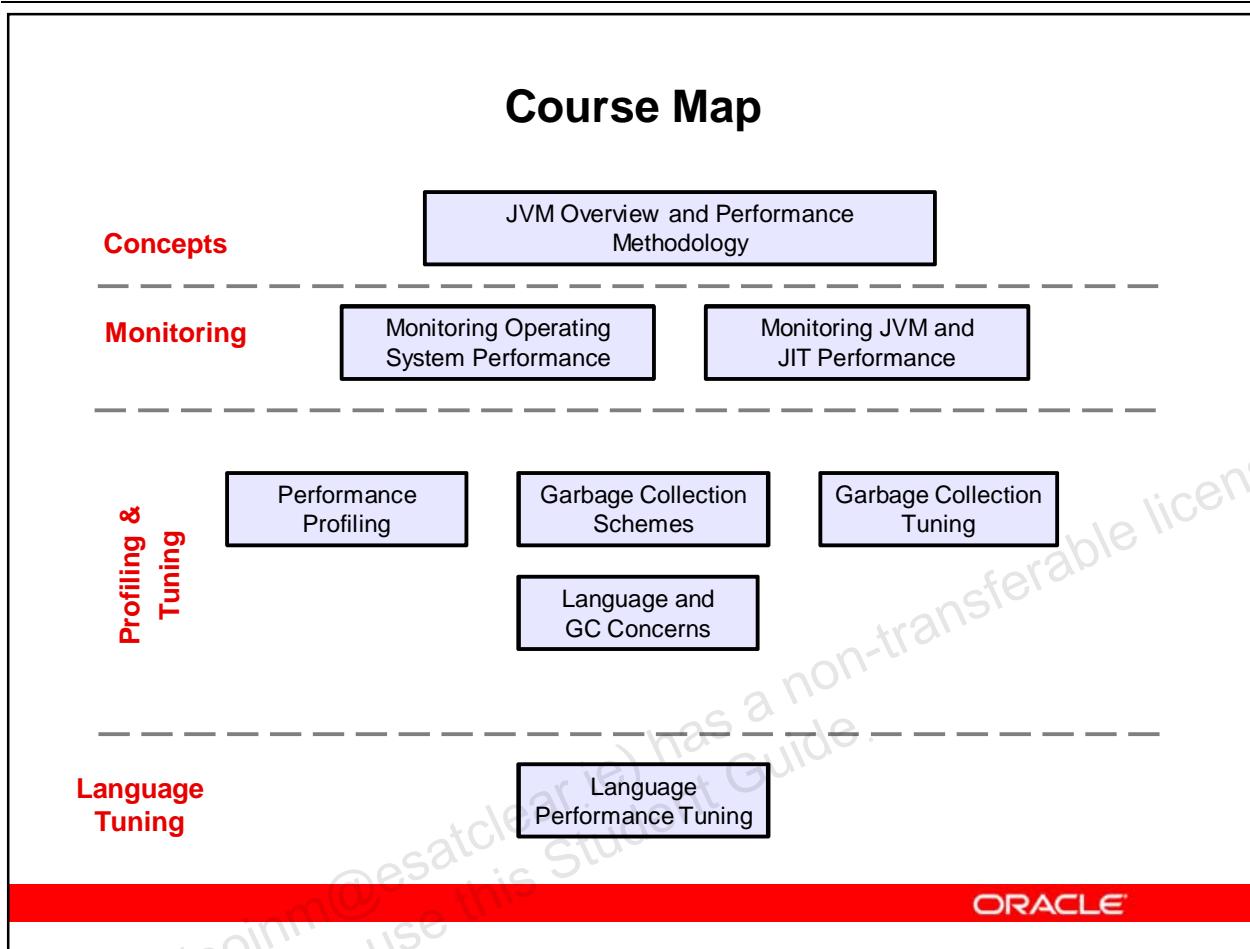
- Develop applications using the Java programming language
- Implement interfaces and handle Java programming exceptions
- Use object-oriented programming techniques

ORACLE

Prerequisites

The slide lists the key main prerequisites for this course.

Course Map



Course Map

The course map shows all the lessons of this course, and how they are grouped into logical sections.

Course Topics

Course Topics

In this course, you master Java SE performance monitoring by learning:

- What and where to monitor performance
- What to profile and what tools work best for different use cases
- Commonly observed patterns indicating performance issues
- How Java HotSpot VM garbage collectors work and how they are tuned
- What you need to know about the JIT compiler
- How JVM ergonomics works
- How to tune the JVM for specific hardware platforms

ORACLE

Course Schedule

Course Schedule

Session		Module
Day 1	A.M.	1: Introduction 2: Java Virtual Machine and Performance Methodology
	P.M.	3: Monitoring Operating System Performance 4: Monitoring JVM and JIT Performance
Day 2	A.M.	4: Monitoring JVM and JIT Performance 5: Performance Profiling
	P.M.	5: Performance Profiling
Day 3	A.M.	6: Garbage Collection Schemes 7: Garbage Collection Tuning
	P.M.	8: Language-Level Concerns and Garbage Collection 9: Performance Tuning at the Language Level

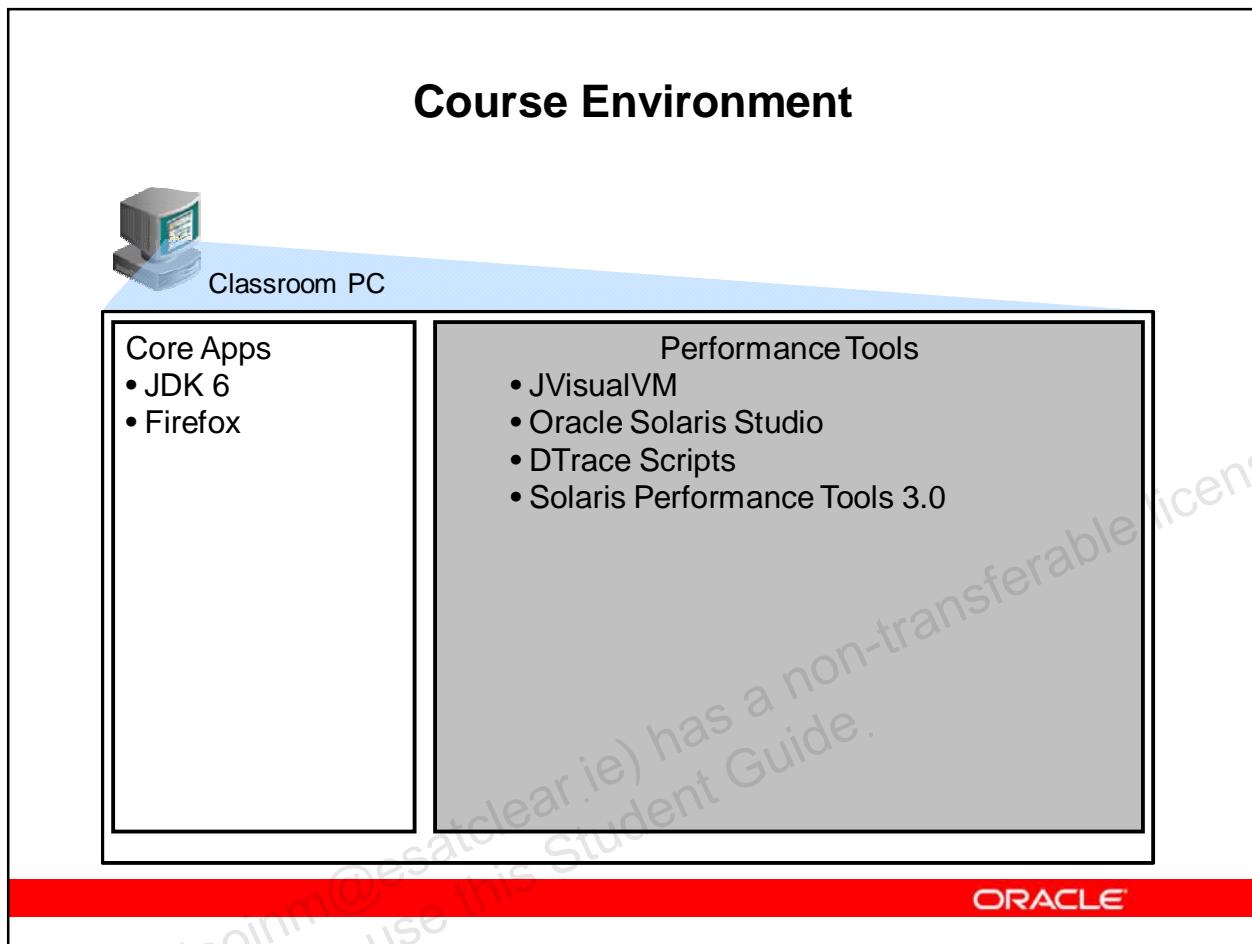
ORACLE

Course Schedule

The class schedule might vary according to the pace of the class. The instructor will provide updates.

At the end of the course, the instructor facilitates a feedback session that includes a written questionnaire. Oracle University uses your feedback to improve our training programs. We appreciate your honest evaluation.

Course Environment



Course Environment

In this course, the following products are preinstalled for the lesson practices:

- JDK 6
- Firefox 3
- Visual VM
- Oracle Solaris Studio
- NetBeans 6.9.1
- Solaris Performance Tools CD 3.0

Additional Resources

Additional Resources

Topic	Website
Education and Training	http://education.oracle.com
Product Documentation	http://www.oracle.com/technology/documentation
Product Downloads	http://www.oracle.com/technology/software
Product Articles	http://www.oracle.com/technology/pub/articles
Product Support	http://www.oracle.com/support
Product Forums	http://forums.oracle.com
Product Tutorials	http://www.oracle.com/technology/obe
Sample Code	http://www.oracle.com/technology/sample_code

ORACLE

How Can I Learn More?

The table in the slide lists various Web resources available to learn more about Java Performance.

JVM and Performance Overview

Chapter 2

JVM and Performance Overview

ORACLE

Objectives

Objectives

After completing this lesson, you should be able to:

- Describe the key features and architecture of the Java Virtual Machine
- Describe common performance principles
- List common application performance problems
- Describe a standard performance methodology
- Describe how to incorporate performance into development

ORACLE

JVM Overview

There are two key components to Java: the programming language and the Java Virtual Machine (JVM).

- Java language
- Architecture of the HotSpot VM
- Key HotSpot VM components



ORACLE

Java Programming Language

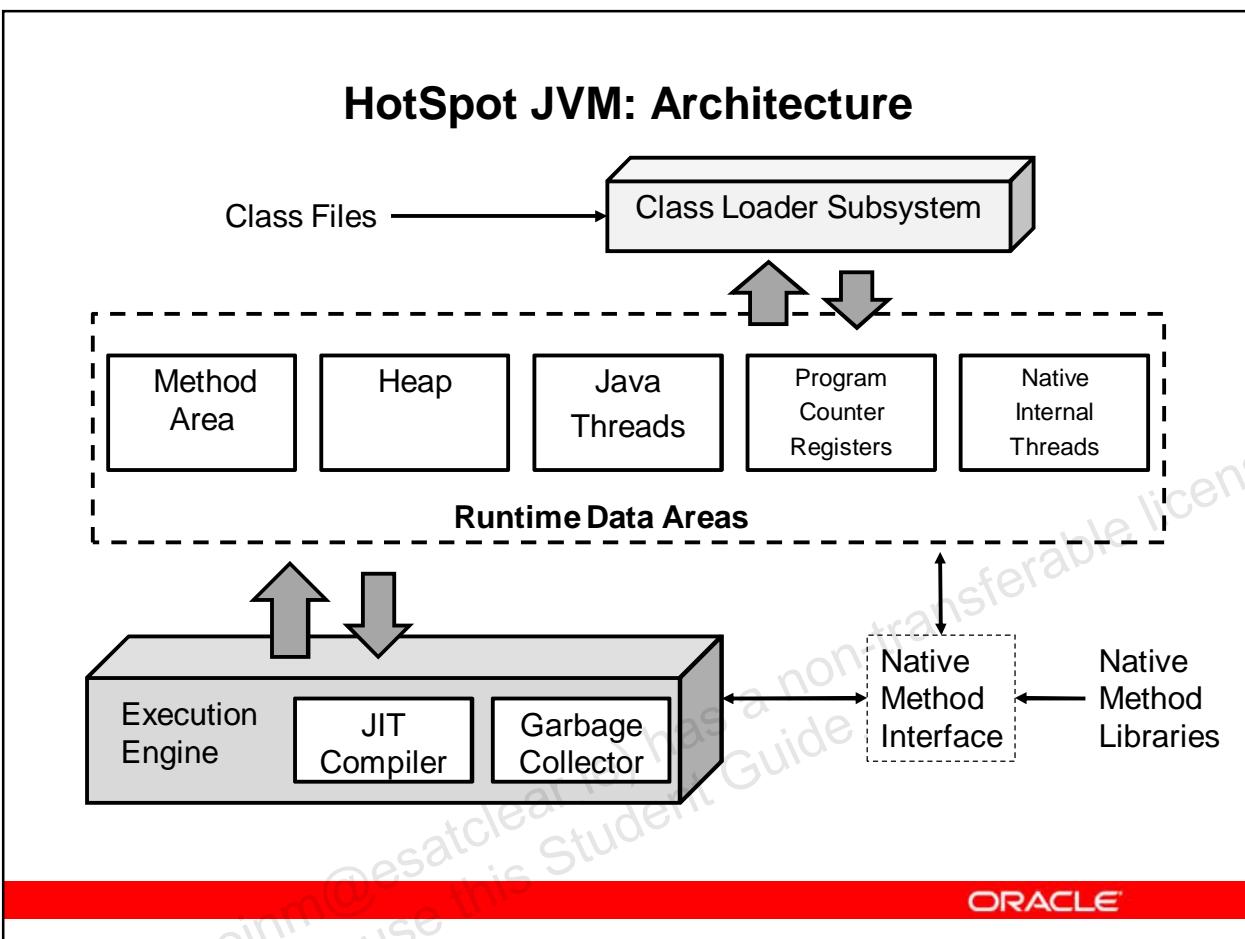
- Java is an object-oriented programming language that improves on C and C++.
- Garbage collection is automatic.
- Java source code is compiled into byte code.
- Byte code is stored in `.class` files.
- `.class` files are loaded into a Java Virtual Machine (JVM) and executed.
- A separate JVM is created for each Java application.

ORACLE

Java was originally developed by James Gosling at Sun Microsystems (a subsidiary of Oracle Corporation) in 1995. Features include object-oriented syntax, automatic garbage collection, and platform-independent program execution.

Java programs are compiled into bytecode and stored in `.class` files. Java applications are run by loading the `.class` files into a Java Virtual Machine (JVM) and executing the byte code. The default JVM included as part of Java 6 is called the *Java HotSpot Virtual Machine*.

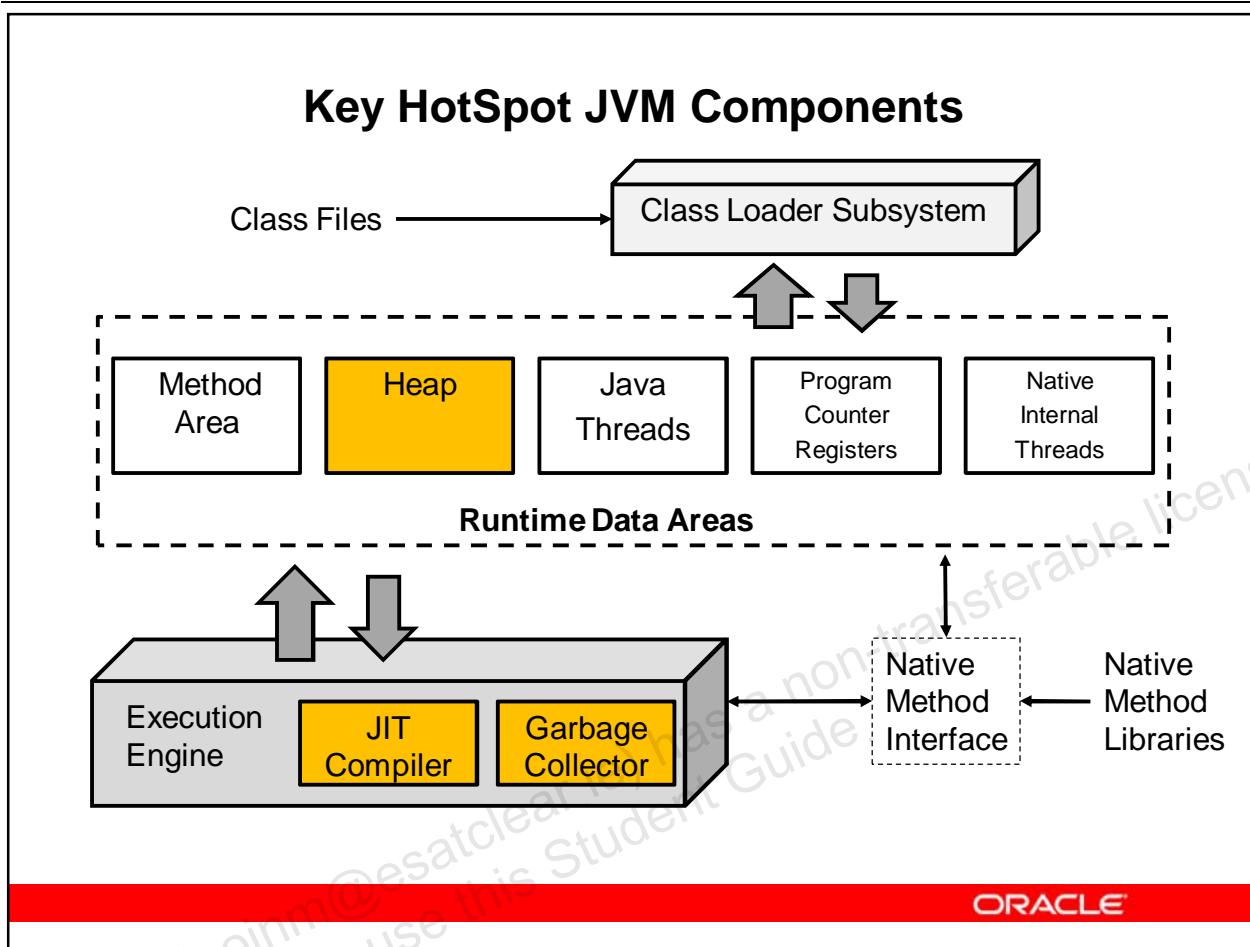
HotSpot JVM: Architecture



The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability. For example, the HotSpot JVM JIT compilers generate dynamic optimizations. In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture.

In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.

Key HotSpot JVM Components



Key HotSpot JVM Components

The JIT compiler — “client” or “server” — is pluggable, as is the choice of garbage collector:

- Serial GC
- Throughput
- Concurrent
- Parallel

The HotSpot JVM Runtime provides services and common APIs to the HotSpot JIT compilers and HotSpot garbage collector. In addition, the HotSpot JVM Runtime provides basic functionality to the JVM, such as a launcher, thread management, Java Native Interface, and so on.

What Is Performance?

What Is Performance?

What are some different aspects of performance?

- Memory footprint
- Startup time
- Scalability
- Responsiveness
- Throughput



ORACLE

There are a number of ways in which you can define performance. Some of these aspects of performance impact the JVM.

Each of these aspects should be a well-defined requirement for an application. In addition, the importance should be prioritized for the application. This clarification is important input to the folks who are tackling the application's performance issues.

Memory Footprint

Memory Footprint

The memory footprint is the amount of memory used by your application and your JVM on a system. Considerations include the following:

- Does the application run on a dedicated system?
- Does the application share the machine with other applications and/or JVMs?
- As memory footprint grows, does it affect Virtual Memory?
 - Accessing data in virtual memory is much slower than accessing data in RAM.



ORACLE

Memory Footprint

It is very important to know the environment and ecosystem in which your application runs. Shared resources can have a significant impact on performance.

Virtual memory swapping should be minimized for any Java application. Consider a scenario where a garbage collection occurs with a large portion of the Java heap in virtual memory. Scanning for referenced objects would take much longer than when the heap is in physical memory. It is very important to configure a system and the Java heap to avoid virtual memory swapping.

Startup Time

Startup Time

Startup time: The time taken for an application to start

- For client applications, this can be very important.
- For server applications, it is less important.
- Time 'til Performance



ORACLE

Time 'til Performance: The time it takes for the JVM to JIT compile the most frequently executed methods. This is an area of interest to many financial services companies if shortly after they launch a JVM or application, the busiest trading hours of the day occur shortly after the launch the application. So, the JVM is trying to JIT compile code and at the same time the application is trying to handle peak load. In other words, the application does not run at peak efficiency until the code is "fully" JIT compiled and the JIT compiler is trying to use CPU cycles while the application is receiving peak load.

In general, the larger the number of classes loaded and the longer and more complex the classpath, the longer it will take to start an application.

Note: Initial start time may not be indicative of an application's performance after Hotspot JVM has had a chance to make optimizations.

Tip: Consider disabling bytecode verification, but only if your company's security policy and the application are not subject to bytecode tampering. Bytecode verification does add overhead to the loading of classes.

Scalability

Scalability

Scalability: How well an application performs as the load on it increases

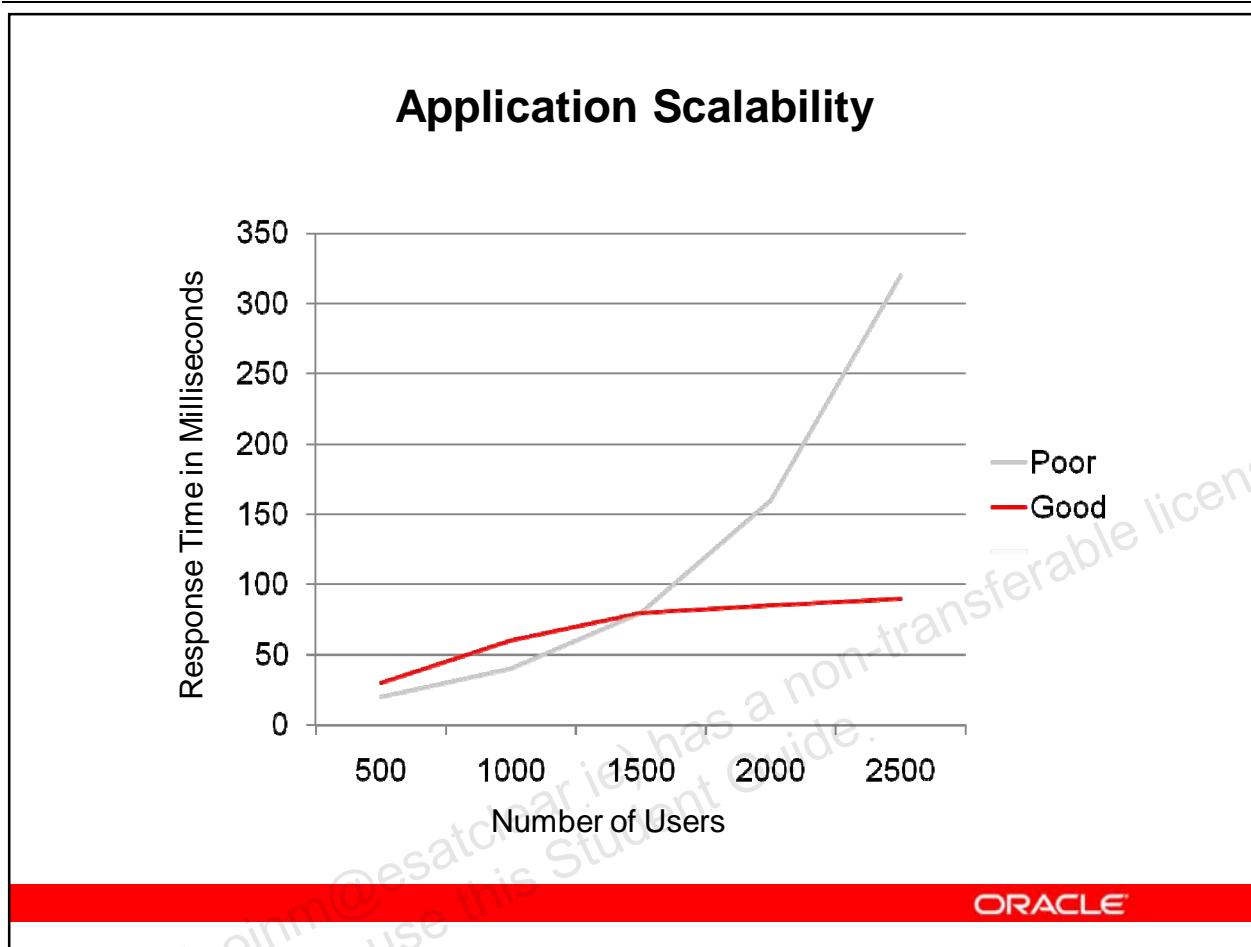
- An application may perform well in development but poorly after it is deployed.
- If an application's response times grow *exponentially* when under load, its scalability is poor.
- If an application's response times grow *linearly* when under load, its scalability is good.
- Scalability can be measured at a number of different levels in a complex system.

ORACLE

Scalability can apply to not only an application but also to many other aspects of a system of which the application is a part.

Tip: It is very important to have a development or qualification environment that replicates production environment situations. This reduces the chance to be caught off-guard with an application that does not scale well.

Application Scalability



Application Scalability

In this example, response times of the application with poor scalability rise exponentially as the number of users increases.

The response times of the application with good scalability rises in a more linear fashion.

Responsiveness

Responsiveness

Responsiveness: How quickly an application or system responds with a requested piece of data

- Examples
 - How quickly a desktop UI responds to an event
 - How fast a website returns a page
 - How fast a database query is returned
- Large pause times are not acceptable.
- The focus is on responding in short periods of time.

ORACLE

Responsiveness can be measured in a number of ways. But the bottom line is that a responsive application returns requested data quickly. Client applications are typically more focused on responsiveness.

Throughput

Throughput

Throughput: Focusing on maximizing the amount of work by an application in a specific period of time

- Examples
 - The number of instructions per second that can be executed
 - The number of jobs that a batch program can complete in an hour
 - The number of database queries that can be completed in an hour
- High pause times are acceptable.
- The focus is on how much work can be done over longer periods of time.

ORACLE

Throughput

Server applications typically focus more on throughput and less on responsiveness.

Performance Focus for This Course

Performance Focus for This Course

This course focuses on:

- Java application performance
- Optimizing the JVM for throughput and/or responsiveness
- Discovering, troubleshooting, and tuning Java performance issues



ORACLE

Performance Issues Covered in This Course

Performance Issues Covered in This Course

Java performance issues covered in this course include:

- Memory leaks
- Lock contention
- Tuning heap size
- Optimizing JVM garbage collection



ORACLE

This course focuses on optimizing Java application performance. Understanding your code and its interaction with the JVM is key to developing high-performing Java applications.

Performance Issues Not Covered in This Course

Performance Issues Not Covered in This Course

Performance issues not covered in this course include:

- Vertical versus horizontal hardware scaling
- Network tuning and design
- Poorly designed AJAX/JavaScript web pages
- Database optimization



ORACLE

Performance is an important topic because of today's complex systems. This course is focused on Java application performance and not as much on hardware-related issues. In addition, parts of your application (like an AJAX web interface) may be very important but are outside the scope of this course.

Performance Methodology

- Performance methodology
 - Monitor
 - Profile
 - Tune
- Definitions
- Incorporating performance methodology into development



ORACLE

Not only will we define these terms, but this is the basic approach for the course and for tuning a Java application. First we monitor, then profile, and finally tune. This process is followed throughout this course.

Performance Monitoring

Performance Monitoring

Definition: An act of non-intrusively collecting or observing performance data from an operating or running application.

- For troubleshooting
 - Identify problems
 - Determine problem characteristics
- For development
 - Test application for meeting requirements
 - Responsiveness or throughput
- Operating system and JVM tools



ORACLE

Non-intrusive: The act of monitoring the application does not materially impact the performance of the application.

In most cases, monitoring is a preventive or proactive action. However, it can be an initial step in a reactive action when troubleshooting a problem. Monitoring helps identify or isolate potential issues without having a severe impact on runtime responsiveness or throughput of an application.

This course focuses on using monitoring techniques and tools to improve application throughput or responsiveness issues. There's less emphasis on using monitoring for troubleshooting or debugging.

Performance Profiling

Performance Profiling

Definition: An act of collecting or observing performance data from an operating or running application.

- Usually more intrusive than monitoring
- Usually a narrower focus than monitoring
- Commonly done in qualification, testing, or development environments.



ORACLE

In general, profiling tends to be a reactive type of activity. However, profiling could be a proactive activity in situations where performance is a well-defined systemic quality or requirement for a target application. Profiling is typically performed during application development and is seldom performed in production environments.

Performance Tuning

Definition: An act of changing tunables, source code and/or configuration attributes for the purposes of improving application responsiveness and/or application throughput.

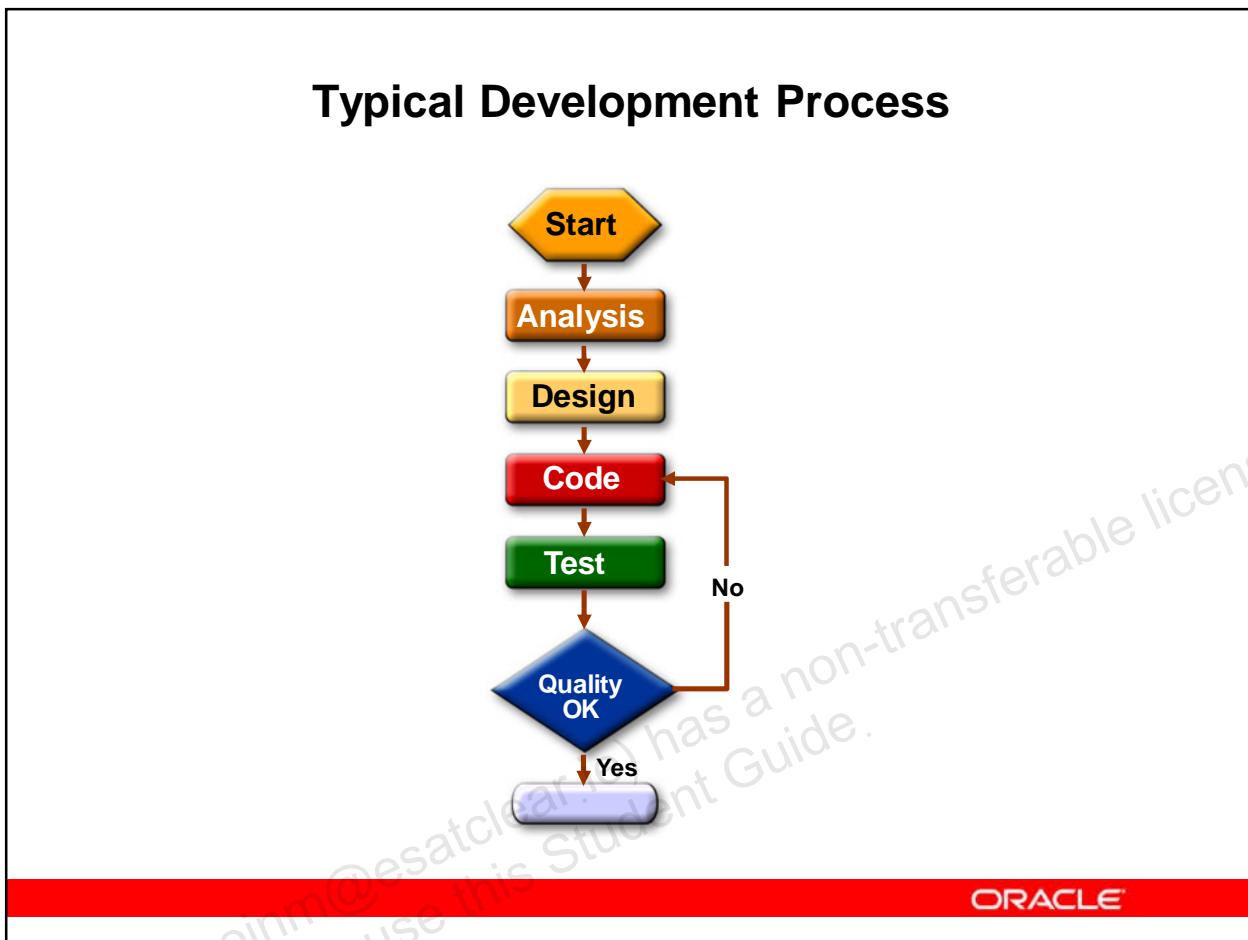
- Relies on performance requirements
- Requires monitoring and/or profiling activities
- Performance goals
 - Responsiveness
 - Throughput



ORACLE

Performance tuning relies on a clear understanding of the performance requirements for an application. For example, does the application have a specific requirement for response time or startup time? Once these requirements are known, monitoring and profiling activities help determine current performance compared to goals. Then this information can be used to start the tuning process.

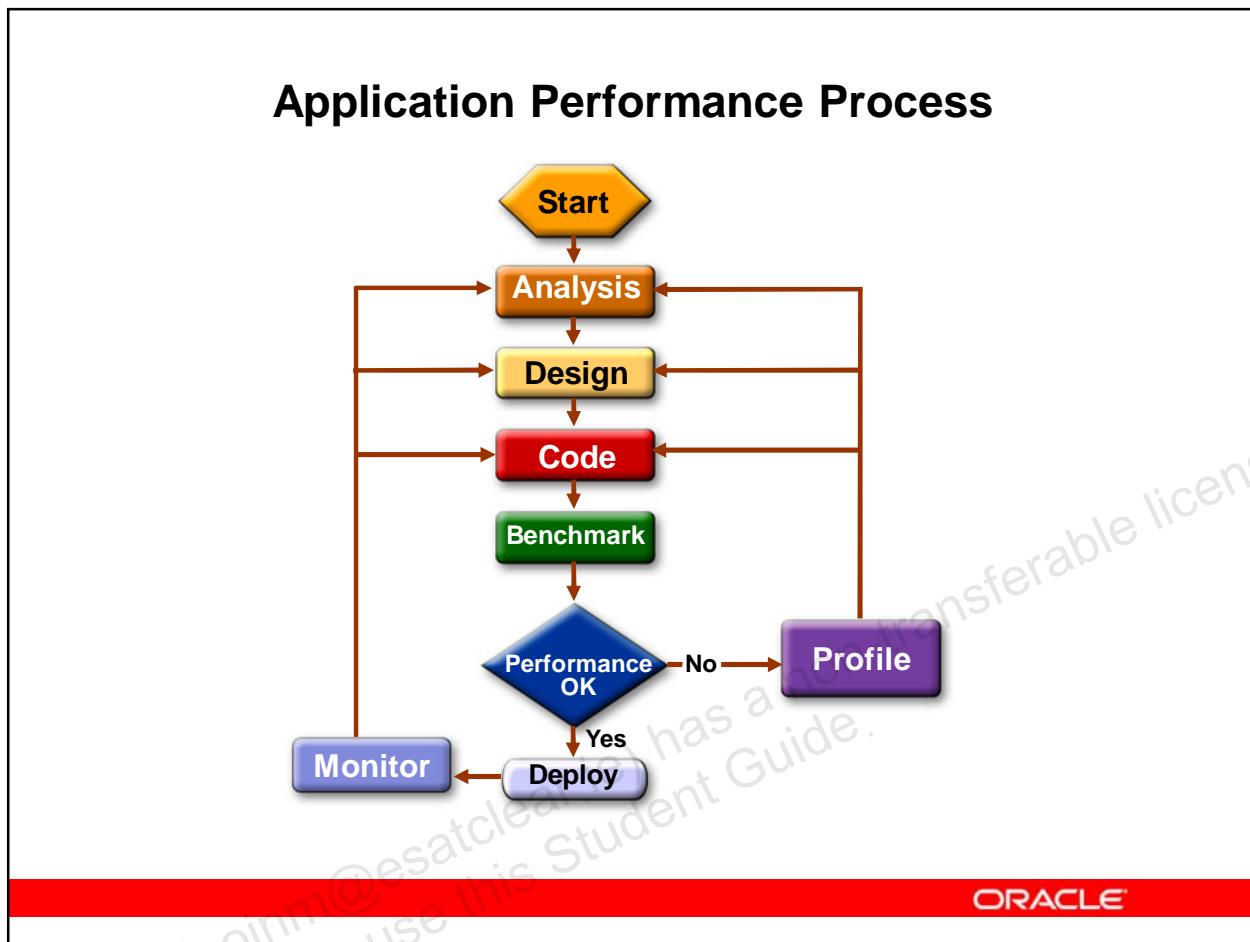
Typical Development Process



The diagram shows a typical development process. In the analysis phase, the problem to be solved is defined and application requirements are created and evaluated. Next, in the design phase, the higher level requirements are turned into more detailed requirements and strategies for the programs and systems that will make up the application. With this information, the coding and testing phases begin. Code is created to fulfill the requirements, then tested, and refactored until the application is complete.

Typically, many applications developed using this traditional model give little attention to performance or scalability requirements until the application is released.

Application Performance Process



A better approach would be to include performance criteria beginning with the analysis phase. Then, performance and scalability requirements could be tested throughout the development of the application. This would result in a better performing application at deployment.

Summary

Summary

In this lesson, you should have learned how to:

- Describe the key features and architecture of the Java Virtual Machine
- Describe common performance principles
- List common application performance problems
- Describe a standard performance methodology
- Describe how to incorporate performance into development

ORACLE

Java Performance Resources

Title: *Java Performance*

Authors: Charlie Hunt and Binu John

ISBN: 0137142528

Release date: August 2011

ORACLE

The original author of this course has written the book described in the slide.

Additional Resources

Additional Resources

- Java Platform Performance: Strategy and Tactics
- Java Developers Journal: Top Ten Performance Problems
- The Server Side: Top Ten Java EE Performance Problems

ORACLE

Additional Resources

The slide lists materials that you might find interesting.

Monitoring Operating System Performance

Chapter 3

Monitoring Operating System Performance

3

ORACLE

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

Objectives

Objectives

After completing this lesson, you should be able to:

- Monitor CPU usage
- Monitor network I/O
- Monitor disk I/O
- Monitor virtual memory usage
- Monitor processes including lock contention

ORACLE

Why Are We Monitoring?

Why Are We Monitoring?

- Get a sense of what the problem is
- Identify the poor performance symptoms
- Based on the symptoms, diagnose the problem



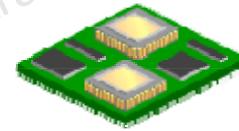
ORACLE

We want to observe the system to get a sense of what the performance problem is. In other words, what "poor performance symptoms" are being exhibited by the application? Based on the symptoms observed, we can determine the next step in the action of diagnosing the performance issue. It's somewhat analogous to going to the doctor when you don't feel well. The doctor has various different tools and instruments to check your performance. Depending on the observed symptoms, the doctor makes recommendations as to the next step to get you back on "performance par."

Monitoring CPU Usage Overview

Monitoring CPU Usage Overview

- Rationale for monitoring CPU usage
 - Get big picture view of CPU demand
 - Get per process measurement of CPU utilization
- Measurements of CPU usage
 - User (usr) time
 - System (sys) time
 - Idle time
 - Voluntary context switching (VCX)
 - Involuntary context switching (ICX)



ORACLE

- **User time:** The amount of CPU time spent in running a process outside the kernel.
- **System time:** The amount of CPU time spent using resources in the operating system kernel.
- **Idle time:** The amount of time the CPU is not being utilized.
- **Voluntary context switch (VCX):** A thread is given a certain amount of CPU time to run. The thread voluntarily yields the CPU after running for its scheduled time.
- **Involuntary context switch (ICX):** A thread is given a certain amount of CPU time to run. The thread is interrupted and yields the CPU before completing its scheduled run time.

CPU Monitoring Performance Indicators

The kind of CPU statistics that may warrant further review:

- High sys/kernel CPU time
- Idle CPU time



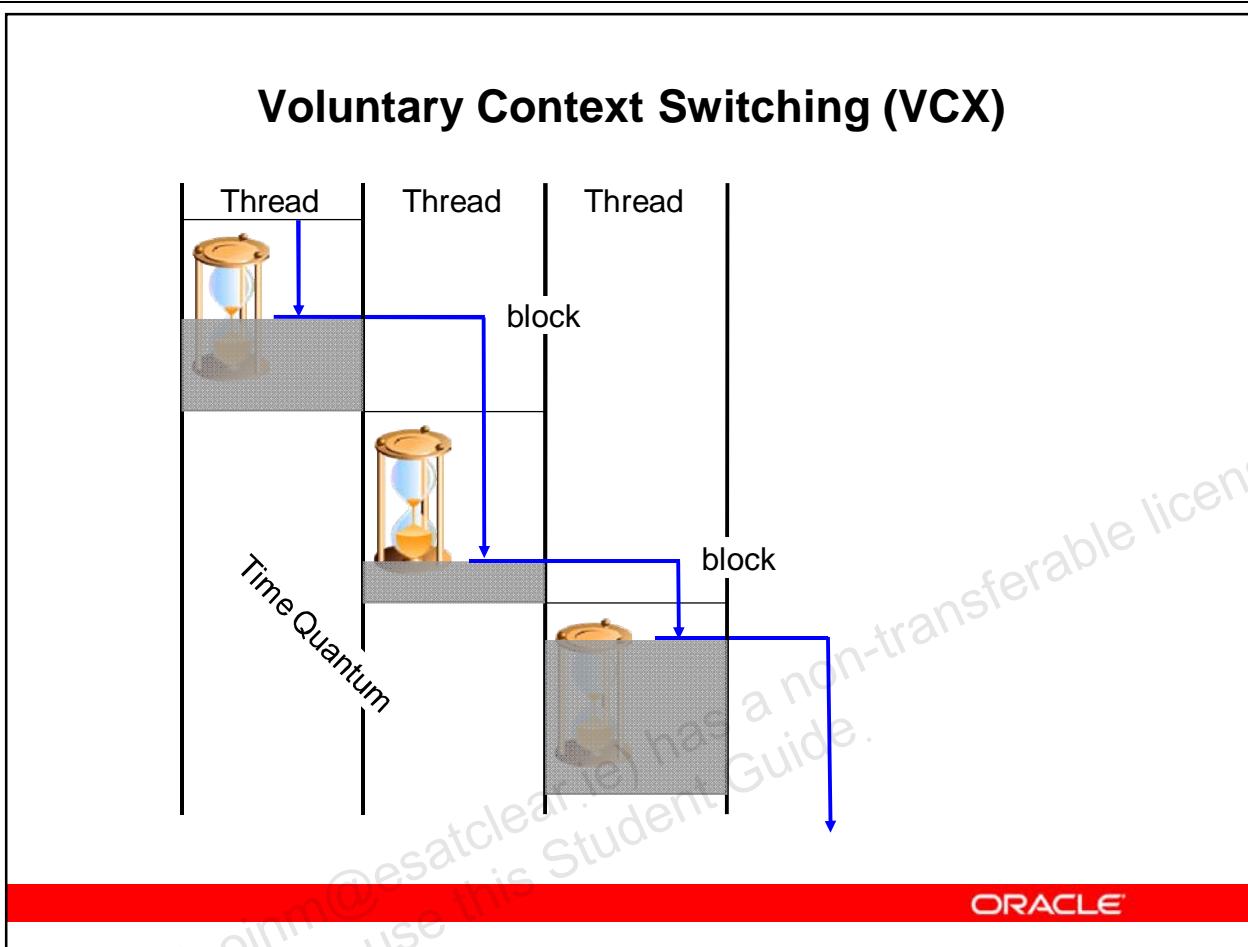
ORACLE

High sys/kernel CPU time indicates a lot of CPU cycles are spent in the kernel. Also, high sys CPU time could indicate shared resource contention, (in other words, locking). Reducing the amount of time spent executing code in the kernel gives the application more CPU time to execute.

Idle CPU Time

On multithreaded applications and multicore systems, idle CPU can be an indicator of an application's inability to scale. Combination of high sys or kernel CPU utilization and idle CPU could indicate shared resource contention as the scalability blocker. This is applicable to all operating systems, that is, Windows, Linux, and Solaris.

Voluntary Context Switching (VCX)



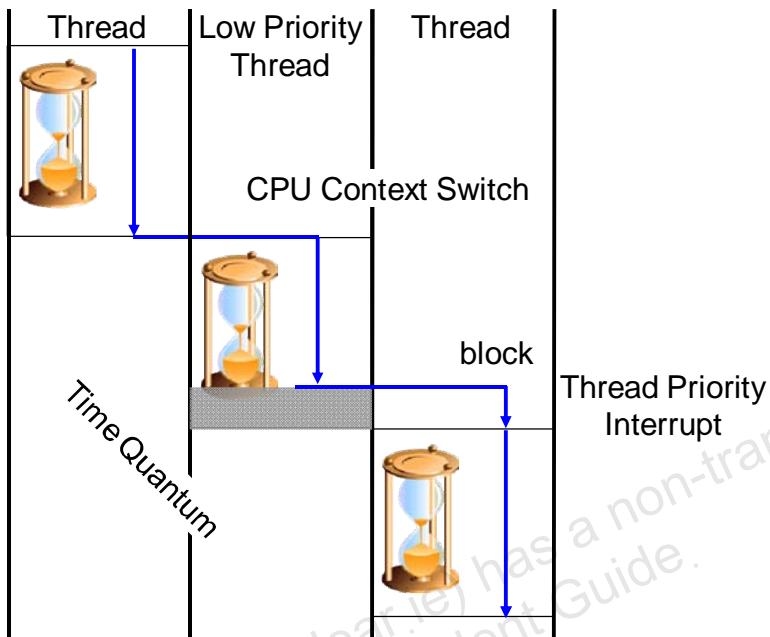
The vertical bars represent a time quantum. A *time quantum* is the amount of time the operating system scheduler makes available for an application thread to run. In this example, an application has multiple threads and the scheduler allocates a time quantum for each thread. Each thread executes until the end of its time quantum and then releases the CPU for the next thread. This demonstrates a voluntary context switching situation.

VCX and Performance

High voluntary context switching can be a result of contention on synchronized method or blocks; or lock implementations available in the Java Concurrent API.

Involuntary Context Switching (ICX)

Involuntary Context Switching (ICX)



ORACLE

In this example, the first thread executes to completion. The second thread is interrupted and is not allowed to finish its time quantum. The thread is involuntarily interrupted. This causes a CPU and a thread context switch. From a CPU perspective, context switches are expensive. It takes thousands of CPU cycles for context switches to occur.

ICX and Performance

From a performance standpoint, this should peak our interest. First, context switches waste a large number of clock cycles. Second, CPUs do all their computations in registers. Ideally, data for these registers is retrieved from the CPU cache. However, if the data is not in the CPU cache, the CPU must waste clock cycles to go get the data from memory. A context switch can cause data fetching from memory. So the cost of doing thread context switches is expensive and can be the source of performance issues. An application that does a large number of involuntary context switches requires investigation.

There are some applications (most often seen in OLTP/database systems) which can benefit by switching to the Solaris FX (fixed) scheduler as a means to reduce context switching. For Solaris, use `priocntl` to set FX scheduling:

```
$ priocntl -c FX [ -s <PID> [<PID> ... ] ] -e <java cmd and args>
```

Man Page Documentation for `priocntl`: <http://download.oracle.com/docs/cd/E19253-01/816-5165/6mbb0m9om/index.html>

Tools For Monitoring CPU Usage

Tools For Monitoring CPU Usage

Tools to monitor CPU utilization

- `vmstat` (Solaris and Linux)
- `mpstat` (Solaris)
- `prstat` (Solaris)
- `top` (Linux; prefer `prstat` on Solaris)
- Task Manager (Windows)
- Performance Monitor (Windows)
- Windows Resource Manager (Windows Server)
- Gnome System Monitor (Linux and Solaris)
- `cpubar` (Solaris: Performance Tools CD)
- `iobar` (Solaris: Performance Tools CD)
- DTrace (Solaris)

ORACLE

CPU Tools

- `prstat`: Similar to `top` on Linux. On Solaris `prstat` is less intrusive than `top`.
- `Gnome System Monitor`: A graphical representation of CPU utilization on Linux. Also works on Solaris if Gnome desktop is installed.
- `cpubar`: A graphical representation of CPU utilization on Solaris
- `iobar`: A graphical representation of I/O and CPU utilization.

CPU Usage: vmstat

CPU Usage: vmstat																						
kthr			memory			page			disk			faults			cpu							
r	b	w	swap	free	mf	pi	po	fr	de	sr	f0	s0	s1	s2	in	sy	cs	us	sy	id		
0	0	0	659620	168400	3	19	6	2	3	0	4	-0	1	-0	1	385	886	432	3	2	95	
0	0	0	129404	26456	2	87	10	0	0	0	0	0	5	0	0	2320	247894	4745	37	20	44	
0	0	0	127920	25524	23	395	90	0	0	0	0	0	27	0	1	1844	241699	4280	45	21	35	
0	0	0	126380	24204	1	92	0	0	0	0	0	0	0	0	0	2455	283529	4916	34	19	48	
0	0	0	126380	24208	1	93	0	0	0	0	0	0	0	0	0	2469	288083	4933	33	19	49	
0	0	0	126376	24204	1	79	0	0	0	0	0	0	0	0	0	2171	241732	4294	33	20	47	
0	0	0	126344	24172	1	78	1	0	0	0	0	0	16	0	3	2202	242373	4782	36	20	44	
0	0	0	126544	24372	1	85	0	0	0	0	0	0	0	0	0	2127	263456	4928	42	21	37	
0	0	0	126504	24332	1	92	0	0	0	0	0	0	0	0	0	6	2498	284583	5041	33	19	48
0	0	0	126112	23940	1	144	2	0	0	0	0	0	0	0	0	2	2027	273269	5311	46	20	34
0	0	0	125252	22952	3	97	53	0	0	0	0	10	0	14	2091	231099	4423	35	20	44		

\$ vmstat 5

ORACLE

To launch, enter: \$ vmstat 5

Use vmstat to obtain time-based (for example every 5 seconds) samples of CPU usage. These samples provide an overview of the percentage of CPU user time, system time, and idle time.

- **User time:** The percentage of CPU time spent on servicing applications.
- **System time:** The percentage of CPU time in kernel-related tasks.
- **Idle time:** The amount of time the CPU is idle.

In this vmstat example, there is a large amount of user and system activity on the system being monitored.

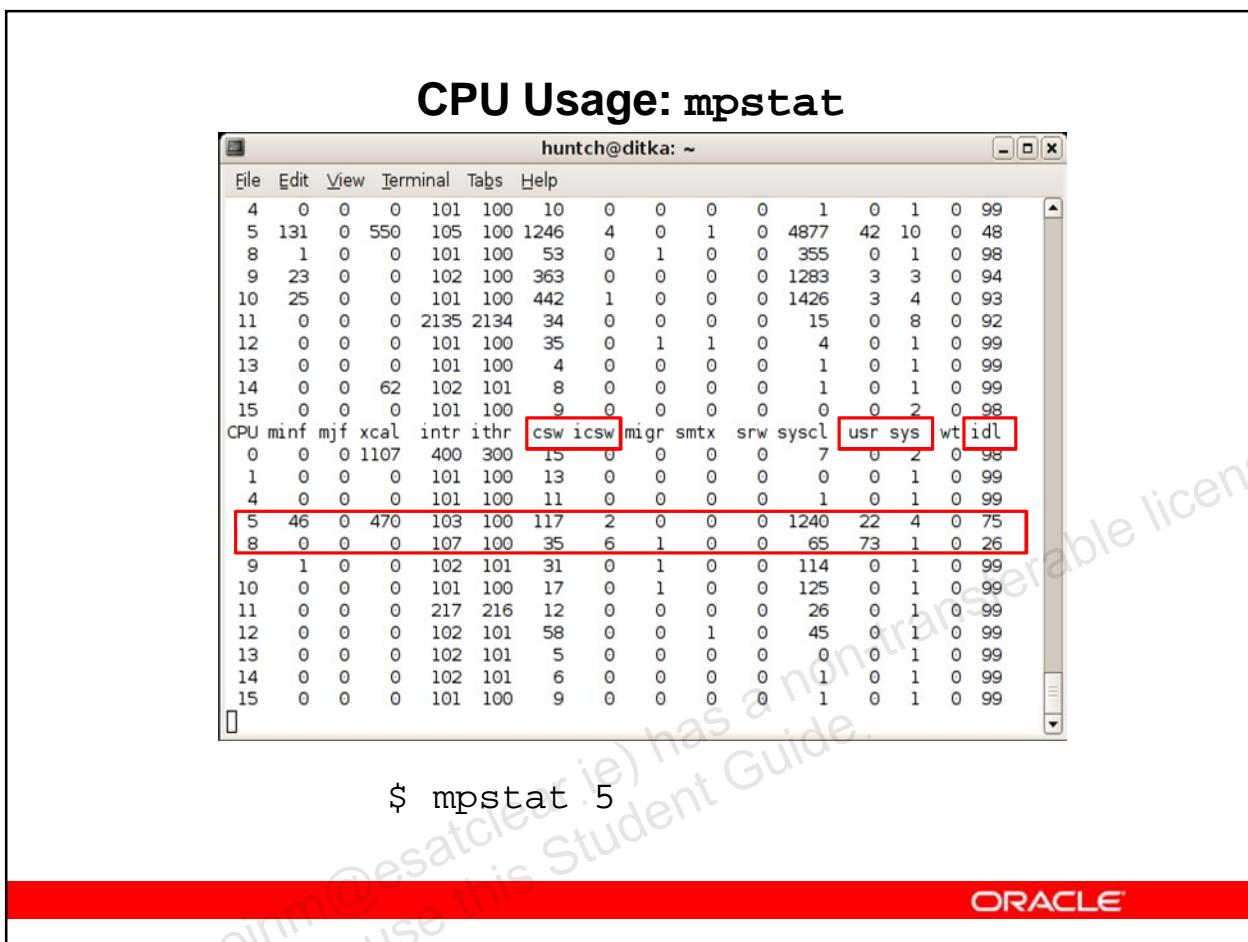
Column Legend

us: user time

sy: system time

id: idle time

CPU Usage: mpstat



To launch, enter: \$ mpstat 5

Use mpstat to capture CPU usage statistics per virtual processor. A *virtual processor* is defined as a hardware thread. A hyperthreaded Intel Xeon processor has two of these hardware threads per core. A dual core, hyperthreaded Intel Xeon processor has four hardware threads, two per core. Therefore, it has four virtual processors and mpstat will report four lines, one for each virtual processor. An UltrSPARC T-series processor has both multiple cores and multiple hardware threads per core. More specifically, an UltrSPARC T2 processor has eight cores and eight hardware threads per core. Therefore, it has 64 hardware threads and reports 64 cores and thus 64 lines of output. For more information about Oracle UltraSparc T-Series, refer to: <http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/index.html>

The screenshot shows usage statistics for a 12 core system. If a machine has multiple cores, each core is shown as a processor. Data is shown as events per second unless otherwise noted. See man page for detailed information.

Column Legend

usr: user time; **sys:** system time; **idl:** idle time; **csw:** context switches; **icsw:** involuntary context switches

CPU Usage: prstat

CPU Usage: prstat

PID	USERNAME	SIZE	RSS	STATE	PRT	NICE	TIME	CPU	PROCESS/NLWP
2647	huntrch	674M	632M	cpu1	50	0	0:08:02	38%	java/18
778	huntrch	49M	23M	sleep	59	0	0:00:25	0.3%	Xorg/1
1136	root	103M	22M	sleep	59	0	0:00:35	0.1%	java/19
2264	huntrch	3924K	1928K	sleep	59	0	0:00:01	0.1%	cpubar/1
1122	noaccess	76M	15M	sleep	59	0	0:00:32	0.1%	java/24
1708	huntrch	110M	9800K	sleep	59	0	0:00:24	0.1%	mixer_applet2/1
525	root	3948K	392K	sleep	59	0	0:00:09	0.1%	vmware-guestd/1
2251	huntrch	55M	10M	sleep	59	0	0:00:15	0.0%	gnome-terminal/2
2653	huntrch	4636K	1096K	cpu0	59	0	0:00:00	0.0%	prstat/1
1198	huntrch	51M	752K	sleep	59	0	0:00:20	0.0%	vmware-user/1
2841	huntrch	47M	10M	sleep	59	0	0:00:00	0.0%	screenshot/1
749	root	4544K	1864K	sleep	59	0	0:00:01	0.0%	nscd/23
1702	huntrch	107M	9596K	sleep	59	0	0:00:02	0.0%	clock-applet/1
9	root	8728K	1252K	sleep	59	0	0:00:38	0.0%	svc.configd/13
1589	huntrch	110M	11M	sleep	59	0	0:00:01	0.0%	gnome-panel/1
604	root	6232K	2568K	sleep	59	0	0:00:01	0.0%	intrd/1
2840	huntrch	42M	4696K	sleep	59	0	0:00:00	0.0%	script-fu/1
2656	huntrch	64M	24M	sleep	59	0	0:00:06	0.0%	gimp-2.3/5
2247	root	4760K	1364K	sleep	59	0	0:00:01	0.0%	devfsadm/8
1592	huntrch	140M	21M	sleep	59	0	0:00:09	0.0%	nautilus/1
1646	huntrch	14M	3888K	sleep	59	0	0:00:00	0.0%	gnome-vfs-daemon/2
Total: 80 processes, 254 TwpS, load averages: 5.45, 3.08, 1.33									

```
$ prstat
```

ORACLE

To launch, enter: \$ prstat

CPU%: The amount of CPU time being used by a process

prstat shows the overall CPU time measured per process

The Java application at the top is using 38% of the CPU. The number 18 tells us how many lightweight processes are in that Java application. Lightweight processes are essentially threads. So the Java application has 18 threads in this example.

Note: The number of threads also includes the number of both Java application threads and internal JVM threads.

CPU Usage: prstat -m

CPU Usage: prstat -m

PID	USERNAME	USR	SYS	TRP	TFL	DFL	LCK	SLP	LAT	VCX	TCX	SCL	STG	PROCESS/LWPID
6506	huntrch	49	24	0.2	0.0	0.0	1.1	19	7.3	4K	1K	.13	0	java/2
6505	huntrch	11	7.1	0.0	0.0	0.0	0.3	74	7.5	4K	162	87K	0	java/2
766	huntrch	3.0	0.5	0.0	0.0	0.0	0.0	94	2.1	266	131	1K	64	Xorg/1
6506	huntrch	1.2	0.6	0.0	0.0	0.0	0.0	98	0.0	0.3	595	10	1K	0 java/3
990	huntrch	1.1	0.1	0.0	0.0	0.0	0.0	97	1.5	8	64	291	0	wnck-applet/1
6506	huntrch	0.5	0.1	0.0	0.0	0.0	0.0	97	0.0	2.3	195	3	196	0 java/5
6506	huntrch	0.3	0.2	0.0	0.0	0.0	0.0	99	0.0	0.1	193	7	385	0 java/4
972	huntrch	0.3	0.2	0.0	0.0	0.0	0.0	99	0.2	97	0	549	0	metacity/1
6190	huntrch	0.2	0.3	0.0	0.0	0.0	0.0	99	0.5	165	1	647	0	java/21
4919	huntrch	0.4	0.0	0.0	0.0	0.0	0.0	100	0.1	27	0	78	0	soffice.bin/1
6417	huntrch	0.3	0.0	0.0	0.0	0.0	0.0	100	0.0	4	0	8	0	thunderbird/-1
6505	huntrch	0.2	0.1	0.0	0.0	0.0	0.0	100	0.0	0.0	44	0	204	0 java/3
6512	huntrch	0.2	0.1	0.0	0.0	0.0	0.0	100	0.0	17	23	111	0	screenshot/1
6506	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	99	0.4	151	38	87	0	java/9
1095	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.0	53	0	211	0	firefox-bin/1
6190	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	99	0.4	156	24	87	0	java/14
6505	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	99	0.0	0.6	32	0	32	0 java/5
980	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	99	0.7	12	1	46	0	nautilus/1
6494	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	100	0.0	13	0	58	0	gimp-2.4/1
6505	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.2	156	1	87	0	java/9
1095	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.0	2.0	104	0	150	0 firefox-bin/3
Total: 99 processes, 321 twpes, load averages: 1.60, 1.28, 0.64														

\$ prstat -m

ORACLE

To launch, as superuser enter: \$ prstat -m

The -m option provides microstate information. Microstate information provides VCX and ICX data.

The application at the top of the screenshot is showing a lot of usr and sys CPU time. Given the high number of voluntary context switching and high SYS CPU time, the application may be experiencing lock contention. It is showing symptoms consistent with high lock contention.

Column Legend

USR: user time

SYS: system time

VCX: voluntary context switches

ICX: involuntary context switches

CPU Usage: prstat -Lm

PID	USERNAME	USR	SYS	TRP	TFL	DFL	LCK	SLP	LAT	VCX	ICX	SCL	SIG	PROCESS/LWPID
2647	hutch	27	0.0	0.1	0.0	0.0	52	0.0	21	7	113	28	0	java/3
2647	hutch	20	0.1	0.1	0.0	0.0	48	0.0	32	48	218	103	0	java/46
2647	hutch	19	0.0	0.1	0.0	0.0	38	0.0	44	60	223	115	0	java/45
2647	hutch	18	0.0	0.0	0.0	0.0	48	0.0	34	70	114	128	0	java/40
2647	hutch	18	0.1	0.1	0.0	0.0	44	0.0	38	52	106	115	0	java/39
2647	hutch	18	0.1	0.1	0.0	0.0	38	0.0	44	69	152	153	0	java/43
2647	hutch	17	0.0	0.1	0.0	0.0	44	0.0	39	54	194	135	0	java/41
2647	hutch	17	0.0	0.1	0.0	0.0	43	0.0	40	60	166	94	0	java/42
2647	hutch	16	0.0	0.1	0.0	0.0	39	0.0	45	70	160	126	1	java/44
2647	hutch	0.1	0.1	0.0	0.0	0.0	98	2.0	144	0	87	0	java/10	
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/9
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/8
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/7
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/6
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/5
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/4
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/2
2647	hutch	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	0	0	0	0	java/1

Total: 1 processes, 18 lwps, load averages: 5.61, 3.66, 1.68

\$ prstat -Lm

ORACLE

To launch, enter: \$ prstat -Lm

The -L option shows information about each lightweight process or thread. In the example, the number to the right of each Java thread is its ID.

How to map that Java thread ID to a particular piece of Java code in an application? That will be covered in the next couple of slides.

Column Legend

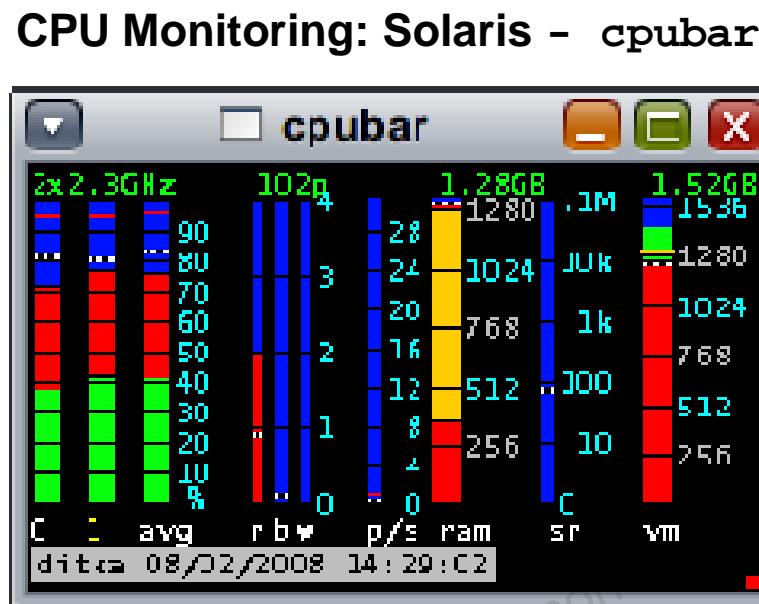
USR: user time

SYS: system time

VCX: voluntary context switches

ICX: involuntary context switches

CPU Monitoring: Solaris - cpubar



- Available on Solaris Performance Tools 3.0 CD, or can be downloaded from: <http://www.schneider4me.de/ToolsCD-v3.0.iso.zip>

ORACLE

Graphical tools often provide more information faster than command-line tools.

On the far left, the 0 and 1 identify the two CPUs on this system. The average bar is the average of the two CPUs. Dashed lines are average CPU utilization.

Bar Colors

Green: User

Red : System

Blue : Idle

Map Lightweight Processes to Java Threads

Map Lightweight Processes to Java Threads

- Lightweight Processes (LWP) are essentially threads in a Java application.
- Use Solaris `prstat` to locate threads using the most CPU
 - `prstat -Lm`
- Use HotSpot's `jstack` to dump thread information
 - `jstack 2874 > temp.txt`
- Find the LWPID in `jstack` output.
 - Convert LWPID from decimal to hex.
 - Search `jstack` output for `nid=0xHexValue`.



ORACLE

Use Solaris `prstat -Lm` to locate the LWP ID consuming the most CPU (usr or sys).

`prstat -Lm`

This will list microstate information including all the threads in an application. Find the thread using the most CPU (usr or sys).

1. Note the process ID (leftmost column) and the thread or LWPID (rightmost column).
2. Use HotSpot's `jstack` to dump thread information about the running application. For example:
`jstack 2874 > temp.txt`
3. Convert the LWPID from step 1 into hex. Search `temp.txt` for `nid=0xHexValue`. You should find the name of the thread in the `jstack` output.

Monitoring Network I/O Overview

Monitoring Network I/O Overview

- Data of interest
 - Network utilization in terms of Transaction Control Protocol (TCP) statistics and established connections
- Tools to monitor network I/O
 - netstat (Solaris & Linux)
 - Performance Monitor (Windows)
 - DTrace (Solaris)
 - nicstat (Solaris – Performance Tools CD)
 - tcptop (DTrace Toolkit)



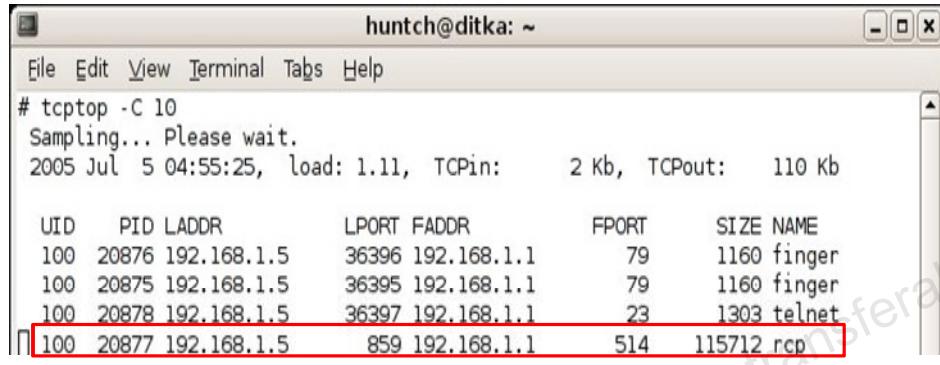
ORACLE

If an application is using the network and/or disk, it realizes the best performance by minimizing the number of network and disk interactions. So, if an application accesses the network and disks, ask yourself: Is the application suffering from too much network and/or disk interaction? Monitoring the application gives you a sense of network and disk activity.

Network I/O: Using tcptop

Network I/O: Using tcptop

- tcptop can show per process TCP statistics:



A screenshot of a terminal window titled "hutch@ditka: ~". The window displays the output of the command "# tcptop -C 10". The output shows network traffic statistics with columns for UID, PID, LADDR, LPORT, FADDR, FPORT, SIZE, and NAME. Several processes are listed, including finger, telnet, and rcp. The row for rcp is highlighted with a red box.

UID	PID	LADDR	LPORT	FADDR	FPORT	SIZE	NAME
100	20876	192.168.1.5	36396	192.168.1.1	79	1160	finger
100	20875	192.168.1.5	36395	192.168.1.1	79	1160	finger
100	20878	192.168.1.5	36397	192.168.1.1	23	1303	telnet
100	20877	192.168.1.5	859	192.168.1.1	514	115712	rcp

- The screen capture shows rcp generating 115 kb of traffic.

ORACLE

tcptop can identify who is doing what and how much traffic are they generating. If this were a production system, you would want to know who is running rcp on this system and why.

Network I/O: Using nicstat

Network I/O: Using nicstat

Time	Int	rKb/s	wKb/s	rPk/s	wPk/s	rAvs	wAvs	%Util	Sat
12:33:04	hme0	1.51	4.84	7.26	10.32	213.03	480.04	0.05	0.00
12:33:05	hme0	0.20	0.26	3.00	3.00	68.67	90.00	0.00	0.00
12:33:06	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:07	hme0	0.14	0.52	2.00	6.00	73.00	88.00	0.01	0.00
12:33:08	hme0	0.24	0.36	3.00	4.00	81.33	92.00	0.00	0.00
12:33:09	hme0	2.20	1.77	16.00	18.00	140.62	100.72	0.03	0.00
12:33:10	hme0	0.49	0.58	8.00	9.00	63.25	66.00	0.01	0.00
12:33:11	hme0	12.16	1830.38	185.06	1326.42	67.26	1413.06	15.09	0.00
12:33:12	hme0	19.03	3094.19	292.88	2229.11	66.53	1421.40	25.50	0.00
12:33:13	hme0	19.55	3151.87	301.00	2270.98	66.50	1421.20	25.98	0.00
12:33:14	hme0	11.99	1471.67	161.07	1081.45	76.25	1393.49	12.15	0.00
12:33:15	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:16	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:17	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00

nicstat 1

ORACLE

To run nicstat with a one-second interval, using the following command:

```
$ nicstat 1
```

The nicstat command displays network statistics. Note that wAvs, write average size, during four intervals is about 1420 bytes, which is the Maximum Transmission Unit (MTU) size. Because 1420 bytes is the maximum MTU for TCP, there was an application using a lot of bandwidth during the middle of this monitoring session.

Monitoring Disk I/O Overview

- Data of interest
 - Number of disk accesses
 - Latency and average latencies
- Tools to monitor disk I/O
 - iostat (Solaris & Linux)
 - iotop (Solaris & Linux)
 - pidstat (Linux)
 - Performance Monitor (Windows)
 - DTrace (Solaris)
 - iobar (Solaris – Performance Tools CD)
- Disk caches



ORACLE

Number of disk accesses: Each disk access you make is a costly event. You want to minimize the number of times you go to disk and read. The round trip time is very expensive.

Latency and average latencies: This is the time it takes to find something on the disk.

Disk I/O: iotop

Disk I/O: iotop

```
hutch@ditka: ~
File Edit View Terminal Tabs Help
hutch@ditka:/$ iotop
2007 Jun 11 00:39:03, load: 1.10, disk_r: 5302 Kb, disk_w: 20 Kb
      UID      PID  PPID  CMD          DEVICE  MAJ MIN D      DISKTIME
        0        0      0  sched       cmdk0   102   0 W        532
        0        0      0  sched       cmdk0   102   0 R    245398
        0  27758  20320  find       cmdk0   102   0 R  3094794
```

- **iotop** reporting at a 5-second interval
- **DISKTIME** reported in microseconds

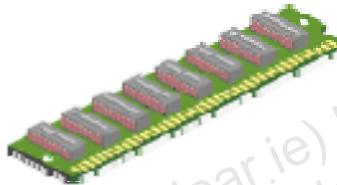
ORACLE

In this example, the *find* command is keeping disk cmdk0 busy almost 60% of time during the 5-second interval. If a Java application were doing something similar, we would want to investigate the reason why.

Monitoring Virtual Memory: Overview

Monitoring Virtual Memory: Overview

- Observe paging to identify swapping
 - Pages in (pi)
 - Pages out (po)
 - Scan rate (sr)
- Fixing the swapping problem
- Why is swapping bad for a Java application?
 - Want to explain? (Without looking at your notes)



ORACLE

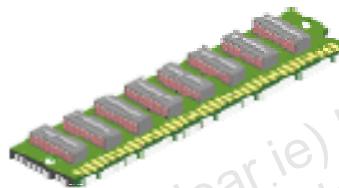
Why is swapping bad for a Java application?

The garbage collector (GC) manages free memory in the JVM. The JVM uses system memory to store the heap and that is what the GC traverses. If the system memory is being swapped to disk, then GC activities are going to be an order of magnitude slower than if the heap were in memory. This is why swapping is bad for a Java application. Garbage collections will be terribly slow.

Virtual Memory Tools

Virtual Memory Tools

- Tools to monitor memory paging and usage
 - vmstat (Solaris & Linux)
 - prstat (Solaris)
 - top (Linux – prefer prstat on Solaris)
 - Performance Monitor (Windows)
 - DTrace (Solaris)
 - cpubar (Solaris – Performance Tools CD)
 - meminfo (Solaris – Performance Tools CD)



ORACLE

Virtual Memory: vmstat

Virtual Memory: vmstat

kthr	memory	page			disk			faults			cpu										
		r	b	w	swap	free	re	mf	pi	po	fr	de	sr	m0	m1	m2	m3	in	sy	cs	us
0	0	0	11532280	3322672	4	18	6	0	0	0	0	1	1	1	0	1599	491	147	1	1	98
0	0	0	11406040	3186248	0	7	0	0	0	0	0	0	0	0	0	1567	348	138	0	1	99
0	0	0	11406832	3187048	0	1	0	0	0	0	0	19	19	19	0	1737	332	153	0	1	99
0	0	0	11408128	3188344	0	1	0	2	2	0	0	0	0	0	0	1573	336	134	0	1	99
0	0	0	11407912	3188128	0	1	0	0	0	0	0	0	0	0	0	1567	331	139	0	1	99
0	0	0	11407464	3187696	2	21	0	0	0	0	0	17	17	17	0	1717	331	157	0	1	99
0	0	0	11407472	3187704	0	1	0	0	0	0	0	0	0	0	0	1563	310	149	0	1	99
0	0	0	11407520	3187744	0	2	0	0	0	0	0	0	0	0	0	1565	317	144	0	1	99
0	0	0	11407504	3187720	0	1	0	0	0	0	0	0	0	0	0	1561	308	138	0	1	99
0	0	0	11407016	3187216	32	196	0	0	0	0	0	0	0	0	0	2764	4997	1381	1	2	97
0	0	0	11396280	3176232	43	351	0	0	0	0	0	0	0	0	0	5036	14128	3772	9	4	87
0	0	0	11399760	3179872	25	65	0	0	0	0	0	8	8	8	0	1740	2450	314	16	2	82
0	0	0	11407392	3187600	0	1	0	0	0	0	0	0	0	0	0	1579	332	150	17	1	82
0	0	0	11407352	3187568	0	3	0	0	0	0	0	0	0	0	0	1572	313	138	17	1	82
0	0	0	11407312	3187536	0	1	0	0	0	0	0	0	0	0	0	1573	324	143	12	1	86
0	0	0	11407280	3187504	0	1	0	2	2	0	0	0	0	0	0	1564	316	142	0	1	99
0	0	0	11407248	3187464	0	1	0	0	0	0	0	0	0	0	0	1562	319	147	0	1	99
0	0	0	11407216	3187440	0	1	0	0	0	0	0	3	3	3	0	1593	306	140	0	1	99
kthr	memory	page			disk			faults			cpu										
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	m0	m1	m2	m3	in	sy	cs	us	sy	id
0	0	0	11407408	3187712	0	4	0	0	0	0	0	0	0	0	0	1583	557	179	0	1	98

\$ vmstat 5

ORACLE

Column Legend

pi: pages in

po: pages out

sr: page scan rate

Scan Rate: The rate at which the operating system is scanning for free pages

Virtual Memory: Swapping Example

Virtual Memory: Swapping Example

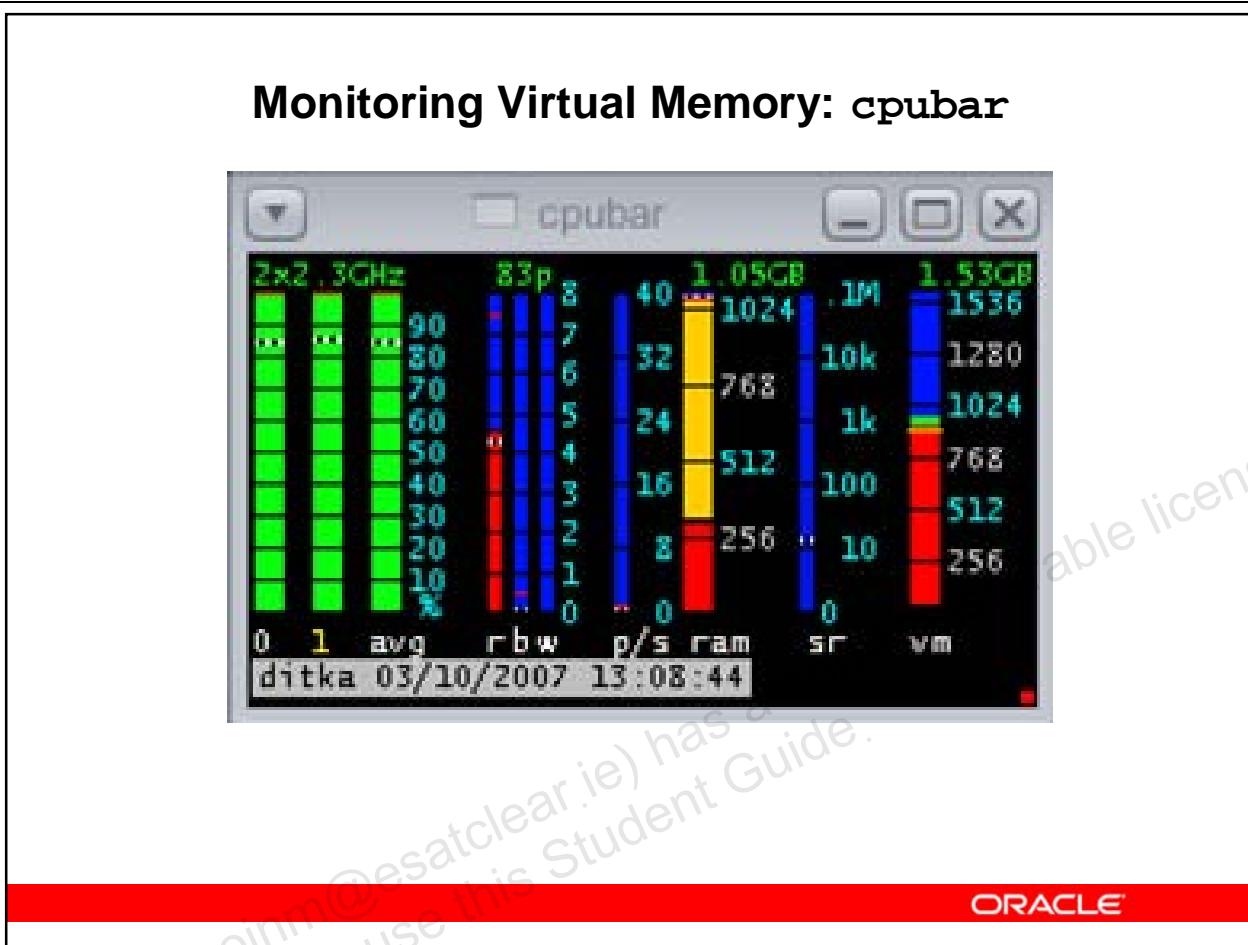
```
hutch@ditka: ~
File Edit View Terminal Tabs Help
hutch@ditka:~$ vmstat 5
kthr      memory          page          disk          faults          cpu
r b w   swap free  mf pi po fr de sr f0 s0 s1 s2  in  sy  cs us sy id
1 0 0 499792 154720 1 1697 0 0 0 0 0 0 0 0 0 12 811 612 1761 90 7 4
1 0 0 498856 44052 1 3214 0 0 0 0 0 0 0 0 0 12 1290 2185 3078 66 18 15
3 0 0 501188 17212 1 1400 2 2092 4911 0 37694 0 53 0 12 5262 3387 1485 52 27 21
1 0 0 500696 20344 26 2562 13 4265 7553 0 9220 0 66 0 12 1192 3007 2733 71 17 12
1 0 0 499976 20108 3 3146 24 3032 10009 0 10971 0 63 0 6 1346 1317 3358 78 15 7
1 0 0 743664 259080 61 1706 70 8882 10017 0 19866 0 178 0 52 1213 595 688 70 12 18
```

- Data of interest
 - pi – pages in; po – pages out; sr – page scan rate
 - Watch for high scan rate (see rows 3 to 6), or increasing trend. Low scan rate is ok if they occur infrequently.

ORACLE

On Solaris, as a system begins to swap to virtual memory, the page in, page out, and scan rate increase. Page in and page out with no scan rate activity, or a small amount of scan rate that terminates very quickly, is ok.

Monitoring Virtual Memory: cpubar



Data of interest

p/s: pages per second

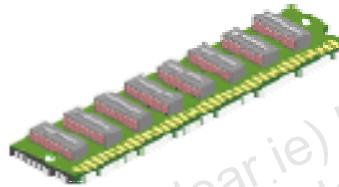
sr: scan rate

Watch for high scan rate or an increasing trend. Low scan rate is ok if not very frequent.

A high page rate with a low scan rate is acceptable.

Virtual Memory: Fixing the Swapping Problem

- Have smaller Java heap sizes.
- Add physical memory.
- Reduce number of applications running on the machine.
- Any one, or any combination of the above will help.



ORACLE

Monitoring Processes Overview

Monitoring Processes Overview

- Data of interest
 - Footprint size
 - Number of threads and thread state
 - CPU usage
 - Runtime stack
 - Context switches
 - Lock contention



ORACLE

Process questions

Why are footprint size, number of threads, thread state, lock contention, and context switching important to monitor?

What do lock contention and context switching look like on Solaris?

How can you find the lock or locks causing problems?

How can you address the thread context switching problem?

Process Monitoring Tools

- ps (Solaris & Linux)
- vmstat (Solaris & Linux)
- mpstat (Solaris)
- prstat (Solaris)
- pidstat (Linux)
- Performance Monitor (Windows)
- top (Linux – prefer prstat on Solaris)
- DTrace (Solaris)

ORACLE

Processes: prstat-Lm

Processes: prstat-Lm

PID	USERNAME	USR	SYS	TRP	TFL	DEF	LCK	SLP	LAT	VCX	ICX	SCL	SIG	PROCESS/LWPID
10597	huntrch	94	1.4	0.0	0.0	0.0	4.1	0.0	0.9	139	266	71K	2	java/24
10597	huntrch	94	1.3	0.1	0.0	0.0	4.0	0.0	1.0	150	350	67K	4	java/23
10597	huntrch	93	1.3	0.1	0.0	0.0	4.1	0.0	1.6	150	532	68K	11	java/22
10597	huntrch	92	1.3	0.1	0.0	0.0	4.1	0.0	2.2	156	381	65K	52	java/21
10597	huntrch	12	0.1	0.0	0.0	0.0	2.8	86	0.0	23	32	2K	1	java/2
10597	huntrch	2.8	0.0	0.0	0.0	0.0	96	0.0	1.3	56	10	143	0	java/11
10597	huntrch	2.3	0.0	0.0	0.0	0.0	98	0.0	0.1	5	911	906	0	java/5
10597	huntrch	2.3	0.0	0.0	0.0	0.0	98	0.0	0.1	5	850	850	0	java/4
10597	huntrch	2.3	0.0	0.0	0.0	0.0	98	0.1	0.1	3	1K	1K	0	java/3
10597	huntrch	1.7	0.0	0.0	0.0	0.0	98	0.0	0.0	8	5	29	0	java/12
10597	huntrch	1.3	0.0	0.0	0.0	0.0	98	0.0	1.1	6	4	8	0	java/6
1953	huntrch	0.6	0.4	0.0	0.0	0.0	0.0	97	1.6	103	11	5K	0	Xvnc/1
10597	huntrch	0.3	0.1	0.0	0.0	0.0	100	0.0	0.0	262	5	1K	0	java/7
1875	huntrch	0.2	0.1	0.0	0.0	0.0	100	0.1	0.1	46	0	1K	0	Xvnc/1
2017	huntrch	0.1	0.2	0.0	0.0	0.0	0.0	100	0.0	46	1	1K	46	perfbar/1
2031	huntrch	0.1	0.2	0.0	0.0	0.0	0.0	100	0.0	45	0	991	45	perfbar/1
2234	huntrch	0.2	0.1	0.0	0.0	0.0	0.0	100	0.0	44	1	263	0	gnome-terminal/1
5558	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	99	0.7	161	4	84	0	jstadv/14
10597	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	98	1.5	160	4	86	0	java/14
10606	huntrch	0.0	0.1	0.0	0.0	0.0	0.0	100	0.0	23	0	249	0	prstat/1
5558	huntrch	0.1	0.0	0.0	0.0	0.0	100	0.0	0.0	5	1	233	0	jstadv/15
Total: 71 processes, 182 lwps, load averages: 2.07, 0.65, 0.25														

ORACLE

Data of Interest

When monitoring processes, these columns should be looked at:

- **Number of Threads per Process:** Sum of all the LWPs for a given process. For example, all the threads listed for Java. (**PROCESS/LWPID**)
- **CPU Usage:** **USR** and **SYS** columns
- **Locks:** **LCK** column
- **Context Switches:** **VCX** and **ICX**

The number of threads is just an idea of the complexity of the application. A large number or small number is not necessarily bad. But, if you are on a large hardware thread system and there is only a small number of threads in the Java application, you have found the reason why the application does not scale: there's not enough Java threads to keep the available number of hardware threads busy.

Processes: mpstat

Processes: mpstat

huntrch@ditka: ~																	
File		Edit		View		Terminal		Tabs		Help							
huntrch@ditka:~\$ mpstat 5																	
CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl		
0	21	1	6	319	133	261	18	23	3	0	551	4	3	0	92		
1	19	1	4	36	14	59	15	23	3	0	491	4	3	0	93		
2	17	1	4	19	12	61	13	23	3	0	355	4	3	0	90		
3	18	1	4	16	15	49	12	23	3	0	390	4	3	0	91		
CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl		
0	28	2	0	192	83	92	32	14	2	0	185	78	15	0	7		
1	49	1	0	37	1	80	28	16	2	0	139	80	16	0	4		
2	28	1	0	20	7	94	34	17	1	0	283	83	12	0	5		
3	39	1	2	52	1	99	36	16	3	0	219	74	19	0	7		
CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl		
0	34	0	2	171	75	78	32	12	1	0	173	90	9	0	2		
1	38	1	0	39	1	84	29	13	2	0	153	66	12	0	23		
2	28	8	0	21	9	97	31	20	2	0	167	67	13	0	20		
3	35	3	1	43	1	98	29	20	3	0	190	52	25	0	23		

ORACLE

Columns of interest

csw: Context switches

icsw: Involuntary context switches

smtx: Spin on mutex

A high SMTX value is an indication of severe lock contention. Java 5 HotSpot changed the way Java locks are implemented in the HotSpot JVM. It used to delegate them to operating system primitives that mapped directly to monitoring SMTX values. But Java 5 HotSpot implements them in user code and delegates to operating system primitives only in worse case lock contentions.

Monitoring the Kernel

Monitoring the Kernel

- Data of interest
 - Kernel CPU utilization, locks, system calls, interrupts, migrations, run queue depth
- Tools to monitor the kernel
 - vmstat (Linux & Solaris)
 - mpstat (Solaris)
 - lockstat & plockstat (Solaris)
 - Performance Monitor (Windows)
 - DTrace (Solaris)
 - intrstat (Solaris)



ORACLE

Why are high sys/kernel cpu, run queue depth, lock contention, migrations and context switching important to monitor?

CPU Usage: Moving usage out of kernel provides more cycles for running applications and better performance.

Lock contention: Inhibits scalability

Migrations: Process migration from one CPU to another should be minimized. When data is moved from one CPU to another, it is not cached until it is moved over. It can take thousands of cycles to move the data over, which can be quite an expensive operation.

Run Queue Depth: As the CPU schedules threads to execute if there are not enough CPUs available for the number threads that want to execute, those threads are queued up in the run queue. If the queue gets really deep, we are saturating the system with work.

Kernel: vmstat

Kernel: vmstat

kthr			memory			page			disk			faults			cpu							
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	f0	s0	s1	s2	in	sy	cs	us	sy	id	
0	0	0	659620	168400	3	19	6	2	3	0	4	-0	1	-0	1	385	886	432	3	2	95	
0	0	0	129404	26456	2	87	10	0	0	0	0	0	5	0	0	0	2320	247894	4745	37	20	44
0	0	0	127920	25524	23	395	90	0	0	0	0	0	27	0	1	1844	241699	4280	45	21	35	
0	0	0	126380	24204	1	92	0	0	0	0	0	0	0	0	0	0	2455	283529	4916	34	19	48
0	0	0	126380	24208	1	93	0	0	0	0	0	0	0	0	0	0	2469	288083	4933	33	19	49
0	0	0	126376	24204	1	79	0	0	0	0	0	0	0	0	0	0	2171	241732	4294	33	20	47
0	0	0	126344	24172	1	78	1	0	0	0	0	0	16	0	3	2202	242373	4782	36	20	44	
0	0	0	126544	24372	1	85	0	0	0	0	0	0	0	0	0	0	2127	263456	4928	42	21	37
0	0	0	126504	24332	1	92	0	0	0	0	0	0	0	0	6	2498	284583	5041	33	19	48	
0	0	0	126112	23940	1	144	2	0	0	0	0	0	0	0	2	2027	273269	5311	46	20	34	
0	0	0	125252	22952	3	97	53	0	0	0	0	0	10	0	14	2091	231099	4423	35	20	44	

Columns of Interest

us and **sy**: Kernel CPU utilization

r: Run queue depth. Represents number of runnable kernel threads.

In this example, the **sy** column seems to be a little high assuming that there is not a high level of network or disk I/O.

Kernel: mpstat

Kernel: mpstat

CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl
0	21	1	6	319	133	261	18	23	3	0	551	4	3	0	92
1	19	1	4	36	14	59	15	23	3	0	491	4	3	0	93
2	17	1	4	19	12	61	13	23	3	0	355	4	3	0	90
3	18	1	4	16	15	49	12	23	3	0	390	4	3	0	91
CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl
0	28	2	0	192	83	92	32	14	2	0	185	78	15	0	7
1	49	1	0	37	1	80	28	16	2	0	139	80	16	0	4
2	28	1	0	20	7	94	34	17	1	0	283	83	12	0	5
3	39	1	2	52	1	99	36	16	3	0	219	74	19	0	7
CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl
0	34	0	2	171	75	78	32	12	1	0	173	90	9	0	2
1	38	1	0	39	1	84	29	13	2	0	153	66	12	0	23
2	28	8	0	21	9	97	31	20	2	0	167	67	13	0	20
3	35	3	1	43	1	98	29	20	3	0	190	52	25	0	23

ORACLE

Columns of Interest

sys: kernel CPU utilization

smtx: locks

syscl: system calls

intr: interrupts

migr: migrations

csw: context switches

icsw: involuntary context switches

In this example, the system is fairly busy. Once again, if there is limited network and disk I/O, the level CPU usage may merit further investigations.

Note: Why does CPU have a larger number of interrupts than the other? Typically one CPU is designated by the operating system to handle interrupts. Consequently, that CPU has a higher number of interrupts.

Kernel: prstat-Lm

Kernel: prstat-Lm

PID	USERNAME	USR	SYS	TRP	TFL	DFL	LCK	SLP	LAT	VCX	ICX	SCL	SIG	PROCESS/LWPID	
6506	huntrch	49	24	0.2	0.0	0.0	1.1	19	7.3	4K	1K	.13	0	java/2	
6505	huntrch	11	7.1	0.0	0.0	0.0	0.3	74	7.5	4K	162	87K	0	java/2	
766	huntrch	3.0	0.5	0.0	0.0	0.0	0.0	91	2.1	266	131	1K	64	xorg/1	
6506	huntrch	1.2	0.6	0.0	0.0	0.0	0.0	98	0.0	0.3	595	10	1K	0	java/3
990	huntrch	1.1	0.1	0.0	0.0	0.0	0.0	97	1.5	8	64	291	0	wnck-applet/1	
6506	huntrch	0.5	0.1	0.0	0.0	0.0	0.0	97	0.0	2.3	195	3	196	0	java/5
6506	huntrch	0.3	0.2	0.0	0.0	0.0	0.0	99	0.0	0.1	193	7	385	0	java/4
972	huntrch	0.3	0.2	0.0	0.0	0.0	0.0	99	0.2	97	0	549	0	metacity/1	
6190	huntrch	0.2	0.3	0.0	0.0	0.0	0.0	99	0.5	165	1	647	0	java/21	
1919	huntrch	0.1	0.0	0.0	0.0	0.0	0.0	100	0.1	27	0	78	0	soffice.bin/1	
6417	huntrch	0.3	0.0	0.0	0.0	0.0	0.0	100	0.0	4	0	8	0	thunderbird-/1	
6505	huntrch	0.2	0.1	0.0	0.0	0.0	0.0	100	0.0	0.4	44	0	204	0	java/3
6512	huntrch	0.2	0.1	0.0	0.0	0.0	0.0	100	0.0	17	23	111	0	screenshot/1	
6506	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	99	0.4	151	38	87	0	java/9	
1095	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.0	53	0	211	0	firefox bin/1	
6190	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	99	0.4	156	24	87	0	java/14	
6505	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	99	0.0	0.6	32	0	32	0	java/5
980	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	99	0.7	12	1	46	0	nautilus/1	
6494	huntrch	0.2	0.0	0.0	0.0	0.0	0.0	100	0.0	13	0	58	0	gimp-2.4/1	
6505	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.2	156	1	87	0	java/9	
1095	huntrch	0.1	0.1	0.0	0.0	0.0	0.0	100	0.0	0.2	104	0	150	0	firefox-bin/3
total: 99 processes, 321 lwps, load averages: 1.60, 1.28, 0.61															

Columns of Interest

SYS: kernel CPU utilization

LCK: locks

SCL: system calls

VCX: voluntary context switches

ICX: involuntary context switches

Voluntary context switching can be used to identify an application that may be suffering high lock contention with a Java 5, Java 6, or Java 7 HotSpot or JRockit JVM.

Summary

Summary

In this lesson, you should have learned about monitoring various aspects of the operating system including how to:

- Monitor CPU usage
- Monitor network I/O
- Monitor disk I/O
- Monitor virtual memory usage
- Monitor processes including lock contention

ORACLE

Monitoring the JVM

Chapter 4

4

Monitoring the JVM

ORACLE

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

Objectives

Objectives

After completing this lesson, you should be able to:

- Describe HotSpot generational garbage collector components and operation
- Monitor the JVM garbage collector with command-line tools
- Monitor the JVM garbage collector with GUI tools like JConsole and VisualVM
- Monitor the JVM JIT compiler
- Define application throughput and responsiveness

ORACLE

What to Monitor

What to Monitor

- The two major areas to monitor at the JVM level:
 - Garbage collector
 - JIT compilation
- Data of interest
 - Frequency and duration of collections
 - Java heap usage
 - Number of application threads



Garbage Collector: The portion of the JVM responsible for freeing memory no longer utilized by application logic; the “magic” that lets programmers not have to worry about “managing memory”

Garbage collection involves traversing Java heap spaces where application objects are allocated and managed by the JVM's garbage collector.

JIT compilation: The portion of the JVM responsible for turning byte code into executable instructions for the target hardware platform

Questions to Ask

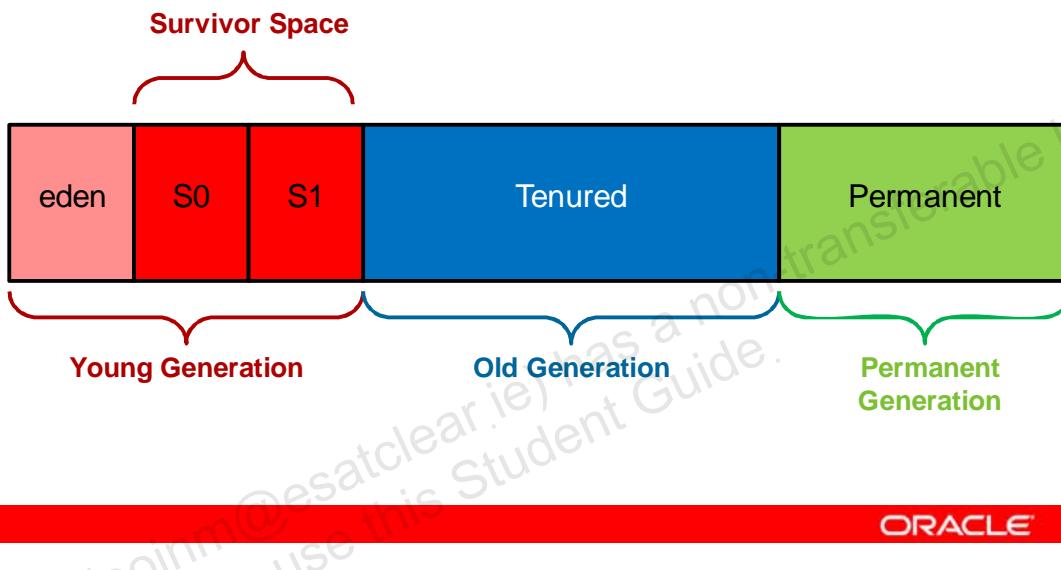
- How frequently are garbage collections occurring?
- What is the amount of time that is taken for a GC to complete?

Note: The amount of time that garbage collection takes is not necessarily related to the size of a given Java heap space. Instead, it is related to the size or number of live objects in that heap space.

HotSpot GC Basics

HotSpot GC Basics

- HotSpot uses what is termed “generational collectors.”
- HotSpot Java heap is allocated into generational spaces.



Generational Garbage Collector

A garbage collector that identifies long-lived objects based on the number of garbage collections the object survives

Young Generation

The young generation holds newly created objects and consists of an eden space plus two survivor spaces.

Eden space – The memory space where objects are initially allocated.

Survivor spaces – The memory space used to age newer objects.

Tenured or Old Generation

Long-lived objects are moved from the survivor spaces to the tenured generation. The tenured space is generally larger than young generation and holds these long-lived objects.

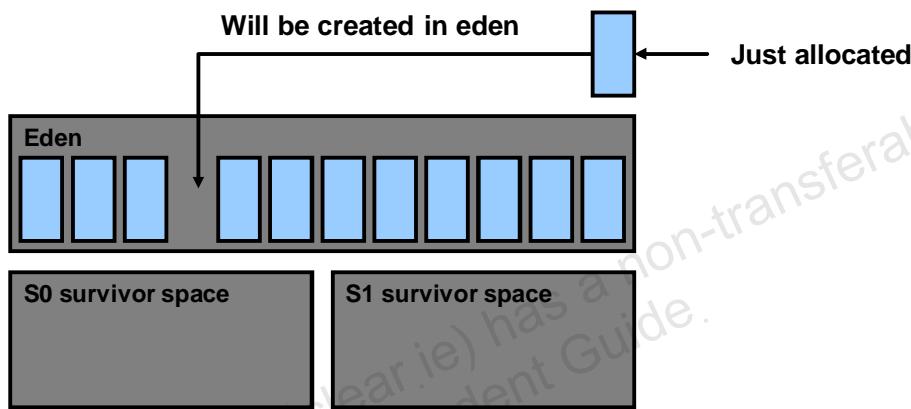
Permanent Generation

The permanent generation holds data needed by the virtual machine to describe objects that do not have an equivalence at the Java language level. For example, objects describing classes and methods are stored in the permanent generation.

The Young GC Process: Part 1

The Young GC Process: Part 1

- New objects are allocated to the eden space.
- When eden space is full, a minor garbage collection is triggered:
 - “Stop the World” event

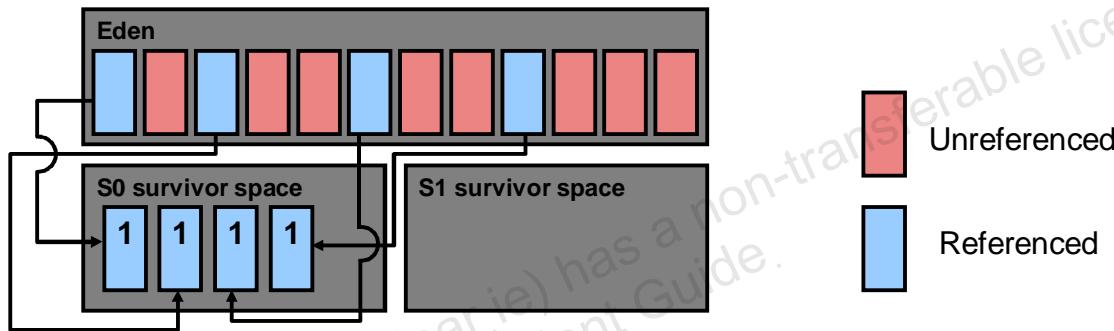


All new objects are allocated to the eden space. When the eden space is full, a minor garbage collection is triggered. The garbage collection is considered minor as it only affects the young generation, which is generally smaller and has fewer objects than the old generation. Minor garbage collections are faster with a minimal impact on performance.

Stop the World Event – A minor garbage collection is a Stop the World operation. This means that all application threads are stopped until the operation completes. Minor garbage collections are *always* Stop the World events.

The Young GC Process: Part 2

- Unreferenced objects are garbage collected.
- Referenced objects are copied to survivor space and have their age incremented.



When the minor garbage collection is triggered:

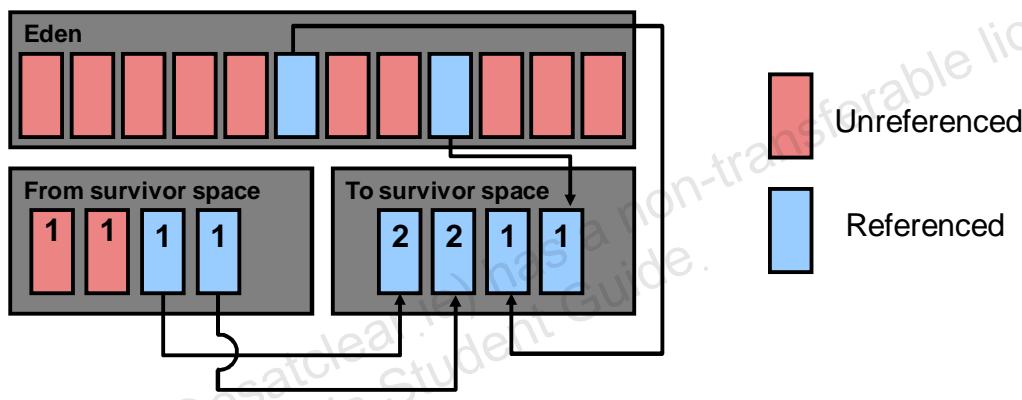
1. The GC identifies any referenced objects.
2. These are copied to a survivor space and incremented.
3. The eden space is then cleared of all objects.

After the minor garbage collection, object allocation to the eden space begins again.

The Young GC Process: Part 3

The Young GC Process: Part 3

- Next minor GC
 - Referenced objects from last GC become “from” survivor space.
 - Referenced objects are copied to the “to” survivor space.
 - Surviving object ages are incremented.



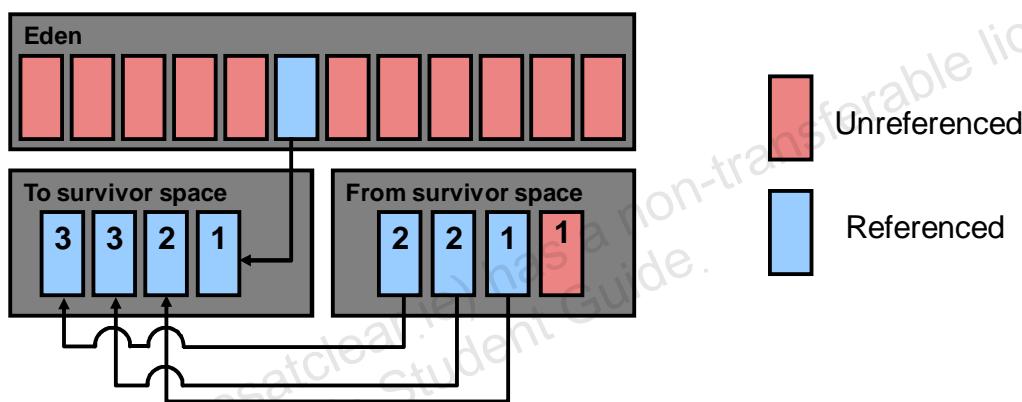
Once the eden space fills up, it again triggers a minor garbage collection.

1. The survivor space with referenced objects from the last GC is designated as the “from” survivor space.
2. Referenced objects from the eden and the “from” space are now copied to the previously empty survivor space and incremented. The survivor space is designated as the “to” survivor space.
3. The eden space and the “from” survivor spaces are cleared of objects.

The Young GC Process: Part 4

The Young GC Process: Part 4

- Next minor GC
 - Referenced objects from last GC become “from” survivor space.
 - Referenced objects are copied to the “to” survivor space.
 - Surviving object ages are incremented.



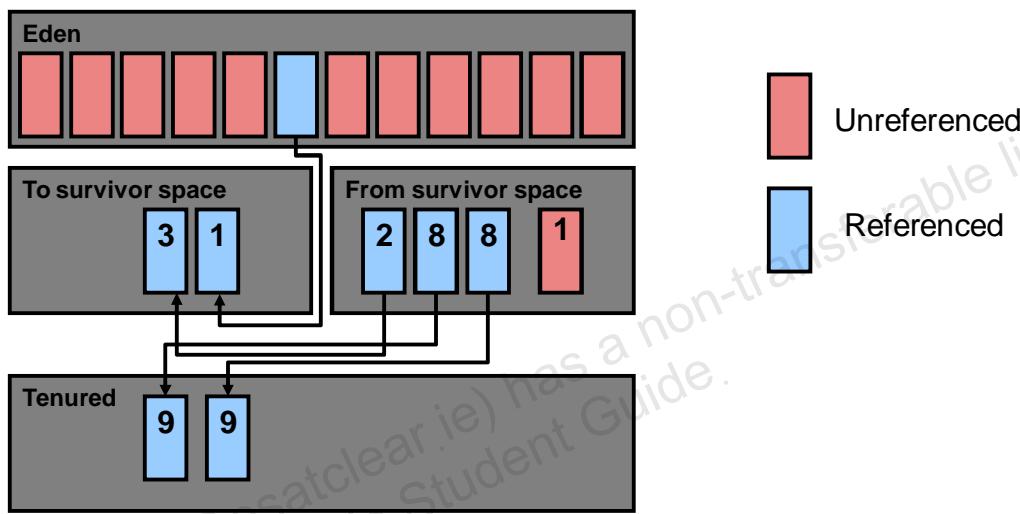
At the next minor GC:

1. The “to” survivor space from the last GC becomes the “from” survivor space for this GC.
2. Referenced objects from the eden and “from” spaces are copied to the “to” survivor spaces. Object ages are incremented.
3. The eden space and “from” survivor space are cleared.

The Young GC Process: Part 5

The Young GC Process: Part 5

- Promoted to Old Space
 - When age threshold is reached, objects are eventually promoted to tenured space.



ORACLE

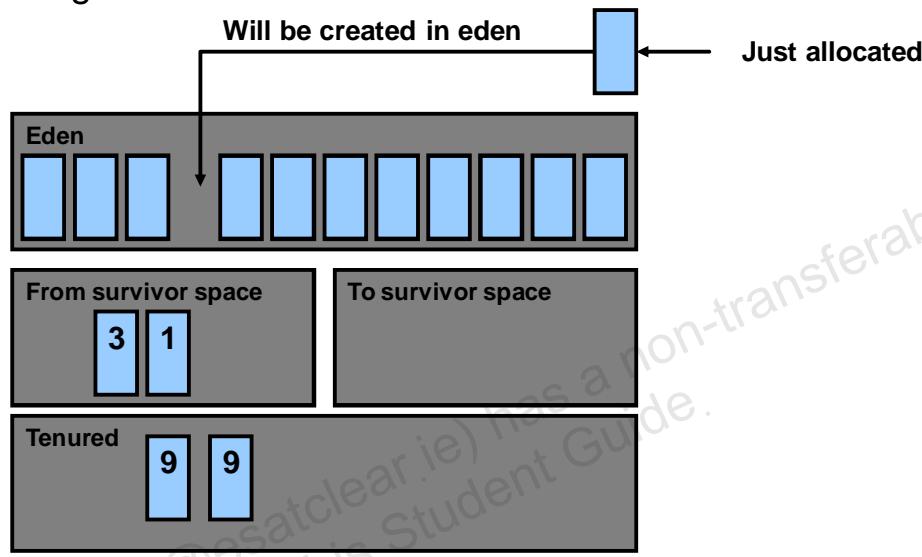
Eventually, when objects reach a certain age threshold, they are promoted to tenured space.

- The “to” survivor space from the last GC becomes the “from” survivor space for this GC.
- Referenced objects from the eden and “from” spaces are copied to the “to” survivor spaces.
- In this example, objects surviving 8 GCs are promoted to tenured space.
- The eden space and “from” survivor space are cleared.

The Young GC Process: Summary

The Young GC Process: Summary

- Process repeats at each minor GC.
- When objects reach an age threshold, they are copied to old generation.



The process as outlined in the last four steps repeats itself until an object age threshold is met. At that point, aged objects are copied to the old generation space. The more often minor garbage collections occur, the more quickly objects are promoted to old generation space.

Young Generation Recap

Young Generation Recap

- Young generation
 - Eden
 - Survivor spaces
 - Objects age here
- Minor garbage collections are always “Stop the World” events
- Minor garbage collections can be
 - Single-threaded events
 - Multithreaded (Parallel) events



ORACLE

Serial Garbage Collector: The single-threaded young generation garbage collector. It can be specified on the command line with: `-XX:+UseSerialGC`. This enables a serial garbage collector for both young and old generation. The command-line option is discussed in more detail later in the course.

Multithreaded (Parallel) Garbage Collector: For multithreaded young generation GC, you have two options, both of which are considered parallel collectors. By default, each option is used with a specific old generation collector.

`-XX:+UseParallelGC` (used with the old generation throughput collector)

`-XX:+UseParNewGC` (used with the old generation concurrent collector)

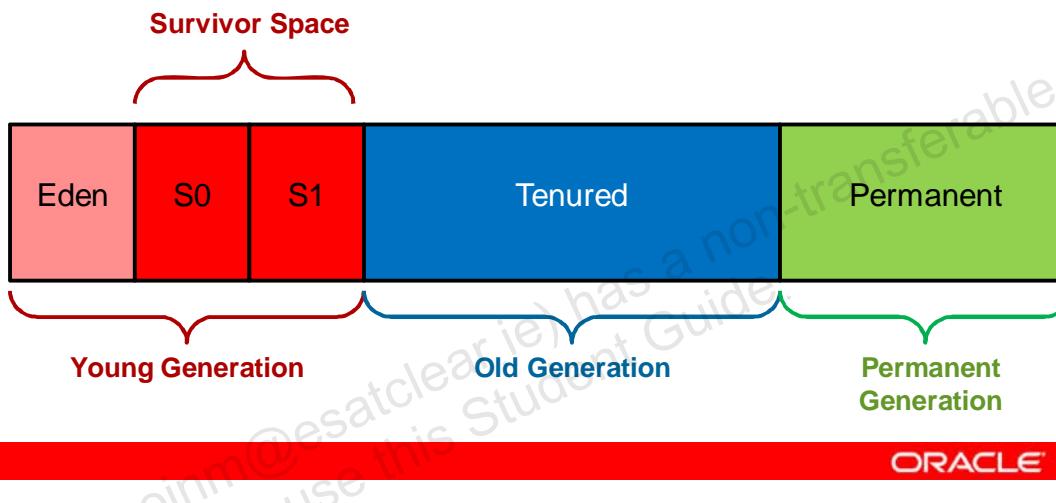
Both the throughput and concurrent old generation collectors are talked about in more detail later in the course.

Server applications rarely use the serial collector. Desktop client applications typically use it. However, as client applications grow in size, they are moving toward using multithreaded collectors as well.

Tenured (old) Generation

Tenured (old) Generation

- Contains objects that have reached the minor collection age threshold
- Full garbage collections can be:
 - Single-threaded
 - Multithreaded (Parallel)
 - Mostly concurrent



Tenured (old) Generation

The terms “old” and “tenured” generation are commonly used interchangeably.

Full Garbage Collection: This occurs when the old or permanent generation fills up. All generations are collected. The young generation is collected first using one of the young generation collectors described on the previous page. Then, an old and permanent generation collection is performed using a collector designed for those spaces.

Old Generation Collectors

The old and permanent generation spaces have a serial, parallel, and mostly concurrent garbage collectors.

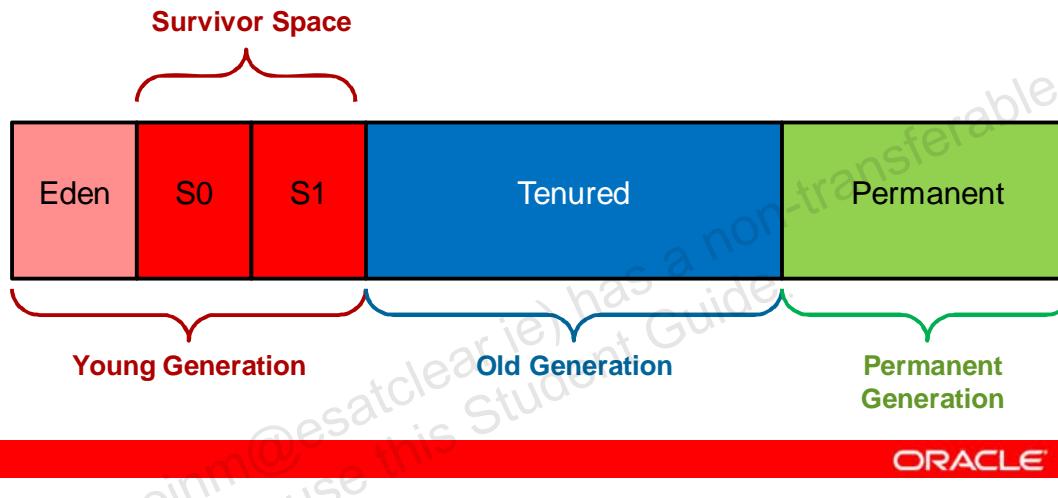
- XX:+UseSerialGC (Serial collector, implies a young generation serial collector)
- XX:+UseParallelOldGC (Parallel collector. -XX:UseParallelGC for young generation is implied.)
- XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode (Concurrent mark sweep collector. A mostly concurrent collector. Implies a -XX:+UseParNewGC young generation collector.)

Detailed coverage of each of these collectors is provided later in the course (Lesson 7).

Permanent Generation

Permanent Generation

- Contains metadata required by the JVM:
 - Class objects and methods
 - Interned Strings



Permanent Generation

The permanent generation contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

Interned Strings: A literal defined in double quotes in Java or by calling the intern method on a string variable, (e.g., `stringVar.intern()`)

Tools for Monitoring GC

Tools for Monitoring GC

- `-verbose:gc`
- `-XX:+PrintGCTimeStamps`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCApplicationStoppedTime`
- `-XX:+PrintGCApplicationConcurrentTime`
- `Jstat, jps`
- `JConsole`
- `Java VisualVM`
- `VisualGC`
- `DTrace` (Java HotSpot JDK 6 contains samples.)

ORACLE

Monitoring GC: The Tools

There are probes in Hotspot VM for the DTrace utility.

Using -verbose:gc

Using -verbose:gc



- What does -verbose:gc do?
- What to look for?

```
host:~ $ java -client -verbose:gc -Xmx12m -Xms3m -Xmn1m  
-XX:PermSize=20m -XX:MaxPermSize=20m  
-jar /usr/java/demo/jfc/Java2D/Java2Demo.jar  
...  
[GC 1884K->1299K(5056K), 0.0031820 secs]  
...
```

- Adding -XX:+PrintGCTimeStamps

```
host:~ $ java -client -verbose:gc -XX:+PrintGCTimeStamps  
-Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m  
-jar /usr/java/demo/jfc/Java2D/Java2Demo.jar  
...  
3.791: [GC 1884K->1299K(5056K), 0.0031820 secs]  
...
```

ORACLE

-verbose:gc displays detailed garbage collection information to the console for each GC while the Java application runs.

What to Look For: Over a period of GC events, the overall amount of live data is constantly increasing and you see a full GC. A large amount of space is reclaimed. Then this pattern repeats itself. The pattern indicates that objects are being promoted too quickly or that the Heap size is too small.

-verbose:gc Output Defined

- **GC:** Means a minor GC. “Full GC” is listed for a full GC.
- **1884K:** Heap size before the GC
- **1299K:** Heap size after the GC
- **5056K:** Overall size of the Java heap
- **0.0031820 secs:** Time to complete the GC

-verbose:gc -XX:+PrintGCTimeStamps

Displays the seconds since the application started to each outline from -verbose:gc.

3.791 – Time since the launch of the JVM.

Additional -verbose:gc Print Options

Additional -verbose:gc Print Options

- `-XX:+PrintGCDateStamps` (Java 6u4 and later)

```
host:~ $ java -client -verbose:gc -XX:+PrintGCDateStamps
-Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m
-jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
...
2008-06-10T06:12:47.513-0500: [GC 10308K->2725K(101376K),
0.0320270 secs]
...
```

- `-XX:+PrintGCDateStamps` and `-XX:+PrintGCTimeStamps`
 - PrintGCTimeStamps output after date & time.

```
host:~ $ java -client -verbose:gc -XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m
-XX:MaxPermSize=20m -jar /usr/java/demo/jfc/Java2D/Java2Demo.jar
...
2008-06-10T06:21:29.711-0500: 5.551:[GC 10377K->2786K(101376K),
0.0271350 secs]
...
```

ORACLE

`-XX:+PrintGCDateStamps` – Adds a date stamp to each line of `-verbose:gc` output.

2008-06-10T06:12:47.513-0500 – GMT time with offset when the GC occurred. This is ISO 8607 format.

Format: YYYY-MM-DDTHH.MM.SS.mmm-tttt

YYYY = year

MM = month

DD = day of month

HH = hour

MM = minute

SS = seconds

mmm = milliseconds

ttt = time zone offset

Using -XX:+PrintGCDetails

Using -XX:+PrintGCDetails

-XX:+PrintGCDetails



```
host:~ $ java -client -XX:+PrintGCDetails -Xmx12m -Xms3m -Xmn1m  
-XX:PermSize=20m -XX:MaxPermSize=20m  
-jar /usr/java/demo/jfc/Java2D/Java2Demo.jar  
...  
[GC [DefNew: 490K->64K(960K), 0.0032800 secs] 5470K->5151K(7884K),  
0.0033270 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  
  
[Full GC (System) [Tenured: 5087K->5151K(6924K), 0.0971070 secs]  
6047K->5151K(7884K), [Perm : 11178K->11178K(16384K)],  
0.0972120 secs] [Times: user=0.10 sys=0.01, real=0.10 secs]  
...
```

ORACLE

Command line used most frequently by engineering and is recommended for customers to use frequently as well. Looking for the same sort of pattern as mentioned into -verbose:gc.

GC: Minor GC

DefNew: The serial collector is being used (Default New)

Young Space

490K: Amount of live data in the young generation heap space

64K: Amount of live data in the young generation after the minor GC

(960K): Young generation space is 960K

0.0032800 secs: Time for the minor GC

5470K: Overall Java Heap space before GC

5151K: Overall Java Heap space after GC

7884K: Total size of the Java Heap

Times: user=0.00 sys=0.00, real=0.00 secs – CPU time in user, sys, and overall.

Full GC (System) – A full GC triggered by a `System.gc()` call in code.

Tenured Space

5087K: Tenured space before GC
5151K: Tenured space after full GC
6924K: Total Tenured space.
0.0971070 secs: Time for the full GC

Full Heap Space

6047K: Java Heap occupancy before full GC
5151K: Java Heap occupancy after full GC
7884K: Total size of the Java heap

Permanent Generation Space

11178K: Size of permanent space before full GC
11178K: Size of permanent space after full GC
16384K: Total size of permanent space
0.0972120 secs: Time for the full GC

Times: user=0.10 sys=0.01, real=0.10 secs – “user” and “sys” are CPU clock cycles for GC.
“real” represents the elapsed time for GC.

Printing Pause Time

Printing Pause Time

- `-XX:+PrintGCApplicationStoppedTime`
- `-XX:+PrintGCApplicationConcurrentTime`
- Helpful when tuning pause time-sensitive applications



ORACLE

`-XX:+PrintGCApplicationStoppedTime`

The amount of time an application has been paused for a safe point operation. The most common safe point operation is a GC.

Safe Point Operations – JVM operations that require a Stop the World event. Safe point operations require a known state before they can take place.

For example, JDK 5 introduced the idea of biased locks. Generally, locking operations are requested by the last thread that acquired the lock. So, locking operations by default are biased to this heuristic. However, if this turns out not to be the case in an application, the JVM requires a safe point operation to turn off the locking bias.

`-XX:+PrintGCApplicationConcurrentTime`

The amount of time the application runs between safe point operations. The time between the last safe point operation and the current safe point operation.

Note: These two command-line options can be very useful in identifying JVM-induced latencies.

Using jps

Using jps

jps

- Command-line utility to find running Java processes
- Included in the HotSpot JDK
- Capable of local and remote monitoring
- Example:

```
jps [-q] [-mlvV] [<hostid> where <hostid> =  
<hostname>[:<port>]]
```



ORACLE

jps is very similar to ps in Solaris. Its purpose is to identify the process ID of Java processes.

Note: See jps man page on oracle.com for details about -q, -mlvV options:

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

Using jstat

Using jstat



jstat

- Command-line utility that runs on a local or remote JVM
- Included in the HotSpot JDK
- Example:

```
jstat -<option> [-t] [-h<lines>] <vmid>  
[<internal> [<count>]]
```

- Garbage collection options:

```
-gc, -gccapacity, -gccause, -gcnew, -gcnewcapacity,  
-gcold, -gcoldcapacity, -gcpermcapacity, -gctuil
```

ORACLE

jstat is a command-line tool that displays detailed performance statistics for a local or remote HotSpot VM. The `-gctuil` command-line option is used most frequently.

See the `jstat` man page on download.oracle.com for details on garbage collection options:
<http://download.oracle.com/javase/1.5.0/docs/tooldocs/share/jstat.html>

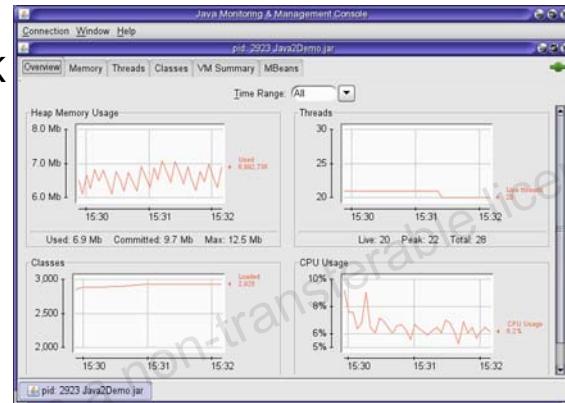
Caution: When using the Concurrent Mark Sweep (CMS) collector (also known as concurrent collector), `jstat` reports two full GC events per CMS cycle, which is obviously misleading. However, young generation stats are accurate with the CMS collector.

Using jconsole

Using jconsole

jconsole

- Graphical Monitoring and Management Console
 - Included in the HotSpot JDK
 - Can attach multiple JVMs, locally or remotely
 - Graphical view of
 - heap memory
 - threads
 - CPU usage
 - class loading
- Multiple jconsole sessions can attach to a single JVM



ORACLE

jconsole is a graphical monitoring and management console that comes with the HotSpot JDK. jconsole supports both Java Management Extensions (JMX) and MBean technology. This allows jconsole to monitor multiple JVMs at the same time. In addition, more than one jconsole session can monitor a single JVM session at the same time. jconsole can monitor the following JVM features:

- Memory usage by memory pool/spaces
- Class loading
- JIT compilation
- Garbage collection
- Threading and logging
- Thread monitor contention

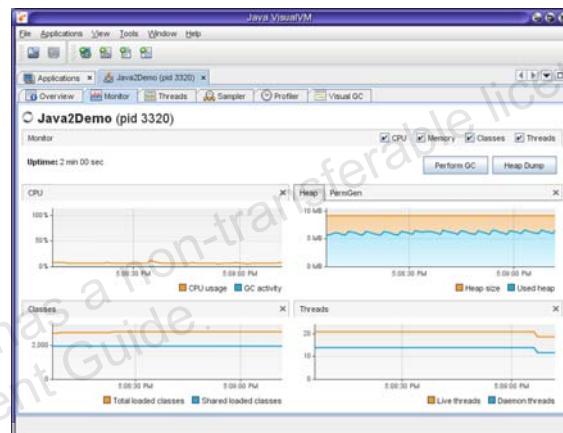
Note: MBeans are managed beans, Java objects that represent resources to be managed. They can be used with JMX applications.

Using VisualVM

Using VisualVM

VisualVM

- Graphical JVM monitoring tool
- Included with JDK 6 as of update 7
- Open source project
- Integrates with other JDK tools
- Includes analysis and troubleshooting abilities
- Can be enhanced with plugins
 - Plugin Center
 - <https://visualvm.dev.java.net>



VisualVM is a lightweight graphical monitoring tool that is included with JDK 6 update 7 or later. Based on an open source project (<https://visualvm.dev.java.net>), it has a number of cool features including:

- Integration with JDK tools including jconsole and a subset of NetBeans Profiler
- Performance analysis and troubleshooting abilities including thread deadlock detection and thread monitor contention
- Easily extended through its plug-in API
- Plug-ins can be installed directly into VisualVM using its plug-in center. Available plug-ins include:
 - jconsole
 - VisualGC: covered in the next slide
 - Glassfish plug-in
 - btrace plug-in: A byte code trace plugin for the JVM. Similar to DTrace, but for a JVM.

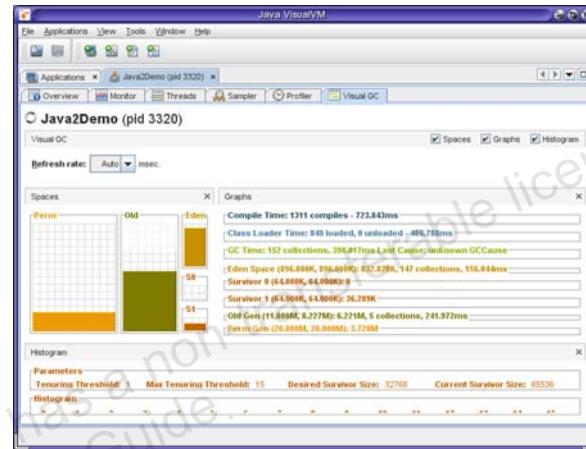
Visual VM can look visually for the same sort of GC patterns discussed earlier in the course.

Using VisualGC

Using VisualGC

VisualGC

- Stand-alone GUI or VisualVM plug-in
- Not included in HotSpot JDK
- Visually observes garbage collection behavior
- Also includes class loading and JIT compilation information

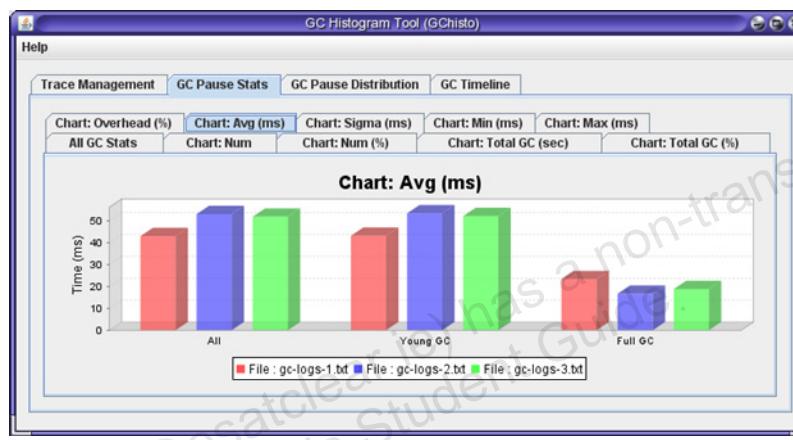


ORACLE

VisualGC is a stand-alone graphical JVM monitor or a VisualVM plugin. In this course and typically, VisualGC is used as a VisualVM plug-in. You can install it directly using the VisualVM plug-in center. With VisualGC, a picture is worth a thousand words, because you can see visually exactly what is going on with the garbage collector. In addition to garbage collection, VisualGC also provides information about class loading and JIT compilation.

Using GCHisto

- Stand-alone GUI
- Open Source project: <http://gchisto.dev.java.net>
- Not included with JDK
- Summarizes GC activity obtained from GC logs
 - Can Compare Multiple Logs



ORACLE

GCHisto is a stand-alone GUI for analyzing GC log data. (There is also a VisualVM plugin under development.) You can analyze multiple log files at the same time. GCHisto also allows the comparison of heap sizes or collector types for JVM tuning by comparing GC logs. A GCHisto lab is included in Lesson 7.

Open Source Project: <http://gchisto.dev.java.net>

Monitoring JIT Compilation

Monitoring JIT Compilation

- Tools for monitoring JIT Compilation
 - jstat
 - jconsole
 - VisualVM
 - VisualGC
 - `-XX:+PrintCompilation` (can be mildly intrusive, 2% CPU)
 - DTrace (HotSpot JDK 6 contains samples)
- Data of interest
 - Frequency, duration, possible opt/de-opt cycles, failed compilations



ORACLE

JIT compilation is monitored infrequently. However, it is good to know what tools you can use when JIT compilation analysis is required.

- **Opt/de-opt Cycles:** The JIT compiler optimizes a piece of code. Then at some point later, it decides to de-optimize the same piece of code. When the compiler repeats this cycle on the same piece of code, this can negatively impact performance.
- **Failed Compilation:** JIT bug that causes the compiler to fail to do an optimization. This was a more frequent occurrence in Java 1.2 and 1.3. It is much more rare in Java 5 and 6.

The JIT compiler will de-optimize because it has learned some assumption in a previous optimization turned out to be wrong. So, it must de-optimize and re-optimize the affected piece of code.

Note: When you start a JVM with the `-server` option, there are a lot more JIT compiler optimizations done than when using `-client`. This is why server applications take much more time to load.

Using -XX:+PrintCompilation

Using -XX:+PrintCompilation

- -XX:+PrintCompilation Sample
 - Blue text added to output
 - Shows an opt/de-opt cycle

```
1 java.util.Properties$LineReader::readLine (452 bytes)
2 java.lang.String::hashCode (60 bytes)
3 java.lang.String::equals (88 bytes)
3 made not entrant (2) java.lang.String::equals (88 bytes)
4 java.lang.Object::<init> (1 bytes)
5 java.lang.String::indexOf (151 bytes)
6 java.lang.String::equals (88 bytes) <----- redoing 3
6 made not entrant (2) java.lang.String::equals (88 bytes)
7 java.lang.String::indexOf (151 bytes)
8 java.lang.String::equals (88 bytes) <----- redoing 6
8 made not entrant (2) java.lang.String::equals (88 bytes)
...
...
```

ORACLE

The example above shows output from the `-XX:+PrintCompilation` command-line option. The lines highlighted in blue text identify a repeating optimize/de-optimize cycle. In this example, the `java.lang.String.equals()` method is caught in this cycle.

Seeing an optimize/de-optimize cycle is considered normal. However, when the same method is repeatedly de-optimized and re-optimized this may indicate a JIT compiler bug (that is, the JIT got stuck in a deopt/reopt loop trying to do some optimization).

Made not entrant: JIT compiler tells itself it needs to disregard the optimization it made.

What Is the .hotspot_compiler File?

What Is the .hotspot_compiler File?

- File turns off JIT compilation in specific methods
 - Used very rarely
- When to use the .hotspot_compiler file
 - JIT compiler in an endless loop attempting a “heroic” optimization which will not converge
 - JIT compiler in a de-optimization – re-optimization cycle
 - JIT compiler producing “bad” code resulting in a core dump or other severe problem

ORACLE

If you see this endless repeating de-optimize/re-optimize of the same method, then you can tell HotSpot not to compile it. But, this tends not to be an issue with Java 5 and Java 6.

Using .hotspot_compiler File

Using .hotspot_compiler File

- The .hotspot_compiler file must be placed in the directory where the Java command is launched.
- The .hotspot_compiler file format
 - exclude A/B/C/D *methodName*
 - exclude java/util/HashMap clear
- Can specify on the command line
 - XX:CompileCommand=exclude , java/util/HashMap , clear



ORACLE

The .hotspot_compiler file format

exclude A/B/C/D *methodName*

Where A.B.C.D is the fully qualified package and class name and methodName is the method name.

Example: To exclude java.util.HashMap.clear(), specify:

exclude java/util/HashMap clear

You can also specify an exclusion on the command line.

-XX:CompileCommand=exclude ,A/B/C/D,*methodName*

-XX:CompileCommand=exclude ,java/util/HashMap ,clear

You can specify the path to the .hotspot_compile file.

-XX:CompileCommandFile=/path/to/file

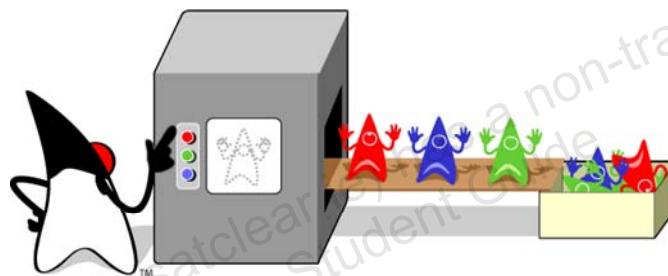
-XX:CompileCommandFile=/export/home/student/.hotspot_compile

Focusing on Throughput

Focusing on Throughput

Throughput

- Have as highest priority the raw throughput of the information or data being processed
- Maximize application throughput even at the expense of responsiveness
- Will tolerate high pause times in order to maximize throughput



ORACLE

Throughput-Sensitive Applications

A Java application that focuses on throughput emphasizes the raw throughput of the information or data being processed. This is the most important quality for the application. Pause times resulting from JVM garbage collection events are not an issue, or of very little interest. As long as the overall throughput of the application over a period of time is not sacrificed, long pause times are ok. Examples of applications that focus on throughput include:

- A large phone company printing bills or statements
- A large credit card company printing statements
- A bank calculating interest for accounts

Focusing on Responsiveness

Focusing on Responsiveness

Responsiveness

- Have as highest priority the servicing of all requests within a predefined maximum time
- Raw throughput of data or speed of processing requests are secondary to max response time goal
- Are sensitive to GC pause time



ORACLE

Responsiveness-Sensitive Applications

A Java application that focuses on responsiveness emphasizes how quickly an application responds in a given scenario rather than focusing on the raw throughput of the application. Most applications emphasizing responsiveness have a maximum pause time the application can tolerate. Examples of applications that focus on responsiveness include:

- Applications connected to a user interface such as a web browser or an IDE
- Financial trading applications
- Telecommunication applications

Java applications focusing on responsiveness are sensitive to the time it takes for garbage collection events to complete.

Summary

Summary

In this lesson, you should have learned how to:

- Examine Java HotSpot VM generational garbage collectors
- Monitor the JVM
 - GC
 - JIT compiler
- Monitor the application for throughput and responsiveness

ORACLE

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

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

Performance Profiling

Chapter 5

Performance Profiling

ORACLE

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

Objectives

Objectives

After completing this lesson, you should be able to:

- Describe the key features of NetBeans Profiler, Oracle Solaris Studio, and jmap/jhat
- Profile CPU usage to improve application performance
- Profile the JVM heap to improve application performance
- Profile applications to find memory leaks
- Profile applications to identify lock contention
- Identify anti-patterns in heap profiling
- Identify anti-patterns in method profiling

ORACLE

Tools for Profiling Java Applications

Tools for Profiling Java Applications

- Open source tools
 - NetBeans and NetBeans Profiler
 - <http://netbeans.org/>
 - [Oracle Solaris Studio](#)
 - jmap/jhat
- Free Tools
 - [Oracle JDeveloper](#)
- Commercial profilers
 - Optimizeit
 - YourKit



ORACLE

NetBeans (Covered in this course)

A free, open-source Integrated Development Environment for software developers. All the tools needed to create professional desktop, enterprise, web, and mobile applications with the Java platform, as well as with C/C++, PHP, JavaScript, and Groovy.

Oracle Solaris Studio (Covered in this course)

Oracle Solaris Studio delivers an advanced suite of tools designed to work together to provide an optimized environment for the development of single, multithreaded, and distributed applications. The debugging and analysis tools take advantage of compiler features to provide application context with high levels of accuracy, leading to more robust software. Oracle Solaris Studio also comes with an integrated development environment (IDE) tailored for use with the included compilers, the debugger, and the analysis tools.

Oracle JDeveloper (Not covered in this course)

Oracle JDeveloper integrates development features for Java, SOA, Web 2.0, Database, XML and Web services into a single development tool. The various artifacts share the same project structure and development experience, simplifying both the learning curve and the development process of composite applications that leverage a multitude of technologies.

NetBeans and NetBeans Profiler

- Characteristics:
 - CPU performance profiling using bytecode instrumentation
 - Low overhead profiling
- Capabilities:
 - Method profiling
 - Select all methods for profiling or specific method(s)
Note: You can limit everything but JDK classes
 - Memory profiling/heap profiling
 - Memory leak detection



ORACLE

The NetBeans Profiler is included in the standard NetBeans distribution. It works by injecting bytecode into your application's bytecode. The NetBeans Profiler is powerful and easy to use. The profiler can minimize profiler overhead with root method selection also making it easy to target specific parts of an application.

NetBeans

NetBeans

- Supported platforms:
 - Solaris (SPARC and x86)
 - Linux
 - Windows
 - Mac OS X
- Requirements:
 - Requires HotSpot JDK 5 or later
- Download:
 - Included out of the box in NetBeans IDE 6.0 and later
 - <http://netbeans.org>



ORACLE

Oracle Solaris Studio

- Oracle Solaris Studio Collector/Analyzer capabilities:
 - Statistical CPU profiling using JVMTI
 - Can specify sampling interval; default: one second
 - User and system CPU time
 - Inclusive or exclusive method times
 - Time spent in locks
 - View Java bytecode in User Mode and Expert Mode
 - View JIT compiler-generated assembly code in Machine Mode
 - Supports specific CPU counter collection



ORACLE

Oracle Solaris Studio has a very minimal intrusiveness right out of the box. Typically, intrusion is only 10% of application performance or less. This is much less than typical Java profilers, which have a 30% to 50% performance impact.

With Oracle Solaris Studio you collect data from the application, then store it and analyze it later. This differs from traditional Java profilers, which take snapshots and analyze applications while they run. It also enables you to analyze both user and system CPU time.

Inclusive method time is the time it takes to execute a method and anything that method calls. Exclusive method time is the time it takes to execute the method by itself. Inclusive times are good for measuring the performance of algorithms in code.

CPU counters enable you to analyze which pieces of code experience the most CPU cache misses or translation lookaside buffer misses. A translation lookaside buffer is a cache used to improve virtual address translation in a CPU.

Oracle Solaris Studio

- Solaris Studio Collector/Analyzer characteristics:
 - Easily invoked with `collect -j` on prefixed to Java command line
 - `collect -j` on `java BatchProcessor`
- Supported platforms:
 - Solaris (SPARC and x86) and Linux
- Requirements:
 - Requires HotSpot JDK 5 or later
- Additional Information:
 - <http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/index-jsp-137155.html>



ORACLE

Oracle Solaris Studio is really simple to use. Just prepend `collect -j` to your normal Java command line. For example:

`collect -j` on `java BatchProcessor`

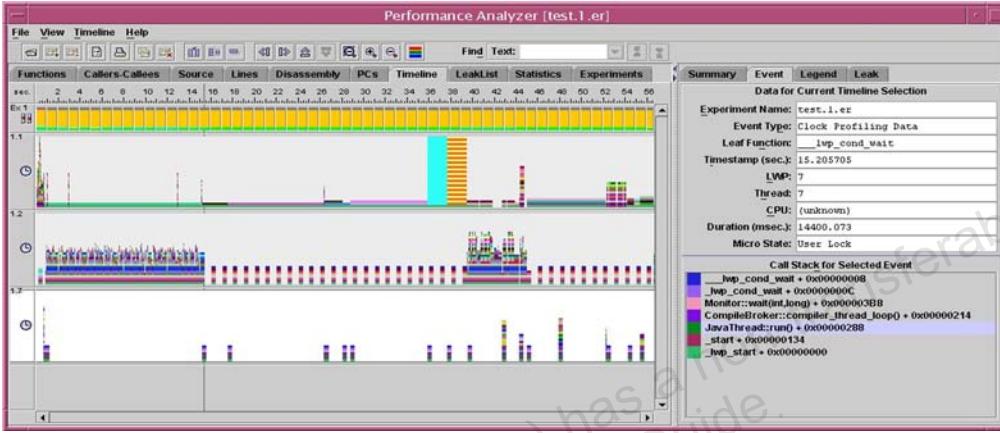
Note: Although the product is called Oracle Solaris Studio, it also works on Linux.

Oracle Solaris Studio

Oracle Solaris Studio

View options for “collected” data:

- GUI option: Analyzer GUI



- Command-line option: er_print

ORACLE

After running the `collect -j` option, an experiment file is created. The file in the slide shows the experiment file in the Analyzer. If you want a command-line option, you can use `er_print`. The `er_print` utility can be controlled by scripts. This makes the automated analysis possible.

jmap and jhat

jmap and jhat

- Used in combination:
 - jmap: Produces heap profile
 - jhat: Reads and presents the data
- Additional information:
 - Shipped with JDK 5 and later
 - Command-line tools
 - Heap memory profiling
 - Perm gen statistics
 - Finalizer statistics
 - Supported on all platforms



ORACLE

Both tools are distributed with the JDK. Both are command-line tools that are not as flashy as Studio.

Profiling Tips

Profiling Tips

- CPU profiling
- Heap profiling
- Memory leak profiling
- Lock contention profiling
- Profiling tools selection
- Inlining effect



ORACLE

Profiling Tips

- **CPU Profiling:** Used when there is a large amount of CPU time in system or kernel utilization. Also used when there is not enough application throughput.
- **Heap Profiling:** Should be used for throughput or responsiveness issues. Also should be used when full garbage collections are occurring frequently.
- **Memory Leak:** Used when a Java heap continues to grow and grow over time without bound. This can lead to “out-of-memory” errors from the JVM.
- **Lock Contention:** Used when there is a large number of context switches, which correlate to high-CPU utilization
- **Tool Selection:** Selecting the right tool for the kind of issue that needs to be addressed
- **Inlining Effect:** A JVM optimization that automatically consolidates methods for execution. This reduces the overhead associated with moving methods on and off the stack. This can lead to confusion when profiling, as two- or three-line methods may behave in a confusing manner.

CPU Profiling: Why and When

- Why perform CPU profiling?
 - CPU profiling provides information about where an application is spending most of its time.
- When is CPU profiling needed or beneficial?
 - Poor application throughput measured against a predetermined target
 - Saturated CPU utilization
 - High system or kernel CPU utilization
 - High lock contention
 - To a lesser extent, idle CPU or poor application scalability

ORACLE

Determine where the application is spending the most amount of time. Instead of focusing on individual methods, focus on the use case and call space of where your application is spending the most time. How could the algorithm be improved?

CPU Profiling: Strategies

- Start with a holistic approach to isolate major CPU consumers or hot methods:
 - Look at methods with high user and/or system CPU usage.
 - Look at both inclusive and exclusive method times.
- Profile a subset of the application:
 - Best when holistic approach is too intrusive
 - Easily done with NetBeans Profiler



ORACLE

Methods that have a high system CPU usage without performing much I/O are generally a good place to start.

Inclusive method time: The time it takes to execute a method and anything that method calls. Inclusive times are good for measuring the performance of algorithms in code. Looking at inclusive times may help identify a change in implementation or design that could be a good corrective approach.

Exclusive method time: The time it takes to execute the method by itself. Looking at exclusive times focuses on specific implementation details within a method.

By isolating a portion of the application, you can narrow the focus of your investigation. The NetBeans Profiler is typically only 10% intrusive compared to some commercial profiling tools, which can be anywhere from 50% to 90% intrusive.

CPU Profiling Entire Application

CPU Profiling Entire Application

- Oracle Studio Collector works well
 - One-second default sampling rate
 - Easy to set up; just prepend `collect -j on` to Java command line
 - Can fine-tune sampling rate
 - Can direct output to specified file name
- Solaris DTrace scripting
 - Can customize to target specific areas
 - May require DTrace scripting expertise to author the script

ORACLE

The sampling rate could be increased up to five seconds. This reduces the size of the data set produced when analyzing an application over a few hours.

Improved Lock Information

If you want to get a little better lock information, prepend the following to the command line:
`collect -j on -s on`

Solaris Studio will try to differentiate between locks that are used to block and wait for an event, that is, block and wait for input to arrive, versus contending for a shared.

DTrace

Creating your own DTrace scripts can be a bit daunting for most Java developers. However, there are several DTrace scripts in the samples directory of the JDK.

CPU Profiling a Portion of the Application

CPU Profiling a Portion of the Application

- NetBeans Profiler works very well
 - Can easily configure which classes or packages to profile, (include or !include)
 - Easy to set up if application is set up as a NetBeans Project
 - Remote or local profiling
 - Can view profiling as application is running
 - Can compare profile against another profile
- Solaris DTrace scripting
 - Customize to target specific portions.
 - May require DTrace scripting expertise to author the script

ORACLE

Remote profiling can be a bit more difficult to set up compared to local profiling. To make this easier, the NetBeans IDE includes wizards to help design a Java command line that you can use to remote profile.

Accuracy

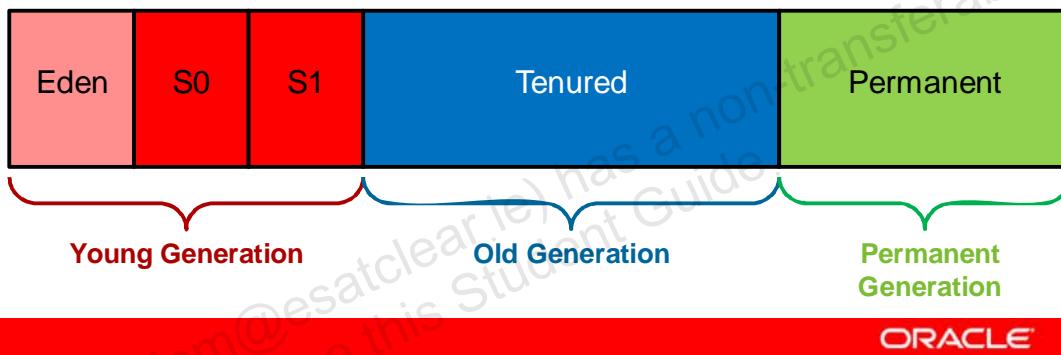
With NetBeans profiler, instrumentation is added to your application. This can affect the performance of your application, especially for very small methods. This may result in conflicting output.

Oracle Solaris Studio should give you better data because it does not add the instrumentation to your application.

Heap Profiling: Why and When

Heap Profiling: Why and When

- When is heap profiling needed or beneficial?
 - Observing frequent garbage collections
 - Application requires a large Java heap
 - Can be useful for obtaining better CPU utilization or application throughout and responsiveness
 - Less time allocating objects and/or collecting them means more CPU time spent running the application.



Heap profiling provides information about the memory allocation footprint of an application. In general, if you can minimize the number of objects being allocated, you should be able to reduce the frequency of GC events and improve application performance. However, you should avoid taking this concept to an extreme and try to avoid object allocations completely.

Heap Profiling: Strategies

Heap Profiling: Strategies

- Start with a holistic approach to isolate major memory allocators.
- If holistic approach is too intrusive, profile a subset of the application with a tool like NetBeans.
- What to look for:
 - Look at objects with a large amount of bytes being allocated.
 - Look at objects with a high number/count of object allocations.
 - Look at stack traces for locations where large amounts of bytes are being allocated.
 - Look at stack traces for locations where large number of objects being allocated.

ORACLE

What are two major memory allocators?

- Very large allocated objects
- Small objects allocated at a high rate, which also results in consuming substantial memory

String Builder Example

Take a StringBuilder object as an example. In normal usage, a StringBuilder combines some strings, produces a string object, and is garbage collected. This is a necessary object allocation.

However, if you declare a StringBuilder that is too small for the size of the strings it is combining, this forces the StringBuilder to constantly resize, thus throwing away its allocated memory and creating a new bigger space to hold the strings. This second situation is an unnecessary object allocation and is the kind of practice that should be avoided. It leads to high CPU utilization and poor performance.

Heap Profiling: Strategies

Heap Profiling: Strategies

- Cross-reference CPU profiling with heap profiling.
- Look for alternative classes, objects, and possibly caching approaches where high number/count of bytes are being allocated.
- Consider profiling while an application is running to observe memory allocation patterns.



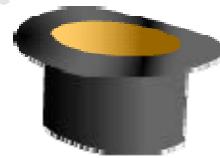
ORACLE

Look for objects that may have lengthy initialization times and allocate large amounts of memory. They are good candidates for caching.

Heap Profiling: jmap/jhat

Heap Profiling: jmap/jhat

- jmap and jhat can also capture heap profiles:
 - Not as sophisticated as the NetBeans Profiler
 - Quick and easy to use
 - Limited to a snapshot at the time of capture
 - Easy viewing of top memory consumer at time when snapshot was taken
 - Can look at stack traces for allocation location



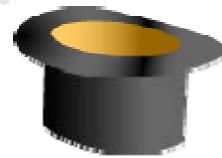
ORACLE

Although not as sophisticated as the NetBeans Profiler, jmap and jhat are a quick-and-dirty way to look at the memory profile of an application.

jmap captures the heap snapshot, and jhat displays the data.

Heap Profiling: jmap/jhat Strategies

- Focus on large memory allocators.
- Capture several snapshots.
- Compare top memory allocators.
- Snapshot can be intrusive.



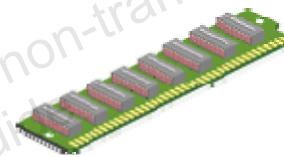
ORACLE

Consider alternative classes, objects, and possibly caching approaches for large allocators.

Note: jmap can be intrusive in that it must stop all the application threads to take the snapshot. Generally, the more live objects there are in the heap, the longer it will take for the JVM to write the dump file to disk.

Memory Leak Profiling: Why and When

- Memory leaks are situations where a reference to allocated objects remains unintentionally reachable and as a result cannot be garbage collected.
- Leads to poor application performance
- Can lead to application failure
- Can be hard to diagnose



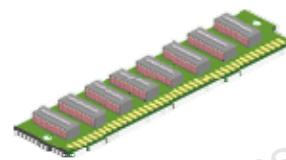
ORACLE

Map Example

The most common example of an unintended memory leak is the use of hash maps or maps. Developers will continue to add data to the map without ever removing it. The map will grow until the JVM runs out of memory. This leads to poor performance due to frequent garbage collections.

Memory Leaks Profiling Tips: Tools

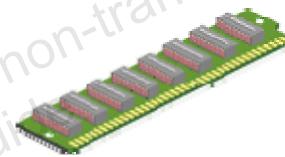
- Tools that help find memory leaks:
 - NetBeans Profiler
 - VisualVM
 - jmap/jhat



ORACLE

Memory Leaks: NetBeans/VisualVM Strategies

- View live heap profiling results while application is running.
- Pay close attention to surviving generations.
 - Increasing object age could indicate a memory leak.
- Use HeapWalker to traverse object references.



ORACLE

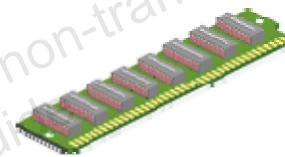
Pay close attention to surviving generations or object age when looking for memory leaks. Surviving generations is the number of different object ages for a given class. An increasing number of surviving generations over a period of time can be a strong indicator of a source of a memory leak.

Remember, the object age is the number of garbage collections that the object has survived.

Memory Leaks: jmap/jhat Strategies

Memory Leaks: jmap/jhat Strategies

- Capture multiple heap profiles and compare footprints, (that is, look for obvious memory usage increases).
- `-XX:+HeapDumpOnOutOfMemoryError`
- Use the `jhat` Object Query Language (OQL) to query with interesting state information.



ORACLE

Capturing multiple heap profiles for comparison could easily be applied to the map scenario discussed previously.

`-XX:+HeapDumpOnOutOfMemoryError`

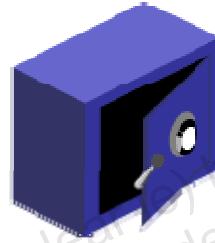
Use this JVM command-line switch when starting an application. It can be used with the
`-XX:HeapDumpPath=<path>/<file>`.

For example, the following `jhat` OQL query returns the number of live HTTP requests for the application:

```
select s from com.sun.grizzly.ReadTask s s.byteBuffer.position > 0
```

Lock-Contention Profiling: Overview

- Use of Java synchronization can lead to highly contended locks.
- Observing high values of voluntary context switches can be an indication of lock contention.
- Collector/Analyzer is very good for identifying Java objects experiencing lock contention.



ORACLE

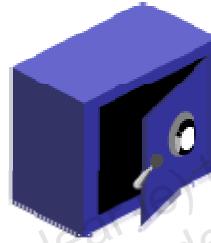
HotSpot and many modern JVMs do optimistic locking within the JVM. Therefore, detecting these locking operations outside the JVM can be challenging. Voluntary context-switching can be an indicator of this sort of activity.

Fortunately, Collector/Analyzer is very good at identifying locking issues.

How to Reduce Lock Contention

How to Reduce Lock Contention

- Use a concurrent data structure introduced in Java SE 5:
 - `java.util.concurrent`
- Partition “guarded” data.
- When writes are infrequent, use `ReentrantReadWriteLock`.
- Favor “synchronized” over `java.util.concurrent` locks for equally weighted guarded reads/writes.



ORACLE

Identify ways to partition the “guarded” data such that multiple locks can be integrated at a finer-grained level as a result of partitioning.

If writes are much less frequent than reads, separate read locks from write locks by using a Java SE 5 `ReentrantReadWriteLock`. This class allows multiple threads to read an object simultaneously. However, only a single write thread has access to the object for an update. In this case, reads are blocked only in the rare instance of a write.

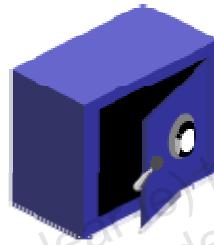
Concurrent data structures might introduce additional CPU utilization overhead and might in some cases not provide as good a performance as a synchronized collection. This is due to JVM optimizations targeted at synchronized usage. Therefore, compare the approaches with meaningful workloads before selecting which approach is best.

Concurrent data structures tend to optimistically update data. The concurrent structure anticipates the data states before and after writing. This can result in the JVM spinning in a tight loop while waiting for the expected state to occur.

Biased Locking

Biased Locking

- HotSpot JVM biased locking may also improve synchronized collection performance.
 - `-XX:+UseBiasedLocking`
- Must be explicitly enabled in JDK 5 versions
- Enabled by default in JDK 6 versions
 - To disable: `-XX:-UseBiasedLocking`



ORACLE

Biased Locking: In most cases, the last thread that used a lock is the most likely to request that lock again. So the JVM will bias the lock to the last thread that held the lock.

- Introduced in JDK 5.0_06
- Improved in JDK 5.0_08

When Should You Disable Biased Locking?

For example, suppose that you have a thread pool that frequently hands off work to worker threads. The lock in this case is spread across a number of threads. So if application profiling indicates that a lock is “hot” and is being contended for by the worker threads, it might be good to disable biased locking in this scenario.

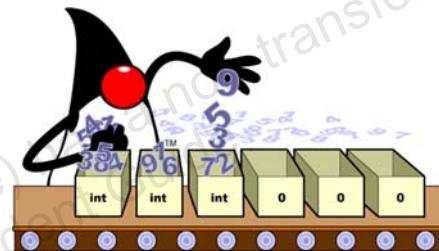
However, in general, biased locking being enabled helps the vast majority of Java applications.

Inlining Effect

Inlining Effect

Inlining: A JIT optimization where the code from a called method is included in the calling method.

- In rare cases, inlining produces profiling results that are:
 - Misleading
 - Confusing
- In those cases, disable inlining.
- `-XX:-Inline`
 - Disables inlining



Inlining: A JIT optimization where the code from a called method is included in the calling method. So, in effect, the method that was called no longer exists as its code is now part of the method that originally called it. For example, you may see a method that merely calls other methods having a high CPU utilization when it should not.

To disable inlining, use the following JVM command-line switch: `-XX:-Inline`

Note: Disabling inlining may distort the “actual” performance profile of your application. In effect, you are testing a different application with this command-line option.

Caution: This command-line option should be used only for testing purposes.

Identifying Anti-Patterns

Identifying Anti-Patterns

- Identifying anti-patterns in heap profiles
- Identifying anti-patterns in method profiles



ORACLE

Anti-Pattern: Some repeated pattern of action, process, or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results.

Anti-Patterns in Heap Profile

Anti-Patterns in Heap Profile

- Large number of `String` or `char[]` allocations in heap profile:
 - Possible overallocation of `String`
 - Possible benefit from `StringBuilder` or `StringBuffer`
- Possible `StringBuilder` or `StringBuffer` resizing
 - Set better initial size to reduce `char[]` allocations.
- Observing `StringBuffer` too often in older application
- Reducing `char[]` and `String` allocations is likely to reduce garbage-collection frequency.



ORACLE

As discussed previously, constant `StringBuilder` or `StringBuffer` resizing can negatively impact performance by increasing the number of garbage collections. `StringBuffer` is synchronized, where `StringBuilder` is not. There may be cases where older applications use `StringBuffer` but could be updated to use `StringBuilder`.

Anti-Patterns in Heap Profiles

Anti-Patterns in Heap Profiles

- Observing `Hashtable` in heap profile
 - Possible candidate for `HashMap` if synchronized access is not required
 - Possible candidate for `ConcurrentHashMap` if synchronized access is required
 - Further partitioning of data stored in `Hashtable` may lead to finer-grained synchronized access and less contention.



ORACLE

ConcurrentHashMap is a data structure introduced in Java 5 that better supports synchronized access.

Anti-Patterns in Heap Profiles

Anti-Patterns in Heap Profiles

- Observing `Vector` in heap profile:
 - Possible candidate for `ArrayList` if synchronized access is not required
 - If you require synchronized access, consider using the following depending on its usage: `LinkedBlockingDeque`, `ArrayBlockingQueue`, `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, or `PriorityBlockingQueue`.



ORACLE

`LinkedBlockingDeque`, `ArrayBlockingQueue`, `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, or `PriorityBlockingQueue` are all improved synchronized data structures added in Java 5.

Anti-Patterns in Heap Profiles

Anti-Patterns in Heap Profiles

- Exception object allocations:
 - Do not use exceptions for flow control.
 - Use flow control constructs such as `if/then/else`, `return`, or `switch`.
- Generating stack traces are expensive operations.



ORACLE

Anti-Patterns in Method Profiles

- Observing `Map.containsKey(key)` in profile:
 - If null keys are allowed in the `Map`, and null keys are not being used as valid keys in the `Map`
 - Look at stack traces for unnecessary call flows that look like:

```
if (map.containsKey(key))
    value = map.get(key);
```
 - Value will be null if a key is not found via `map.get(key)`
 - Other use cases using `Map` methods such as `put(key, value)` or `remove(key)` may be eliminated, too.

ORACLE

Anti-Patterns in Method Profiles

Anti-Patterns in Method Profiles

- Observing high system CPU times:
 - Look for monitor contention.
 - Monitor contention and high system CPU time have a strong correlation.
 - Consider alternatives to minimize monitor contention.
 - Look for opportunities to minimize the number of system calls.
 - An example is: Read as much data as is ready to be read by using nonblocking SocketChannels.
 - Reduction in system CPU time will likely lead to better application throughput and response time.

ORACLE

Monitor Contention: A situation where multiple threads hold global locks too frequently or too long.

Summary

Summary

In this lesson, you should have learned how to:

- Describe the key features of NetBeans Profiler, Oracle Solaris Studio, and jmap/jhat.
- Profile CPU usage to improve application performance
- Profile the JVM heap to improve application performance
- Profile applications to find memory leaks
- Profile applications to identify lock contention
- Identify anti-patterns in heap profiling
- Identify anti-patterns in method profiling

ORACLE

Garbage Collection Schemes

Chapter 6

6 **Garbage Collection Schemes**

ORACLE

Objectives

Objectives

After completing this lesson, you should be able to describe:

- What garbage collection is
- The advantages of generational garbage collectors over non-generational garbage collectors
- GC performance metrics
- The various garbage collection algorithms
- Various types of garbage collectors
- JVM Ergonomics

ORACLE

Garbage Collection Basics

Garbage Collection Basics

- Garbage collection is a way in which Java recollects the space occupied by loitering objects.
- In general, a garbage collector is responsible for three tasks:
 - Allocating memory for new objects
 - Ensuring that any referenced objects (live objects) remain in memory
 - Recovering memory used by objects that are no longer reachable (dead objects)

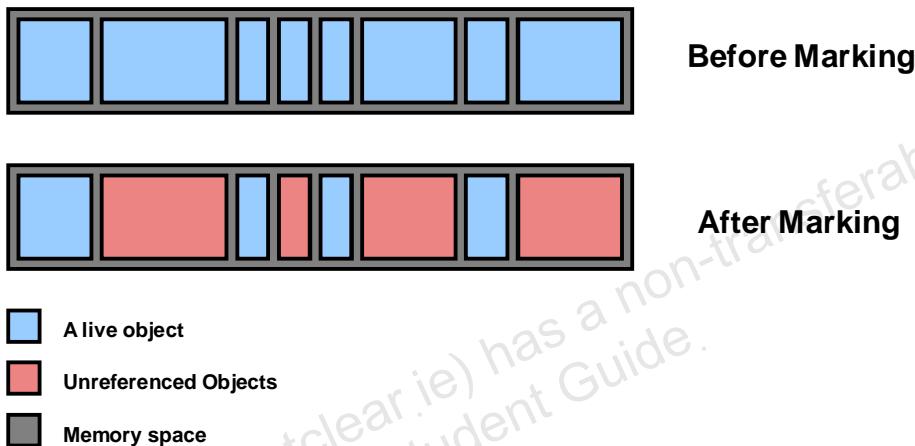


ORACLE

A typical garbage collector is responsible for a high-speed memory allocation with optimal memory utilization and makes sure that there are no long-term fragmentation problems. To understand how to use the garbage collector efficiently and the various performance problems that you may face while running a Java program in a garbage collected environment, it is important to understand the basics of garbage collection and how garbage collectors work.

Garbage Collection Basics

- The garbage collector first performs a task called marking.
- Each object the garbage collector meets is marked as being used, and will not be deleted in the sweeping stage.



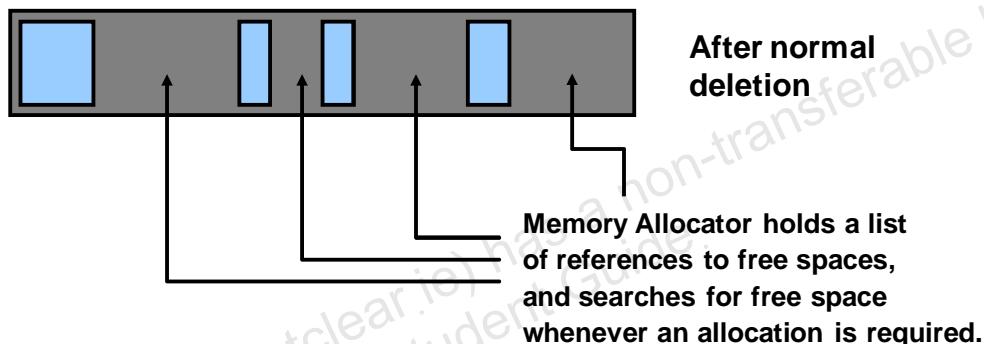
ORACLE

The first step that the garbage collector performs is called marking. The garbage collector iterates one-by-one through the application graph, checks if the object is being referenced, and if so marks the object as being used. The marked objects will not be deleted in the sweeping stage.

Normal Deletion

Normal Deletion

- The *sweeping* stage is where the deletion of objects take place.
- The traditional way is to let the allocator methods use complex data structures to search the memory for the required free space.



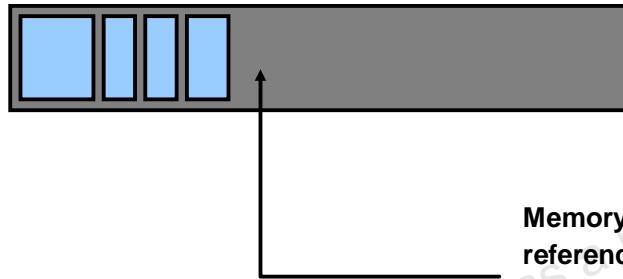
ORACLE

The deletion of objects happens in the sweeping stage. The traditional and easiest way is to mark the space as free and let the allocator use complex data structures to search the memory for the required free space.

Deletion with Compacting

Deletion with Compacting

- Compact the memory by moving objects close to each other.
- Object allocation is faster.



After normal
Deletion with
compacting

Memory Allocator holds the
reference to the beginning of free
space, and allocated memory
sequentially then on.

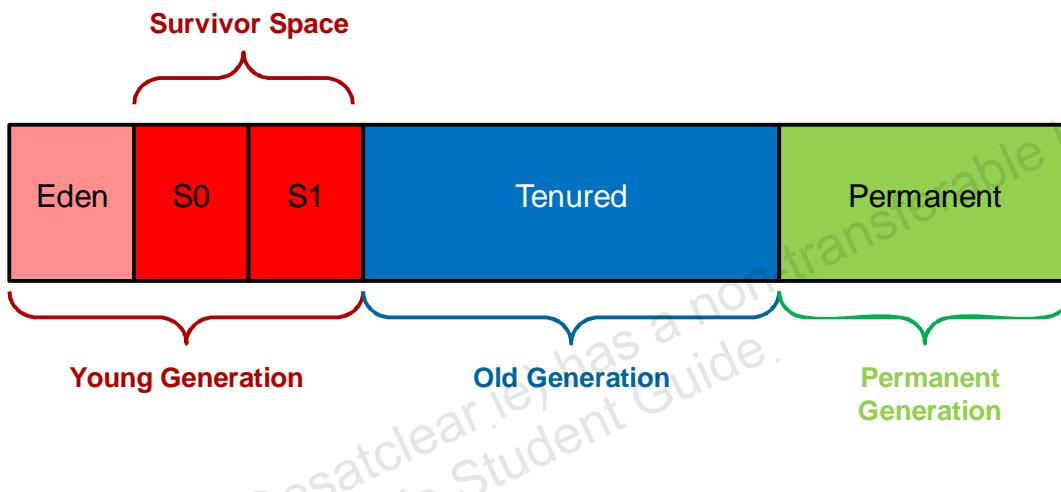
ORACLE

It is obvious that the traditional way of freeing memory has many problems associated with it. An improved way is by providing a defragmenting system that compacts memory by moving objects closer to each other and removes fragments of free space, if any. In this way, the allocation of future objects is much faster.

Generational Garbage Collection

Generational Garbage Collection

- HotSpot uses what is termed “generational collectors.”
- HotSpot Java heap is allocated into generational spaces.



HotSpot uses a type of garbage collection that is termed generational; what that means is the Java heap is partitioned into generational spaces. The default arrangement of generations (for all collectors with the exception of the throughput collector) looks something like the image above.

There are three types of generational spaces:

- Young Generation
- Tenured Generation
- Permanent Generation

Each of these spaces and the role they play are discussed in the subsequent slides.

Why Generational GC?

Why Generational GC?

- Non-Generational GC will just iterate over every object in the heap:
 - Increased objects reduces performance
 - Ignores typical object behavior
- Generational GC improves upon this.



ORACLE

The non-generational garbage collectors iterate over every object in the heap and check whether or not the object has some active references to it. As the number of objects increases in the heap, this process would take a longer time to complete, and therefore, would be inefficient.

A careful observation of a typical object would tell us the following two characteristics:

- Most allocated objects will die young.
- Few references from older to younger objects exist.

To take advantage of this, the Java HotSpot VM splits the heap into two physical areas, which are called generations.

Generational Garbage Collection: Major Spaces

Generational Garbage Collection: Major Spaces

- The memory space is divided into three sections:
- Young generation (for young objects)
 - Eden
 - A “from” survivor space
 - A “to” survivor space
- Tenured (old) generation (for old objects)
- Permanent generation (meta data, classes, and so on)



Major Spaces

At initialization, a maximum address space is virtually reserved but not allocated to physical memory unless it is needed. The complete address space reserved for object memory can be divided into the young and tenured generations.

The young generation consists of eden plus two survivor spaces. Objects are initially allocated in eden. One survivor space is empty at any time, and serves as a destination of the next, copying collection of any live objects in eden and the other survivor space. Objects are copied between survivor spaces in this way until they are old enough to be tenured, or copied to the tenured generation.

Other virtual machines, including the production virtual machine for the J2SE Platform version 1.2 for the Solaris Operating System, use two equally sized spaces for copying rather than one large eden plus two small spaces. This means the options for sizing the young generation are not directly comparable.

A third generation closely related to the tenured generation is the permanent generation. The permanent generation is special because it holds data needed by the virtual machine to describe objects that do not have an equivalence at the Java language level. For example, objects describing classes and methods are stored in the permanent generation.

Features of Young Generational Space

Features of Young Generational Space

- GCs occur relatively frequently.
- GCs are fast and efficient because young generation space is usually small and likely to contain a lot of short-lived objects.
- Objects that survive some number of young generation collections are promoted to old generation heap space.

ORACLE

All newly allocated objects are allocated in the young generation, which is relatively smaller than the Java heap, and is collected more frequently. Because most objects in it are expected to become unreachable quickly, the number of objects that survive a young generation collection (also referred to as a minor garbage collection) is expected to be low. In general, minor garbage collections are very efficient because they concentrate on a space that is usually small and is likely to contain a lot of garbage objects.

Features of Old Generation Space

Features of Old Generation Space

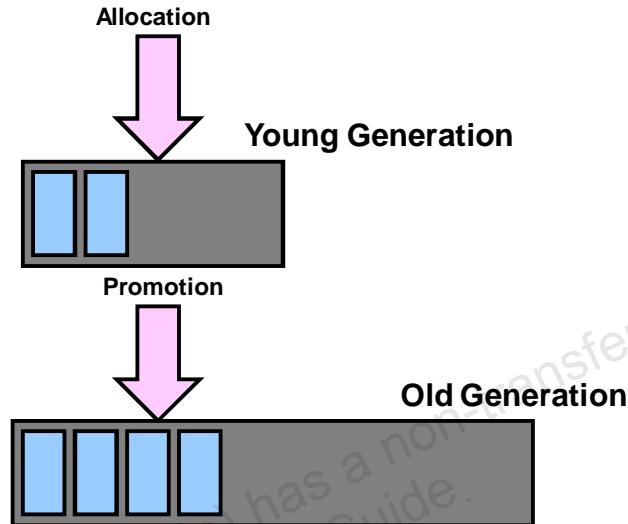
- Typically larger than young generation heap space
- Its occupancy grows more slowly.
- GCs are infrequent but takes significantly longer time to complete than young generational heap space.
 - GCs in old generation space should be minimized.

ORACLE

Objects that are longer-lived are eventually promoted, or tenured, to the old generation. This generation is typically larger than the young generation and its occupancy grows more slowly. As a result, old generation collections (also referred to as major garbage collections or full garbage collections) are infrequent, but when they do occur they can be quite lengthy.

Generational Garbage Collection

Generational Garbage Collection



ORACLE

All newly allocated objects are allocated in the young generation, which is typically small and collected frequently. Because most objects in the young generation are expected to die quickly, the number of objects that survive a young generation collection is expected to be low. In general, minor collections are very efficient because they concentrate on a space that is usually small and is likely to contain a lot of garbage objects. The young generation collection is also called as a minor collection.

Objects that are longer-lived are eventually promoted, or tenured, to the old generation. This generation is typically larger than the young generation and its occupancy grows more slowly. As a result, old generation collections are infrequent, but when they do occur they are quite lengthy. The tenured generation collection is also called major collection.

Garbage Collection Notations

- When the young generation space is full, and young generation GC occurs:
 - Young generation GC is called minor GC or nursery GC.
- When the old generation does not have enough space for the objects that are being promoted, a full GC occurs:
 - Full GC is also called major GC.

ORACLE

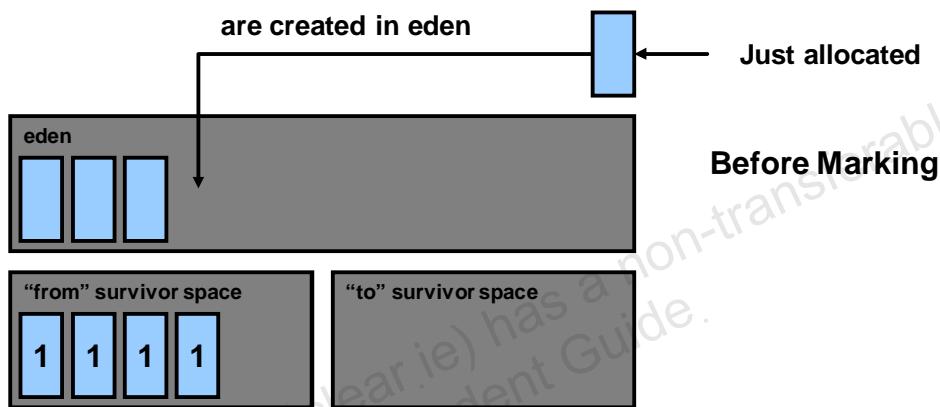
YC: Young Generation

OC: Old Generation

Generational Garbage Collection: Young Collection

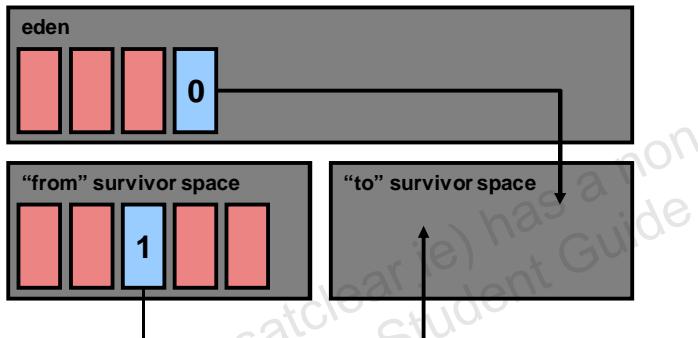
Generational Garbage Collection: Young Collection

- Young generation area consists of an area called eden plus two smaller survival spaces.
- Newly created objects are created in eden.



Generational Garbage Collection: Young Collection

- When the eden space is full, a minor garbage collection event occurs. Live objects in the eden space are copied to a “to” survivor space.
- Additionally, objects in the “from” survivor space are copied to a “to” survivor space.



GC Performance Metric

GC Performance Metric

- The following are the different ways to measure GC Performance:
 - Throughput – % of time not spent in GC over a long period of time
 - Responsiveness – application unresponsive because of GC
 - Footprint – overall memory a process takes to execute
 - Promptness – time between object death, and time when memory becomes available
- There are many right ways to size generations, so make the call based on your applications usage.

ORACLE

There are two primary measures of garbage collection performance. Throughput is the percentage of total time not spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation is generally not needed.) Pauses are the times when an application appears unresponsive because garbage collection is occurring.

Users have different requirements of garbage collection. For example, some consider the right metric for a web server to be throughput, because pauses during garbage collection may be tolerable, or simply obscured by network latencies. However, in an interactive graphics program even short pauses may negatively affect the user experience.

Some users are sensitive to other considerations. Footprint is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate scalability. Promptness is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including remote method invocation (RMI).

There are numerous ways to size generations. The best choice is determined by the way the application uses memory as well as user requirements. Therefore, the virtual machine's choice of a garbage collector is not always optimal, and may be overridden by the user in the form of command-line options.

Choices of Garbage Collecting Algorithms

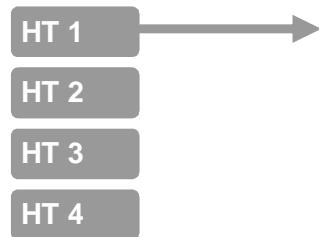
- Serial versus Parallel
- Stop the World versus Concurrent
- Compacting versus Non-compacting versus Copying

ORACLE

Serial versus Parallel

Serial versus Parallel

- In the serial collector, one hardware thread handles the GC task.



- In the parallel collector, multiple hardware threads handle the GC task simultaneously.



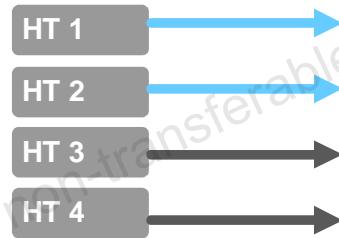
ORACLE

HT – Hardware Thread

Stop the World Versus Concurrent

Stop the World Versus Concurrent

- Stop the World
 - Execution of the application is completely suspended during garbage collection
 - Simpler to implement
 - More pause time
- Concurrent (Mostly)
 - One or more GC tasks can be executed concurrently with the application.
 - Less pause time
 - Additional overhead



ORACLE

Compacting Versus Non-Compacting Versus Copying

Compacting Versus Non-Compacting Versus Copying

- Compacting
 - Move all live objects together and completely reclaim the remaining memory
- Non-compacting
 - Release the space “in-place”
- Copying (sometimes also called Evacuating)
 - Copies objects to a different memory area
 - The source area is empty and available for fast and easy further allocations.

ORACLE

Types of GC Collectors

Types of GC Collectors

Below are the different types of GC Collectors:

- Serial Collector
- Parallel Collector (Throughput Collector)
- Parallel Compacting Collector
- Concurrent Mark-Sweep Collector

ORACLE

The Java Virtual Machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementer's system requirements.

Serial Collector

- Default for client style machines in Java SE 5 and 6
- Both young generation GC and old generation GC are done serially (using a single CPU).
- Serial collectors are used for most of the applications that:
 - Are running on client-style machines
 - Do not have low pause time requirements
- Can be explicitly requested with:
 - `-XX:+UseSerialGC`

ORACLE

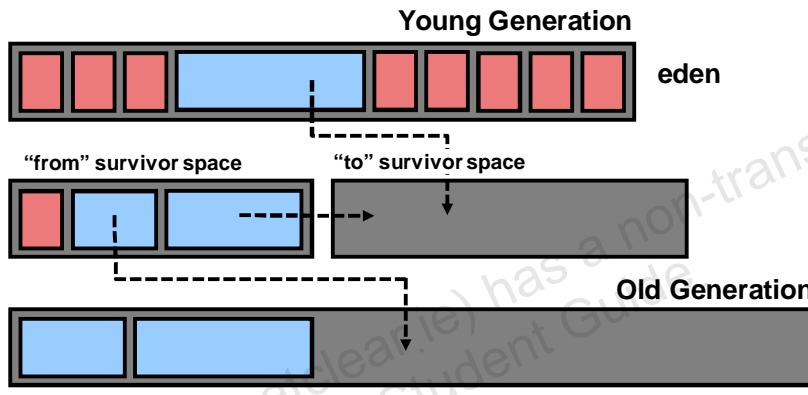
The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work (therefore, its name). Still, on today's hardware, the Serial GC can efficiently manage a lot of non-trivial applications with a few hundred MBs of Java heap, with relatively short worst-case pauses (around a couple of seconds for full garbage collections).

Another popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer. And the Serial GC fits this trade-off nicely.

Serial Collector on Young Generation: Before

Serial Collector on Young Generation: Before

- Live objects are copied from eden to an empty survival space.
- Relatively young live objects in the occupied (From) survival space are copied to the empty (To) survival space whereas relatively old ones are copied to the old generation space directly.

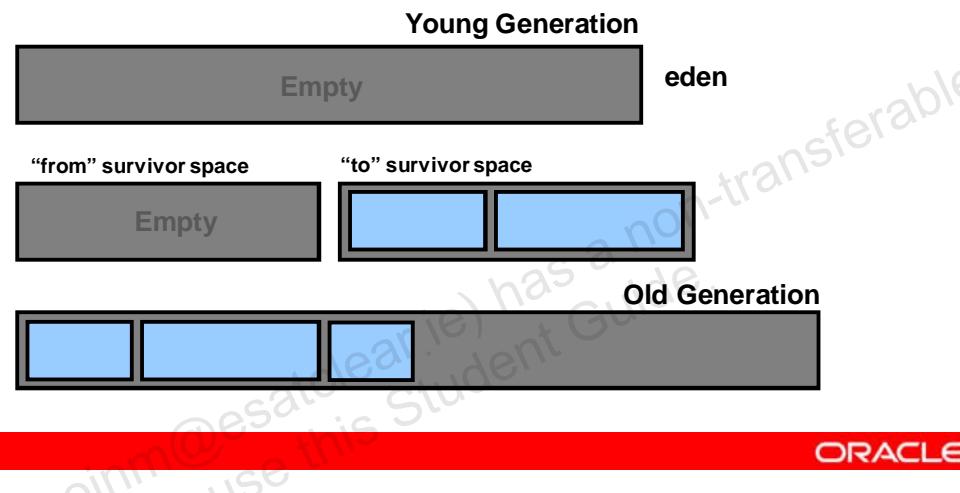


The configuration of the Serial GC is a young generation that operates as described above. Both minor and full garbage collections take place in a Stop the World fashion (that is, the application is stopped while a collection is taking place). Only after garbage collection has finished is the application restarted.

Serial Collector on Young Generation: After

Serial Collector on Young Generation: After

- Both eden and the formerly occupied survival space are empty.
- The formerly empty survival space contains live objects.
- Survivor spaces switch roles.

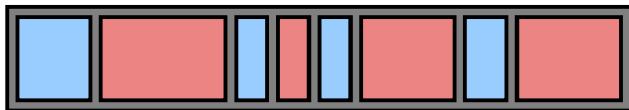


Serial Collector on Old Generation

Serial Collector on Old Generation

The old and permanent generations are collected through a serial mark-sweep-compact collection algorithm.

Start of GC and Compaction



End of GC and Compaction



ORACLE

The configuration of the Serial GC over an old generation is managed by a sliding compacting mark-sweep, also known as a mark-compact garbage collector. The mark-compact garbage collector first identifies which objects are still live in the old generation. It then slides them towards the beginning of the heap, leaving any free space in a single contiguous chunk at the end of the heap. This allows any future allocations into the old generation, which will most likely take place as objects are being promoted from the young generation, to use the fast bump-the-pointer technique.

Parallel Collector: Throughput Matters!

- Uses multiple hardware threads for young generation GC, and thus increases the throughput
- Parallel collectors are used for Java applications that run on machines:
 - With a lot of physical memory
 - With multiple threads
 - That do not have low pause time requirements
- Can be explicitly requested with
 - `-XX:+UseParallelGC`

ORACLE

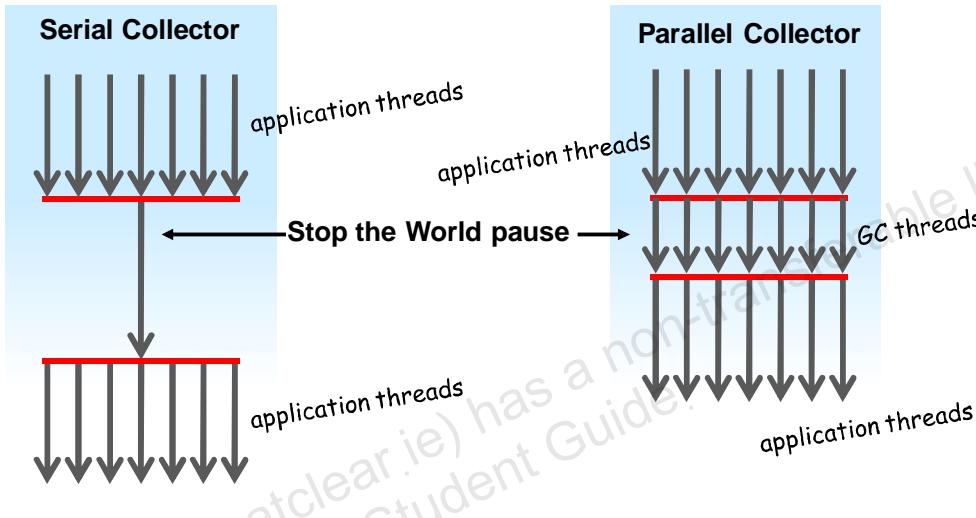
The parallel garbage collector is similar to the young generation collector in the default garbage collector but uses multiple threads to do the collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection. The number of garbage collector threads can be controlled with command-line options. On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested. On a host with two CPUs the parallel garbage collector generally performs as well as the default garbage collector and a reduction in the young generation garbage collector pause times can be expected on hosts with more than two CPUs.

This new parallel garbage collector can be enabled by using command-line product flag `-XX:+UseParallelGC`. The number of garbage collector threads can be controlled with the `ParallelGCThreads` command-line option (`-XX:ParallelGCThreads=<desired number>`). This collector cannot be used with a concurrent low pause collector.

Parallel Collector on Young Generation

Parallel Collector on Young Generation

- Uses a parallel version of young generation collection algorithm of a serial collector:



The parallel young generation collector is similar to the parallel garbage collector (-XX:+UseParallelGC) in intent and differs in implementation. Unlike the parallel garbage collector (-XX:+UseParallelGC) this parallel young generation collector can be used with the concurrent low pause collector that collects the tenured generation.

Note: The old and permanent generations are collected through a serial mark-sweep-compact collection algorithm.

Parallel Compacting Collector

Parallel Compacting Collector

- In the purist sense, the difference between a parallel collector and a parallel compacting collector is “compacting.”
- Parallel collector is enabled via `-XX:+UseParallelGC` or `-XX:+UseParallelOldGC`
- Young generation in HotSpot is considered a copy collector; therefore, there is no need for compaction.
- `-XX:+UseParallelGC` as a parallel collector and `-XX:+UseParallelOldGC` as a parallel compacting collector

ORACLE

In the purist sense, the difference between a parallel collector and a parallel compacting collector is “compacting.” Compacting describes the act of moving objects in a way that there are no holes between objects. After a garbage collection sweep, there may be holes left between live objects. Compacting moves objects so that there are no remaining holes. It is possible that a garbage collector be a non-compacting collector. Therefore, the difference between a parallel collector and a parallel compacting collector could be the latter compacts the space after a garbage collection sweep. The former would not.

A parallel collector also implies a multithreaded garbage collector where it uses multiple threads to perform the garbage collection. A multi-thread parallel compacting garbage collector implies a multithreaded garbage collection and possibly a multithreaded compaction capability too.

In the context of HotSpot, the term parallel collector suggests that the garbage collector is enabled via `-XX:+UseParallelGC` or `-XX:+UseParallelOldGC`. These are also described as the throughput collectors and both can be considered parallel compacting collectors. The former is a multi-thread young generation collector with a single-threaded old generation collector that also does single-threaded compaction of old generation.

Parallel Compacting Collector

Parallel Compacting Collector

- In the purist sense, the difference between a parallel collector and a parallel compacting collector is, of course the "compacting"
- Parallel collector is enabled via `-XX:+UseParallelGC` or `-XX:+UseParallelOldGC`
- Young generation in HotSpot is considered a copy collector, therefore, there is no need for compaction.
- `-XX:+UseParallelGC` as a parallel collector and `-XX:+UseParallelOldGC` as a parallel compacting collector

ORACLE

`-XX:+UseParallelOldGC` is both a multithreaded young generation collector and multithreaded old generation collector. It is also a multithreaded compacting collector. HotSpot does compaction only in the old generation. Young generation in HotSpot is considered a copy collector; therefore, there is no need for compaction. `-XX:+UseParallelGC`

You can think of `-XX:+UseParallelGC` as a parallel collector and `-XX:+UseParallelOldGC` as a parallel compacting collector. But, it is important to realize `-XX:+UseParallelGC` is a multithreaded young generation collector with a single-threaded old generation compacting collector. Garbage collection in HotSpot's young generation is always a copy collector and therefore, compaction is not applicable. In contrast, `-XX:+UseParallelOldGC` is a multithreaded young generation collector and a multithreaded old generation collector. Therefore, it is a multithreaded young generation copy collector and a multithreaded old generation collector that also does multithreaded compaction of old generation.

Concurrent Mark-Sweep (CMS) Collector

- Also called as Low Pause Collector
- CMS collector is used when an application has shorter pause time and can share processor resources with GC when the application is executing.
- Can be explicitly requested with
 - `-XX:+UseConcMarkSweepGC`
 - `-XX:ParallelCMSThreads=<n>`

ORACLE

The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

Note: CMS collector on young generation uses the same algorithm as that of the parallel collector.

CMS Collector Phases

CMS Collector Phases

- Concurrent collector cycle contains the following phases:
 - Initial mark
 - Concurrent mark
 - Remark
 - Concurrent sweep
 - Concurrent reset
- During a concurrent collector cycle, a Java application is paused during the “initial mark and “remark” phases.

ORACLE

CMS Collector on Old Generation

CMS Collector on Old Generation

- Initial mark phase
 - Objects in the tenured generation are “marked” as reachable including those objects which may be reachable from young generation.
 - Pause time is typically short in duration relative to minor collection pause times.
- Concurrent mark phase
 - Traverses the tenured generation object graph for reachable objects concurrently while Java application threads are executing.

ORACLE

Concurrent Mark-Sweep Collector Phases

Phase 1: (Initial Mark) involves stopping all the Java threads, marking all the objects directly reachable from the roots, and restarting the Java threads.

Phase 2: (Concurrent Marking) starts scanning from marked objects and transitively marks all objects reachable from the roots. The mutators are executing during the concurrent phases 2, 3, and 5 and any objects allocated in the CMS generation during these phases (including promoted objects) are immediately marked as live.

CMS Collector on Old Generation

CMS Collector on Old Generation

- Remark
 - Finds objects that were missed by the concurrent mark phase due to updates by Java application threads to objects after the concurrent collector had finished tracing that object
- Concurrent sweep
 - Collects the objects identified as unreachable during marking phases
- Concurrent reset
 - Prepares for next concurrent collection

ORACLE

Concurrent Mark-Sweep Collector Phases (continued)

Phase 3: During the concurrent marking phase mutators may be modifying objects. Any object that has been modified since the start of the concurrent marking phase (and which was not subsequently scanned during that phase) must be rescanned. Phase 3 (Concurrent Precleaning) scans objects that have been modified concurrently. Due to continuing mutator activity the scanning for modified cards may be done multiple times.

Phase 4: (Concurrent Sweep) collects dead objects. The collection of a dead object adds the space for the object to a free list for later allocation. Coalescing of dead objects may occur at this point. Note that live objects are not moved.

Phase 5: (Resetting) clears data structures in preparation for the next collection.

Garbage Collectors: Comparisons

Garbage Collectors: Comparisons

The following table summarizes the trade-offs between the garbage collectors:

(m.sec)	Serial GC	Parallel GC	CMS GC
Parallelism	No	Yes	Yes
Concurrency	No	No	Yes
Young GCs	Serial	Parallel	Parallel
Old GCs	Serial	Parallel	Parallel & Conc

ORACLE

Ergonomics: What It Does

Ergonomics: What It Does

- Evaluates the system and auto-magically choose defaults for the HotSpot JVM. No tuning required!
- Relies on definition of “server class machine”
 - 2 or more processor cores and 2 or more GB of physical RAM
- Server class machines will use -server JIT compiler

ORACLE

New in the J2SE Platform version 1.5 is a feature referred to here as ergonomics. The goal of ergonomics is to provide good performance from the JVM with a minimum of command-line tuning. Ergonomics attempts to match the best selection of the following for an application:

- Garbage collector
- Heap size
- Runtime compiler

This selection assumes that the class of the machine on which the application is run is a hint as to the characteristics of the application (that is, large applications run on large machines). In addition to these selections is a simplified way of tuning garbage collection.

Ergonomics: What It Does

- As of JDK 5, server class machines set:
 - Server JIT compiler
 - Throughput collector
 - Initial heap size ($-Xms$) 1/64th of physical memory up to max of 1 GB
 - Max heap size ($-Xmx$) 1/4th of physical memory up to max of 1 GB
- If not identified as server class
 - Client JIT compiler
 - Default serial collector, same as before

ORACLE

In the Java platform version 5.0, a class of machine referred to as a server-class machine has been defined as a machine with

- Two or more physical processors
- 2 GB or more of physical memory

On server-class machines by default the following are selected:

- Throughput garbage collector
- Heap sizes
- Initial heap size of 1/64 of physical memory up to 1 GB
- Maximum heap size of 1/4 of physical memory up to 1 GB
- Server runtime compiler

Ergonomics

Ergonomics

- Use `-XX:+PrintCommandLineFlags` to tell you what Ergonomics is choosing.
- Example:
 - On 2 socket, 3.0 GHz Hyper threaded Intel, running Solaris with 4 GB RAM:

```
java -XX:+PrintCommandLineFlags -version
-XX:MaxHeapSize=1073626112 (1 GB)
-XX:+PrintCommandLineFlags
-XX:+UseParallelGC
Java HotSpot(TM) Server VM (build 1.6.0_02-b04, mixed mode)
```

ORACLE

Summary

Summary

In this lesson, you should have learned:

- The various garbage collection schemes
- The advantages of generational garbage collectors over non-generational garbage collectors
- GC performance metrics to consider
- The various garbage collection algorithms
- JVM Ergonomics

ORACLE

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

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

Garbage Collection Tuning

Chapter 7

Garbage Collection Tuning

7

Garbage Collection Tuning

ORACLE

Objectives

Objectives

After completing this lesson, you should be able to:

- Tune the garbage collector
- Select the garbage collector that best fits application characteristics
- Interpret GC output

ORACLE

Garbage Collectors: Recap

- The advantage of not having to deal with memory management issues, as is the case with C/C++
- Historically, garbage collection had often been the most common source attributable to poor Java application performance.
- It still can be a source of poor application performance. It just does not seem to get as much attention as it used to.
- To tune the JVM's garbage collectors, you need to understand the basics of how garbage collection works.

ORACLE

A typical garbage collector is responsible for a high-speed memory allocation with optimal memory utilization and makes sure that there are no long-term fragmentation problems. To understand how to use the garbage collector efficiently and the various performance problems that you may face while running a Java program in a garbage collected environment, you must understand the basics of garbage collection and how garbage collectors work.

In general, a garbage collector is responsible for three tasks:

- Allocating memory for new objects
- Ensuring that any referenced objects (live objects) remain in memory
- Recovering memory used by objects that are no longer reachable (dead objects)

The first step that the garbage collector performs is called marking. The garbage collector iterates one-by-one through the application graph, checks if the object is being referenced, and if so, marks the object as being used. The marked objects will not be deleted in the sweeping stage.

Garbage Collection: Myth

Garbage Collection: Myth



Programmer: I thought that I did not need to worry about memory allocation. Java does that for me.



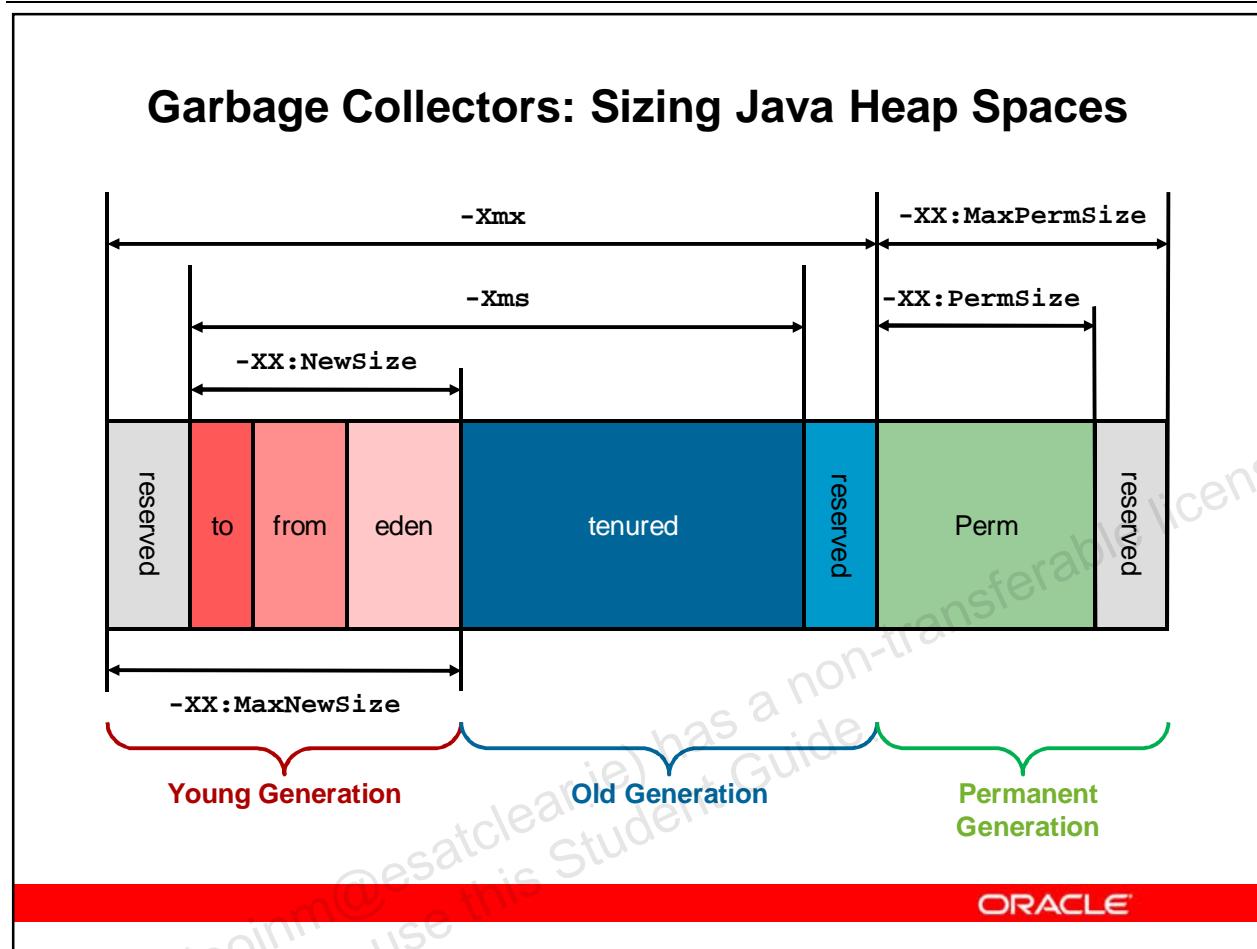
Java Performance Tuning Engineer: That is both true and false. That is the most common perception, though.

Java does much low-level memory allocation and deallocation and also comes with a garbage collector. However, Java does not prevent you from using excessive amounts of memory.

ORACLE

Object creation has a cost, a cost in terms of CPU usage as well as memory. The garbage collector generally takes care of the deallocation and recycling of the memory used by the objects but takes a significant amount of memory and CPU time. Responsible coding practices like avoiding extra temporary objects and unnecessarily creating objects at random can lead to a significant speed gain.

Garbage Collectors: Sizing Java Heap Spaces



The various options available to size the heap are as follows:

- **-Xmx<size>**, max size of Java heap, (young generation + tenured generation)
- **-Xms<size>**, initial size of Java heap (young generation + tenured generation)
- Applications with emphasis on performance usually set **-Xms** and **-Xmx** to the same value.
- When **-Xmx ! = -Xms**, Java heap growth or shrinkage requires a full garbage collection.
- **-Xmn<size>**, size of young generation heap space

Garbage Collectors: Sizing Java Heap Spaces

Garbage Collectors: Sizing Java Heap Spaces

Heap Parameters and Their Effect on Memory Partition	
-Xmx<size>	Maximum size of the Java heap
-Xms<size>	Initial heap size
-Xmn<size>	Sets the initial and the maximum heap size to same
-XX:MaxPermSize=<size>	Maximum size of permanent generation
-XX:PermSize=<size>	Initial size of the permanent generation
-XX:MaxNewSize=<size>	Maximum size of the young generation
-XX:NewSize=<size>	Initial size of the young generation
-XX:NewRatio=<size>	Ratio of young generation space to tenured space

ORACLE

Garbage Collectors: Sizing Java Heap Spaces (continued)

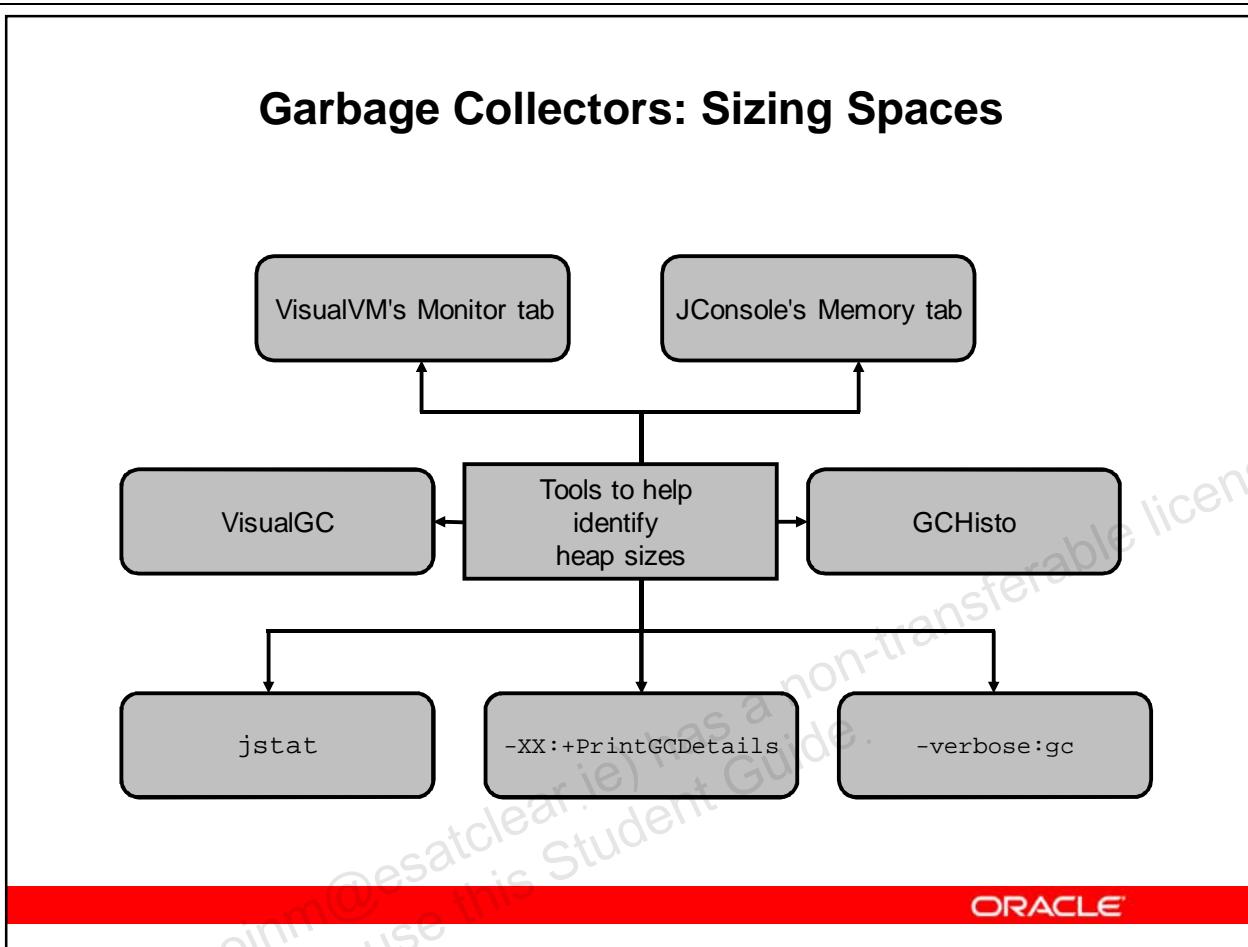
Note

- -XX:NewSize=<size>, initial size of young generation space
- -XX:MaxNewSize=<size>, maximum size of young generation space
- -XX:NewRatio=<size> ratio of young generation space to tenured space
- -XX:PermSize=<size>, initial size of permanent generation space
- -XX:MaxPermSize=<size>, maximum size of permanent generation space

Applications with emphasis on performance tend to use -Xmn to size the young generation because it combines the use of -XX:MaxNewSize and -XX:NewSize and almost always explicitly sets -XX:PermSize and -XX:MaxPermSize to the same value.

A growing or shrinking permanent generation space requires a full garbage collection.

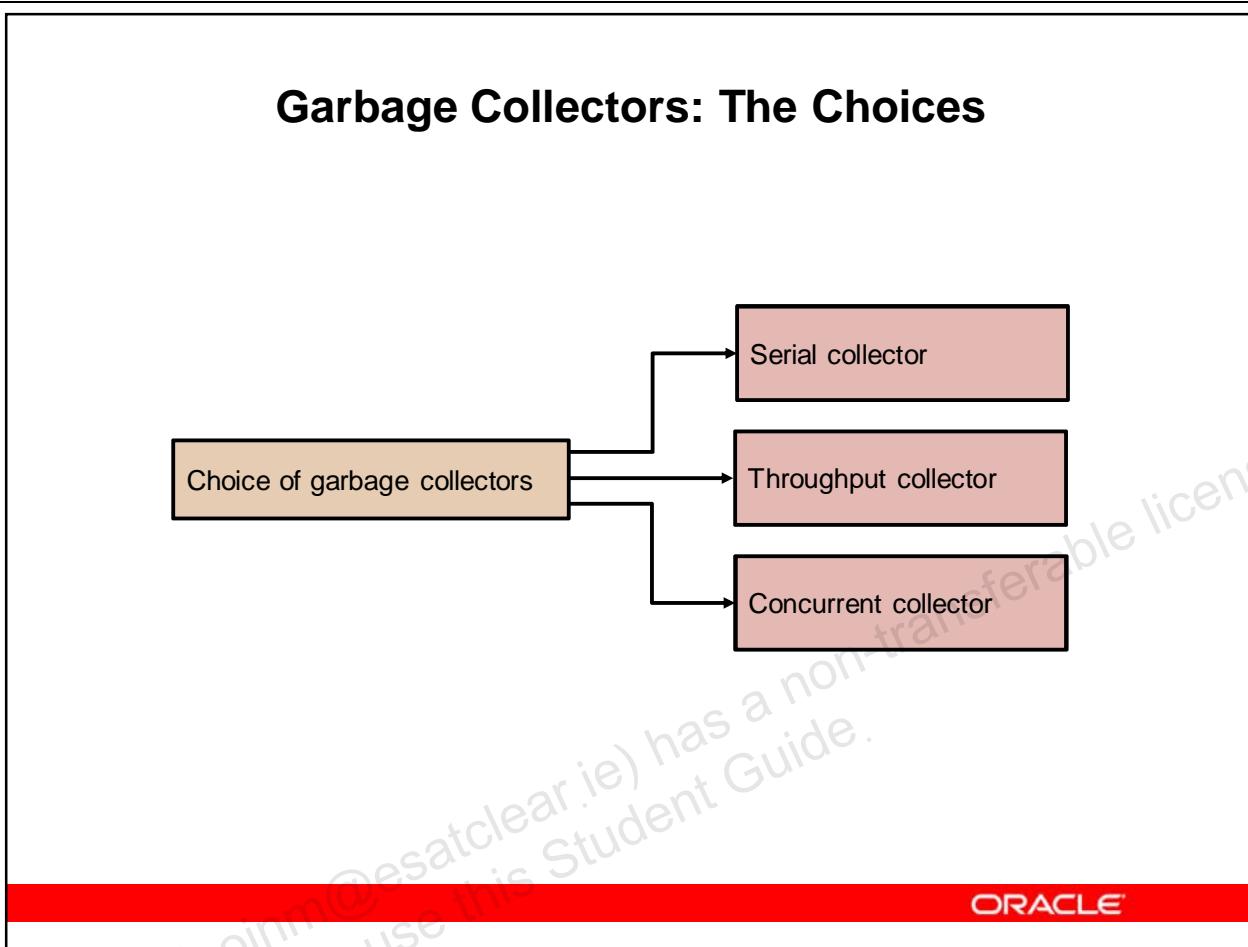
Garbage Collectors: Sizing Spaces



Tools to Help Identify Heap Sizes

There are a number of tools to size heap spaces. Some of the most frequently used tools include:

- VisualVM's Monitor tab
- Jconsole's Memory tab
- VisualGC's heap space sizes
- GCHisto
- jstat's gc options
- -verbose:gc's heap space sizes
- -XX:+PrintGCDetails heap space sizes



In the Java Standard Edition Platform version 1.5, there are three collectors other than the default serial collector:

- Serial collector
- Throughput collector
- Concurrent collector

The fourth one is called Increment Concurrent Collector, which is only recommended for small Java heaps and small number of hardware threads type of systems. It's not meant for large Java heaps and systems with a large number of hardware threads.

Each is a generational collector that has been implemented to emphasize the throughput of the application or low garbage collection pause times.

Garbage Collectors: The Choices

- The major factors that influence the choice of garbage collectors are:
 - Throughput
 - Responsiveness

Throughput

- How much work can be done in a given interval of time

Responsiveness

- Maintaining an elapsed time limit in which the application must respond

ORACLE

Application Throughput Versus Responsiveness

Throughput: Emphasis on how much work can be performed in a given interval of time with no concern of application pauses due to garbage collection

Responsiveness: Emphasis on maintaining an elapsed time limit in which the application must respond to interactions and/or stimuli, or number times a time limit can be exceeded per some interval of time

Garbage Collectors: Serial Collector

Garbage Collectors: Serial Collector

Description

- Enabled with `-XX:+UseSerialGC`
- Single-threaded young generation collector
- Single-threaded tenured generation collector

Suitability

- Well suited for single processor core machines
- Well suited for configurations of one-to-one JVM to processor core configuration
- Tends to work well for applications with small Java heaps

ORACLE

Serial collector is enabled with `-XX:+UseSerialGC`. Both the young generation and the tenured generation serial collector are single threaded (stops all application threads). Serial collector was the first GC in HotSpot and probably the most stable as a result of it being around the longest.

Serial collectors are well suited for:

- Single-processor core machines
- Configurations of one-to-one JVM to processor core configuration
- Applications with small Java heaps, that is, less than 128 MB; may work well up to 256 MB.

Garbage Collectors: Serial Collector

- Events that initiate a serial collector garbage collection:
 - Eden space is unable to satisfy an object allocation request.
Results in a minor garbage collection event
 - Tenured generation space is unable to satisfy an object promotion coming from young generation.
 - Results in a full GC
 - An explicit invocation or call to `System.gc()`

ORACLE

`System.gc()` will always force a full FC.

Garbage Collectors: Serial Collector

- Good throughput performance can be realized with the serial collector if:
 - Well-tuned Java heap spaces are on Java applications with small Java heaps (that is, smaller than 100-MB heaps).
 - The target platform has small number of virtual processors.
 - The number of JVMs deployed on a multicore processor platform is equal or greater than the number of multicore processors.
 - Bind JVMs to processors or processor sets for best results.

ORACLE

Garbage Collectors: Throughput Collector

Garbage Collectors: Throughput Collector

Description

- Multi-threaded young generation space collectors
[-XX:+UseParallelGC]
- Multi-threaded tenured generation space collector
[-XX:+UseParallelOldGC]
- -XX:+UseParallelOldGC also enables
-XX:+UseParallelGC

Suitability

- Significantly reduces garbage collection overhead on multi-core processor systems
- Well suited for applications running on multiprocessor or multi-core systems

ORACLE

Throughput Collector

The throughput collector is a multithreaded young generation space collector enabled with -XX:+UseParallelGC. JDK 5.0_06 introduced the multithreaded tenured generation space collector enabled with -XX:+UseParallelOldGC. This command also enables -XX:+UseParallelGC.

The throughput collector can significantly reduce garbage collection overhead on multicore processor systems as they execute in parallel and therefore, they are well suited for applications running on multiprocessor or multicore systems.

Garbage Collectors: Throughput Collector

- Managing collector threads:
 - Number of parallel throughput collector threads controlled by
-XX:ParallelGCThreads=<N>
 - Defaults to Runtime.availableProcessors(). In a JDK 6 update release, the number of threads created is 5/8ths the available number of processors.
 - In multiple JVM per machine configurations, setting -XX:ParallelGCThreads=<N> lower is likely to yield better results.

ORACLE

Garbage Collectors: Throughput Collector

- Multiple JVM strategies
 - Bind JVMs to processor sets.
 - Works well for JVMs that tend to have equal load because idle processors outside a processor set are not available to other JVMs outside the processor set.
 - **Note:** `Runtime.availableProcessors()` reports the number of processors in a processor set.
 - Create Solaris zones and run a JVM (or possibly more than one JVM) per zone.

ORACLE

Throughput Collector

Solaris Zones is an implementation of operating system-level virtualization technology for x86 and SPARC systems first made available in 2005 as part of Solaris 10. It is present in newer OpenSolaris-based distributions, like OpenIndiana and Solaris 11 Express.

Garbage Collectors: Throughput Collector

- Events that initiate a minor garbage collection:
 - The eden space is unable to satisfy an object allocation request. Results in a minor garbage collection event
- Events that might initiate a full garbage collection:
 - The tenured generation space is unable to satisfy an object promotion coming from young generation.
 - An explicit invocation or call to `System.gc()`

ORACLE

Garbage Collectors: Throughput Collector

Garbage Collectors: Throughput Collector

- Good throughput performance can be realized with the throughput collector if:
 - Pause-time requirements are less important than throughput
 - Java heap spaces are well tuned
 - The application runs on a multicore system
 - Young and tenured generations both use multithreaded collectors, that is, they use `-XX:+UseParallelOldGC` and `-XX:+UseParallelGC`.
 - If full GC events cannot be avoided, a multithreaded old generation collector can reduce the length of full GC events with `-XX:+UseParallelOldGC`.

ORACLE

Other throughput performance recommendations include:

- Bind JVMs to processor sets in multiple JVM configurations for best results.
- Consider `-XX:+BindGCTaskThreadsToCPUs` when scaling across multiple cores.
- Use of `-XX:+UseParallelGC`, the multithreaded young generation collector, and young generation heap are sized so that only long-lived objects are tenured to the old generation and full GC events can be avoided.

Note: `-XX:+UseParallelOldGC` also enables `-XX:+UseParallelGC`.

Garbage Collectors: Concurrent Collector

Description

- Single-threaded tenured space collector
- Enabled with `-XX:+UseConcMarkSweepGC`
- Parallel, multi-threaded young generation collector [Enabled by default]

Suitability

- Well suited when application responsiveness is more important than application throughput

ORACLE

Concurrent Collector

Concurrent collector is a single-threaded tenured space collector that runs mostly concurrent with Java application threads and is enabled with `-XX:+UseConcMarkSweepGC`. A parallel, multithreaded young generation collector is enabled by default.

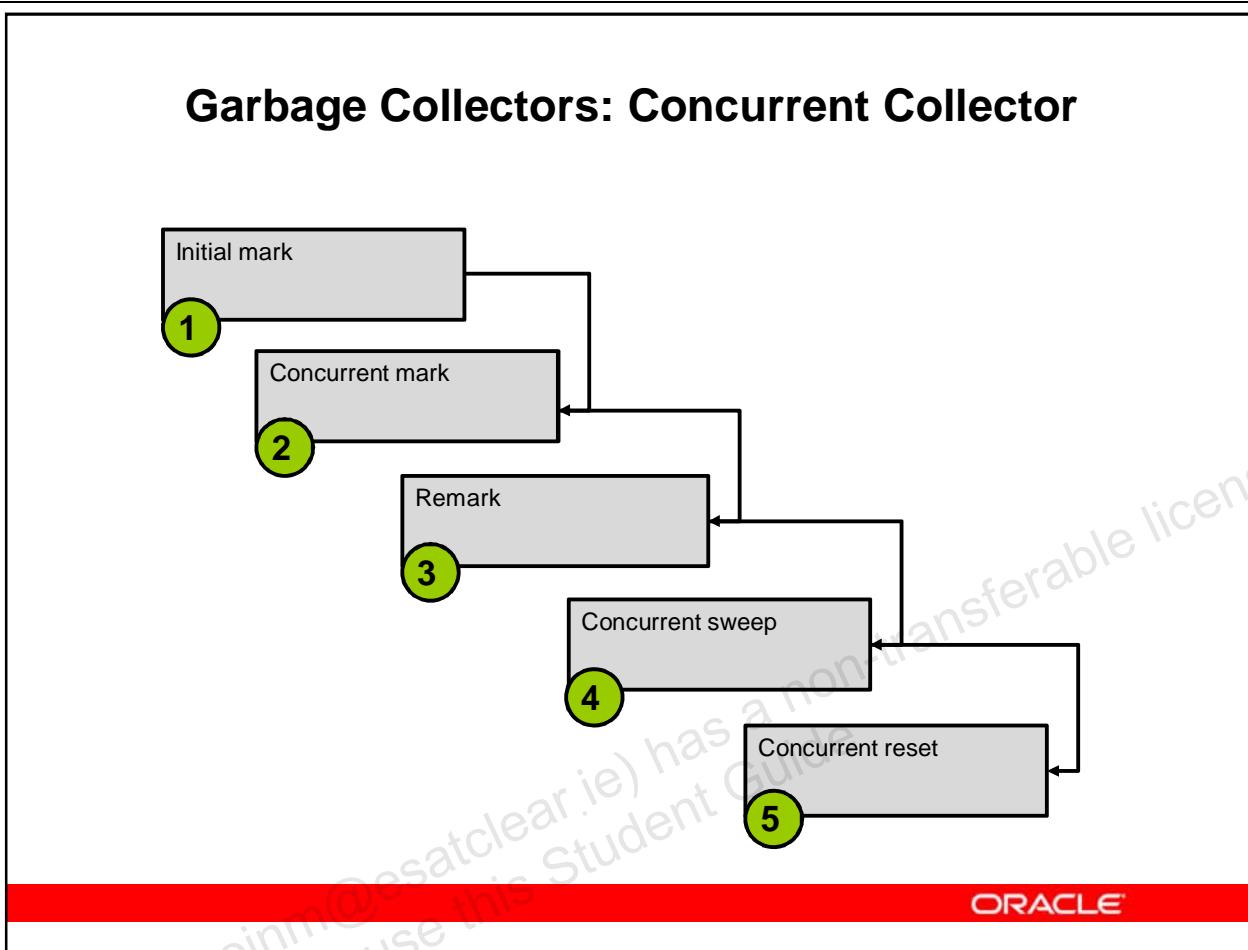
This collector is well suited when application responsiveness is more important than application throughput.

Garbage Collectors: Concurrent Collector

- Issues to be aware of:
 - The cost of the concurrent collections is the additional overhead of more memory and CPU cycles.
 - Takes an additional 20% old generation space
 - Concurrent mode failure can occur when objects are copied to the tenured space faster than the concurrent collector can collect them (“loses the race”).
 - Concurrent mode failure can also occur from tenured space fragmentation.
 - The corrective action by the JVM is to perform a full garbage collection with compaction, which will block all Java application threads.

ORACLE

Garbage Collectors: Concurrent Collector



Garbage Collectors: Concurrent Collector (continued)

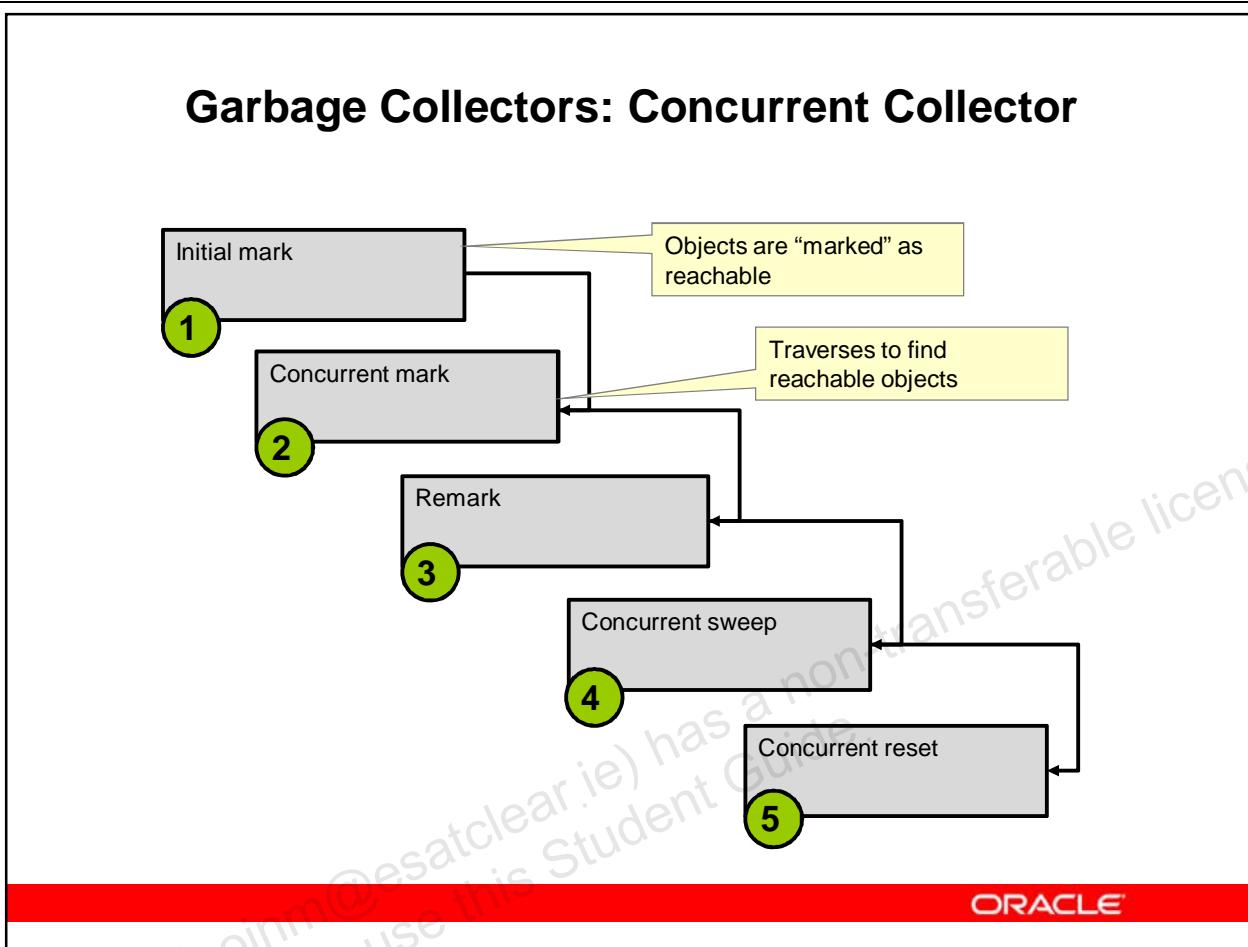
Concurrent Collector Phases

The concurrent collector cycle contains the following phases:

1. Initial mark
2. Concurrent mark
3. Remark
4. Concurrent sweep
5. Concurrent reset

During a concurrent collector cycle, a Java application is paused during the “initial mark” and “remark” phases.

Garbage Collectors: Concurrent Collector



Garbage Collectors: Concurrent Collector (continued)

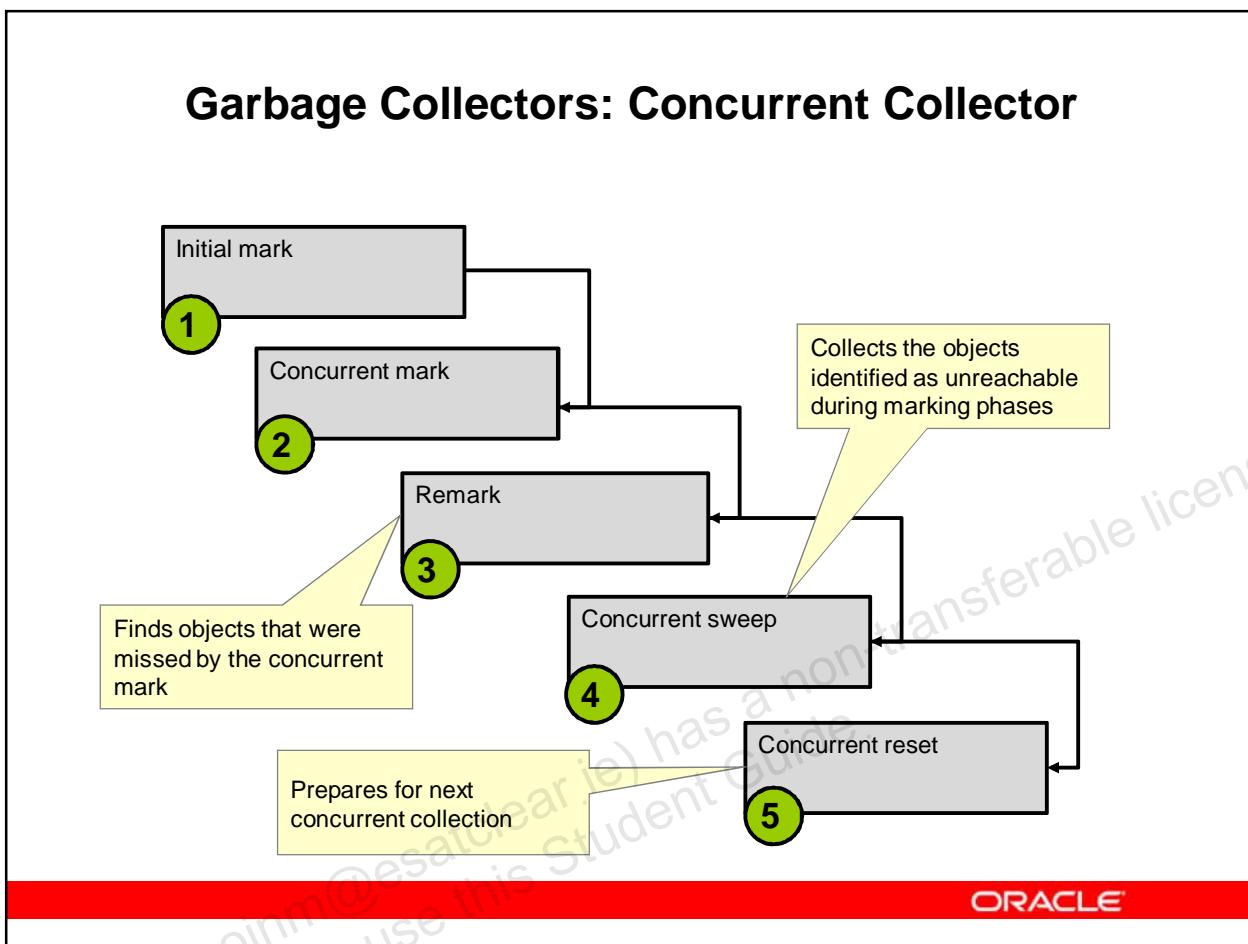
Initial Mark Phase

- Objects in the tenured generation are "marked" as reachable including those objects that may be reachable from the young generation.
- Pause time is typically short in duration relative to minor collection pause times.

Concurrent Mark Phase

Traverses the tenured generation object graph for reachable objects concurrently while Java application threads are executing.

Garbage Collectors: Concurrent Collector



Garbage Collectors: Concurrent Collector (continued)

Remark

- Finds objects that were missed by the concurrent mark phase due to updates by Java application threads to objects after the concurrent collector had finished tracing that object

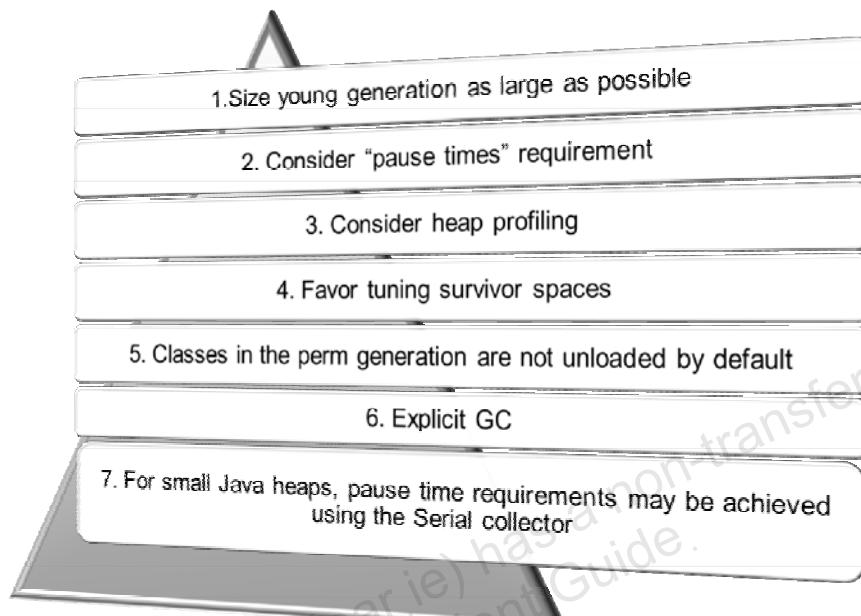
Concurrent Sweep

- Collects the objects identified as unreachable during marking phases

Concurrent Reset

- Prepares for next concurrent collection

Garbage Collectors: Concurrent Collector



ORACLE

Heap Space Sizing Considerations

1. Size the young generation as large as possible with the goal to tenure only long-lived objects to the old generation space and stay within pause-time requirements.
2. If it is difficult to meet pause-time requirements and have CPU available, try sizing young generation spaces smaller (eden and survivor spaces).
3. Realize that sizing the young generation smaller will put more pressure on the old generation concurrent collector because objects will be tenured to the old generation at a faster rate. This may result in an increase in old generation space fragmentation.

Note: As much as 20% additional tenured generation space may be required for floating garbage. Floating garbage consists of objects that are found to be reachable by the concurrent garbage collector, which may become unreachable by the time a concurrent garbage collection cycle finishes.

4. If you are unable to meet pause-time requirements and you cannot avoid full GC events, consider heap profiling to reduce object allocations.
5. Favor tuning survivor spaces rather than –
`XX:CMSInitiatingOccupancyFraction=#` to reduce floating garbage and remark phase times.

6. To date, classes will not by default be unloaded from permanent generation when using the concurrent collector unless explicitly instructed to do so by using both of the following:

- -XX:+CMSClassUnloadingEnabled
- -XX:+PermGenSweepingEnabled

Note: The second switch is not needed in post HotSpot 6.0u4 JVMs.

7. If relying on explicit GC and you want them to be concurrent, use:

- -XX:+ExplicitGCInvokesConcurrent (requires 1.6.0 and later)
- -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

These switches are for permanent generation concurrent collections and for class unloading (requires JDK 1.6.0u4 or later).

8. For small Java heaps, pause-time requirements may be achieved by using the serial collector. Consider the serial collector for Java heaps to 128 MB and possibly up to 256 MB. The serial collector is easier to tune than CMS.

Note

When using `-XX:CMSInitiatingOccupancyFraction`, you should also set `-XX:+UseCMSInitiatingOccupancyOnly`. Without the latter, only the first CMS cycle will use the value from `-XX:CMSInitiatingOccupancyFraction`. The extra switch recommends that you use the `-XX:CMSInitiatingOccupancyFraction` all the time.

Garbage Collectors: Concurrent Collector

- Events initiating concurrent collection cycle:
 - Ideally, the cycle needs to start early enough so that the collection finishes before tenured space becomes full.
 - The concurrent collector will also start if the occupancy of the tenured space exceeds an initiating occupancy percentage threshold.

ORACLE

Garbage Collectors: Concurrent Collector (continued)

Because full garbage collections are expensive, the concurrent collector estimates the time remaining until tenured space is used up and also estimates the amount of time needed to complete a concurrent collection cycle based on recent history.

The concurrent collector will also start if the occupancy of the tenured space exceeds an initiating occupancy percentage threshold. The default value is 60% and may change from release to release. Tune `-XX:CMSInitiatingOccupancyFraction=n` where n is the percentage of the tenured space size.

Note: Minor collection events occur as they do with throughput and serial collectors.

Garbage Collectors: Concurrent Collector

- Good responsiveness performance can be realized with the concurrent collector if:
 - Pause time requirements are more important than throughput
 - Java heap spaces are well tuned
 - By default the concurrent collector is a multithreaded young generation collector

ORACLE

The concurrent collector is likely to require larger tenured heap space sizing than other collectors due to heap fragmentation and floating garbage. The concurrent collector by default is a multithreaded young generation collector and -XX:+UseParNewGC is enabled by default with the concurrent collector.

Serial Collector Versus Parallel Collector

- Both the serial and parallel collectors cause a Stop the World during the GC.
- The serial collector uses only one thread for GC operations.
- The parallel collector can be configured to use one or more threads for GC operations.

ORACLE

Note

A serial collector can work better on applications with small young generation heaps versus a parallel throughput collector. A throughput collector's parallel GC threads may compete for work in small young generation heaps, resulting in thrashing. For small Java heaps, or small young generation Java heaps, try both a serial collector and parallel throughput collector.

Parallel Collector Versus CMS Collector

Parallel Collector Versus CMS Collector

Parallel Collector	CMS Collector
Uses multiple GC threads	Uses only one GC thread
Is a Stop the World collector	Stops the world only during the initial mark and remark phases
	During the concurrent marking and sweeping phases, the CMS thread runs along with the application's threads.
	The CMS collector also does stop the world collection for minor GCs.

ORACLE

Garbage Collectors: Permanent Generation

Garbage Collectors: Permanent Generation



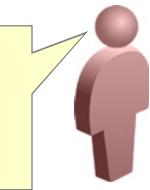
Programmer: Is it necessary that I always tune the permanent generation?

Java Performance Tuning Engineer: Some Java applications require fine-tuning of the permanent generation space.



Programmer: Can you give me an example?

Java Performance Tuning Engineer: Applications that dynamically generate and load many classes need a larger permanent generation space than the provided default max size of 64 MB.



ORACLE

Tuning the Permanent Generation

Some Java applications require fine-tuning of the permanent generation space. Applications that dynamically generate and load many classes such as those commonly seen in web container or application server implementations (especially those utilizing JSPs) need a larger permanent generation space than the provided default maximum size of 64 MB.

Garbage Collectors: Permanent Generation

Garbage Collectors: Permanent Generation



Programmer: How do I tune the permanent generation?

Java Performance Tuning Engineer: Well, you need to follow these steps:

1. Observe the sizing behavior of the permanent generation.
2. Use `-XX:MaxPermSize=<n>` to increase the maximum size.
3. If required, collect the permanent generation by using `-XX:+CMSClassUnloadingEnabled`.



ORACLE

1. Use JConsole, VisualVM or `jstat` to observe the sizing behavior of the permanent generation.
2. Use `-XX:MaxPermSize=<n>` to increase the maximum size. Also consider setting `-XX:PermSize=<n>` to the same value to avoid performance overhead of permanent generation space expansion.
3. The concurrent collector can be specified to collect permanent generation by using `-XX:+CMSClassUnloadingEnabled` and `-XX:+PermGenSweepingEnabled` (not required for HotSpot JVMs on Java 6).

GC Output: Using Serial Collector

GC Output: Using Serial Collector

Minor collection using serial collector:

The screenshot shows a terminal window with the following text output:

```
File Edit New Terminal Tabs Help
bash-3.00$ java -XX:+UseSerialGC -XX:+PrintGCDetails Test
[GC [DefNew: 17213K->126K(19328K), 0.0022428 secs] 17213K->126K(62272K), 0.0023344 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17334K->140K(19328K), 0.0008782 secs] 17334K->140K(62272K), 0.0009261 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17342K->146K(19328K), 0.0006707 secs] 17342K->146K(62272K), 0.0007234 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17362K->150K(19328K), 0.0006329 secs] 17362K->150K(62272K), 0.0006876 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17360K->136K(19328K), 0.0005960 secs] 17360K->136K(62272K), 0.0006452 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17329K->158K(19328K), 0.0005869 secs] 17329K->158K(62272K), 0.0006328 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17374K->161K(19328K), 0.0006554 secs] 17374K->161K(62272K), 0.0007057 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17301K->150K(19328K), 0.0002210 secs] 17420K->269K(62272K), 0.0002601 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 17327K->75K(19328K), 0.0001782 secs] 17446K->194K(62272K), 0.0002174 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 19328K, used 4673K [0xb7e00000, 0xb2f00000, 0xccd50000)
eden space 17216K, 26% used [0xb7e00000, 0xb827d598, 0xb8ed0000)
from space 2112K, 3% used [0xb8ed0000, 0xb8ee2fc8, 0xb90e0000)
to   space 2112K, 0% used [0xb90e0000, 0xb90e0000, 0xb92f0000)
tenured generation total 42944K, used 118K [0xcccd50000, 0xcf740000, 0xf6c00000)
   the space 42944K, 0% used [0xcccd50000, 0xccd6db60, 0xccd6dc00, 0xcf740000)
compacting perm gen total 16384K, used 1808K [0xfc000000, 0xf7c00000, 0xfac00000)
   the space 16384K, 11% used [0xfc000000, 0xf6dc43d8, 0xf6dc4400, 0xf7c00000)
No shared spaces configured.
bash-3.00$
```

ORACLE

Understanding -XX:+PrintGCDetails

In the case of a minor collection:

- DefNew is the young generation space. DefNew indicates the type of young generation collector and that it is also a minor GC. DefNew indicates that the Serial Collector is used for young generation. GC indicates a minor garbage collection.
- The old generation space is not collected.
- The permanent generation space is not collected.

GC Output: Using Serial Collector

GC Output: Using Serial Collector

Full collection using serial collector:

Max, min heap size

```
Terminal
File Edit View Terminal Help
bash-3.00$ java -Xmx16m -Xms16m -XX:+UseSerialGC -XX:+PrintGCDetails Test
[GC [DefNew: 4416K->127K(4928K), 0.0022468 secs] 4416K->127K(15872K), 0.0023375 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4543K->130K(4928K), 0.0008826 secs] 4543K->130K(15872K), 0.0009320 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[GC [DefNew: 4546K->136K(4928K), 0.0011640 secs] 4546K->136K(15872K), 0.0012172 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC [DefNew: 4552K->135K(4928K), 0.0006867 secs] 4552K->135K(15872K), 0.0007310 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4551K->137K(4928K), 0.0005283 secs] 4551K->137K(15872K), 0.0005968 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4609K->271K(4928K), 0.0005746 secs] 12573K->776K(15872K), 0.0006111 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4610K->271K(4928K), 0.0005759 secs] 13115K->319K(15872K), 0.0005602 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Full GC [Tenured: 10674K->919K(10944K), 0.0057642 secs] 15285K->919K(15872K), [Perm : 1797K->1797K(16384K)], 0.0058231 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[GC [DefNew: 4339K->271K(4928K), 0.0007886 secs] 5259K->2275K(15872K), 0.0008242 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4613K->271K(4928K), 0.0004959 secs] 12041K->241K(15872K), 0.0005284 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation total 4928K, used 3345K [0xf5c00000, 0xf6150000, 0xf6150000)
eden space 4416K, 69% used [0xf5c00000, 0xf5f009d8, 0xf6050000)
from space 512K, 52% used [0xf6050000, 0xf6093d58, 0xf60d0000)
to space 512K, 0% used [0xf60d0000, 0xf60d0000, 0xf6150000)
tenured generation total 10944K, used 7970K [0xf6150000, 0xf6c00000, 0xf6c00000)
the space 10944K, 72% used [0xf6150000, 0xf6918a48, 0xf6918c00, 0xf6c00000)
compacting perm gen total 16384K, used 1802K [0xf6c00000, 0xf7c00000, 0xfac00000)
the space 16384K, 11% used [0xf6c00000, 0xf6dc2ba0, 0xf6dc2c00, 0xf7c00000)
No shared spaces configured.
bash-3.00$
```

Full Collection

Understanding -XX:+PrintGCDetails

In the case of a full collection:

- Tenured is the old generation space.
- Perm is the permanent generation space.
- Young generation stats are reported.

GC Output: Using Throughput Collector

GC Output: Using Throughput Collector

Minor collection using parallel collector

The screenshot shows a terminal window with the following text:

```
bash-3.00$ java -Xms16m -Xmx16m -XX:+UseParallelGC -XX:+PrintGCDetails Test
[GC [PSYoungGen: 4608K->152K(5376K)] 4608K->152K(17664K), 0.0009906 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0007968 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0005511 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0009020 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0007383 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4750K->152K(18240K), 0.0012627 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5910K->12K(5952K)] 5910K->12K(18240K), 0.0002611 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5830K->12K(5952K)] 5830K->12K(18240K), 0.0002132 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5854K->123K(5952K)] 5978K->247K(18240K), 0.0003862 secs] [Times: user=0.00 sys=0.00, real=0.06 secs]
[GC [PSYoungGen: 5832K->88K(5952K)] 5956K->212K(18240K), 0.0002132 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5846K->125K(5952K)] 5970K->249K(18240K), 0.0002611 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 5858K->126K(5952K)] 5982K->250K(18240K), 0.0002803 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
bash-3.00$
```

A yellow callout box points to the command line argument `-XX:+UseParallelGC` with the text "Enable parallel collector". A red box highlights the first few lines of the GC output, and a red arrow points from the text "Minor Collection" to the highlighted area.

ORACLE

Understanding -XX:+PrintGCDetails

Here is an analysis of the output:

```
[GC [PSYoungGen: 13737K->1978K(14080K)]
 17407K->6303K(43840K), 0.2144150 secs]
```

Collected 13737 KB minus 1978 KB of space collected in a 14080-KB sized young generation space.

43840 KB overall heap size

Time: It took .2144150 seconds to perform the collection.

PSYoungGen immediately indicates that the garbage collector used is the `-XX:+UseParallelGC` or `-XX:+UseParallelOldGC` collector, that is the throughput collector. Also "GC" indicates that it is a minor GC.

GC Output: Using Throughput Collector

GC Output: Using Throughput Collector

Full collection using throughput collector:

```
Max, min heap size
Terminal
bash-3.00$ java -Xms16m -Xmx16m -XX:+UseParallelGC -XX:+PrintGCDetails Test
[GC [PSYoungGen: 4608K->152K(5376K)] 4608K->152K(17664K), 0.0009906 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0007968 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0005511 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0009020 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4760K->152K(5376K)] 4760K->152K(17664K), 0.0007383 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4750K->152K(5952K)] 4750K->152K(18240K), 0.0012627 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
for [PSYoungGen: 5910K->0K(4480K)] 5910K->0K(4480K), 0.0002647 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Full GC [PSYoungGen: 1643K->0K(4480K)] [PSOldGen: 9883K->3361K(12288K)] [PSPermGen: 1798K->1798K(16384K)], 0.0061904 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [PSYoungGen: 2443K->1643K(4480K)] 7432K->6632K(16768K), 0.01010794 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [PSYoungGen: 4087K->1643K(4480K)] 9076K->8264K(16768K), 0.0191586 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
Heap
PSYoungGen      total 4480K, used 2513K [0xfa600000, 0xfac00000, 0xfac00000)
eden space 2816K, 30% used [0xfa600000, 0xfa6d9900, 0xfabc0000)
from space 1664K, 98% used [0xfaaa60000, 0xfabfae78, 0xfac00000)
to   space 1664K, 0% used [0xfab8c0000, 0xfa8c0000, 0xfaa60000)
PSOldGen       total 12288K, used 6620K [0xf9a00000, 0xfa600000, 0xfa600000)
object space 12288K, 53% used [0xf9a00000, 0xfa077260, 0xfa600000)
PSPermGen      total 16384K, used 1803K [0xf5a00000, 0xf6a00000, 0xf9a00000)
object space 16384K, 11% used [0xf5a00000, 0xf5bc2e00, 0xf6a00000)
bash-3.00$
```

Full Collection

ORACLE

Here is an analysis of the output:

[[Full GC [PSYoungGen: 32K->0K(12800K)]
[PSOldGen: 5180K->5181K(29760K)] 5212K->5181K(42560K)
[PSPermGen: 10974K->10974K(24832K)], 0.0923850 secs]

- PSYoungGen is the young generation space.
- PSOldGen is the old generation space.
- PSPermGen is the permanent generation space.

GC Output: Using Concurrent Collector

GC Output: Using Concurrent Collector

```
File Edit View Terminal Help
bash-3.00$ java -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails
[GC [ParNew: 13060K->243K(14784K), 0.0020213 secs] 13060K->243K(14784K), 0.0021668 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 13375K->242K(14784K), 0.0016899 secs] 13375K->242K(14784K), 0.0017535 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 13208K->408K(14784K), 0.0012977 secs] 13208K->408K(14784K), 0.0013509 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 13445K->621K(14784K), 0.0023626 secs] 13445K->621K(14784K), 0.0024311 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[GC [ParNew: 13691K->93K(14784K), 0.0027265 secs] 13691K->93K(14784K), 0.0027915 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 13649K->427K(14784K), 0.0007557 secs] 13649K->427K(14784K), 0.0008116 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 10795K->7K(14784K), 0.0044174 secs] 64194K->58793K(96704K), 0.0044628 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [ParNew: 10818K->7K(14784K), 0.0005672 secs] 69604K->64192K(96704K), 0.0050229 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [ParNew: 10840K->7K(14784K), 0.0049161 secs] 75025K->69502K(96704K), 0.0049648 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[GC [I CMS-initial-mark: 69594K(81920K) 75012K(96704K), 0.0002133 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-mark: 0.008/0.008 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[GC [ParNew: 10863K->7K(14784K), 0.0052297 secs] 80458K->75023K(96704K), 0.0052836 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[GC [ParNew: 10683K->7K(14784K), 0.0026667 secs] 37312K->31967K(96704K), 0.0027157 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
Heap
  par new generation total 14784K, used 8437K [0xf0c00000, 0xffffc00000, 0xffffc00000)
  eden space 13184K, 63% used [0xf0c00000, 0xf1143d498, 0xf18e0000)
  from space 1600K, 0% used [0xf18e0000, 0xf18e0000, 0xf1a70000)
  to space 1600K, 0% used [0xf1a70000, 0xf1a70000, 0xfc000000)
concurrent mark-sweep generation total 81920K, used 5568K [0xffffc00000, 0xfffff00000, 0xfffff00000)
concurrent-mark-sweep perm gen total 16384K, used 1803K [0xfc000000, 0xff7c0000, 0xfcac0000)
bash-3.00$ bash-3.00$
```

ORACLE

Understanding -XX:+PrintGCDetails

Minor collections when using a concurrent collector follow the serial collector format.

The full GC has the following format:

[Full GC [CMS: 41011K->41011K(81920K), 0.0064932 secs] 41011K->41011K(96704K),
[CMS Perm : 1794K->1794K(16384K)], 0.0065823 secs]

CMS Perm indicates concurrent mark-sweep collection activity in the permanent generation space.

Other Information:

[GC[1 CMS-initial-mark:41011K(81920K)]41011K(96704K),0.0001346 secs]

CMS-initial-mark indicates the start of a concurrent collection cycle.

[CMS-concurrent-mark: 0.007/0.007 secs]

CMS-concurrent-mark indicates the end of the concurrent marking phase.

[CMS-concurrent-preclean: 0.000/0.000 secs]

CMS-concurrent-preclean indicates work performed concurrently in preparation for the remark phase.

[1 CMS-remark: 55994K(81920K)] 59315K(96704K) , 0.0003651 secs]

CMS-remark indicates remarking work

Note: A minor collection occurred concurrently with CMS-remark.

[CMS-concurrent-sweep: 0.000/0.000 secs]

CMS-concurrent-sweep indicates the end of the concurrent sweeping phase.

t[CMS-concurrent-reset: 0.002/0.002 secs]

CMS-concurrent-rese indicates work performed to prepare for the next collection cycle.

Note: When you see the "ParNew" label in a GC log you know that that minor GCs are being executed using the -XX:+UseParNewGC collector, which also happens to be the young generation collector used with the Concurrent collector. When you see the "CMS" label, you know that the Concurrent collector is being used and CMS is the old generation collector.

Concurrent Collector: Losing the Race

Concurrent Collector: Losing the Race

Concurrent mode failure

```
Terminal
File Edit View Terminal Tabs Help
[CMS-concurrent-mark: 0.008/0.013 secs] [Times: user=0
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: us
[GC [ParNew: 9687K->10K(14784K), 0.0053957 secs] 62679
es: user=0.01 sys=0.00, real=0.01 secs]
[GC [ParNew (promotion failed): 9653K->9653K(14784K),
table-preclean: 0.001/0.074 secs] [Times: user=0.04 sy
(concurrent mode failure): 62635K->14592K(81920K), 0.
CMS Perm : 1799K->1799K(16384K)], 0.0183624 secs] [Tim
[GC [ParNew: 9693K->10K(14784K), 0.0047162 secs] 24286
es: user=0.01 sys=0.00, real=0.00 secs]
```

ORACLE

“Losing the race” occurs when the rate at which objects are being promoted to the old generation exceeds the rate at which objects are being collected by the CMS cycle. The corrective action to “losing the race” is tuning the Java heap size for effective object aging, or starting the CMS cycle earlier, or a combination of both.

Garbage Collectors: PrintGCStats

Garbage Collectors: PrintGCStats

PrintGCStats summarizes statistics from the GC activity.

```
PrintGCDetails -v ncpu=<n> [-v interval=<seconds>]  
[-v verbose=1] <gc log file>
```

- ncpu is the number of CPUs on the target machine.
- interval requires the use of the command-line switch
`-XX:+PrintGCTimeStamps` and reports statistics at each “interval.”
- verbose provides more detailed output.

ORACLE

PrintGCStats prints a summary of garbage collection statistics taken from a log file generated by `java -verbose:gc -XX:PrintGCDetails`. These statistics can be very helpful when tuning the GC parameters for a Java application. The tool is applicable only for 32-bit JVMs and can be downloaded from:
<http://java.sun.com/developer/technicalArticles/Programming/turbo/PrintGCStats.zip>.

Garbage Collectors: PrintGCStats

Garbage Collectors: PrintGCStats

Item Name	Description
gen0(s)	Young generation collection time in seconds
cmsIM(s)	CMS initial mark pause in seconds
cmsRM(s)	CMS remark pause in seconds
GC(s)	All stop-the-world GC pauses in seconds
cmsCM(s)	CMS concurrent mark phase in seconds
cmsCS(s)	CMS concurrent sweep phase in seconds
alloc(MB)	Object allocation in young generation in MB
promo(MB)	Object promotion to old generation in MB
elapsed_time(s)	Total wall clock elapsed time for the application run in seconds
tot_cpu_time(s)	Total CPU time = no. of CPUs * elapsed_time

ORACLE

Garbage Collectors: PrintGCStats

Garbage Collectors: PrintGCStats

Item Name	Description
mut_cpu_time(s)	Total time that was available to the application in seconds
gc0_time(s)	Total time used by GC during young generation pauses
alloc/elapsed_time	Allocation rate per unit of elapsed time in MB/seconds
alloc/tot_cpu_time	Allocation rate per unit of total CPU time in MB/seconds
alloc/mut_cpu_time	Allocation rate per unit of total application time in MB/sec
promo/gc0_time	Promotion rate per unit of GC time in MB/seconds
gc_seq_load(%)	Percentage of total time spent in stop-the-world GCs
gc_conc_load(%)	Percentage of total time spent in concurrent GCs
gc_tot_load(%)	Total percentage of GC time (sequential and concurrent)

ORACLE

Garbage Collectors: GCHisto

Garbage Collectors: GCHisto

The GCHisto tool provides a graphical interface for analyzing Java Garbage Collection logs. It includes a summary table and a histogram.

Summary Table:

Name	Num	Num (%)	Total GC (sec)	Overhead (%)	Avg (ms)	Sigma (ms)	Min (ms)	Max (ms)
All	242	100.00%	10.185	100.00%	42.091	38.759	0.241	190.656
Young GC	240	99.17%	10.139	99.54%	1.34%	42.247	30.870	0.241
Full GC	2	0.83%	0.047	0.46%	0.01%	23.386	8.837	17.138

Histogram:

File : gc-logs-4.txt (5 ms buckets)

Count

Buckets (sec)

Young GC Full GC

Tool summarizes GC activity obtained from GC logs

Allows comparison of JVM tuning, such as heap sizes or collector types by comparing GC logs

ORACLE

GCHisto is an open-source project [<http://gchisto.dev.java.net>]. The goal of the GCHisto project is to create a tool that can visualize and provide summary statistics for GC logs so that its user can gain a good understanding of how the GC operated and how it affected the user's applications.

Currently GCHisto is a stand-alone GUI. A VisualVM plug-in is under development. It is not included in the HotSpot JDK. It must be downloaded separately.

Summary

Summary

In this lesson, you should have learned:

- How to tune GC by setting GC generation sizes
- Compare different HotSpot garbage collectors
- Select a HotSpot garbage collector based on application performance requirements
- Use tools to monitor GC and interpret the output

ORACLE

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

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

Language-Level Concerns and Garbage Collection

Chapter 8

8

Language-Level Concerns and Garbage Collection

ORACLE

Objectives

Objectives

After completing this lesson, you should be able to describe:

- The best practices for object allocation
- How garbage collectors can be explicitly invoked
- Reference types in Java
- The use of finalizers in Java

ORACLE

Object Allocation: Best Practices

- Generally, object allocation is inexpensive.
 - About ten native instructions in the fast common case
- Reclamation of objects is inexpensive.
 - Young GCs in generational systems
- There is no performance drawback in allocating small objects for intermediate results.
- Use short-lived immutable objects instead of long-lived mutable objects.
- Use clearer, simpler code with more allocations instead of more obscure code with fewer allocations.

ORACLE

Object Allocation: Working with Large Objects

- Large objects are:
 - Expensive to allocate
 - Expensive to initialize
- Large objects of different sizes can cause fragmentation
 - For non-compacting or partially compacting GCs
- Avoid creating large objects

ORACLE

Garbage Collectors: Explicit GC

Garbage Collectors: Explicit GC

- Do not use `System.gc()` unless there is a specific use case or need to.
- Java HotSpot Virtual Machine
 - Use `-XX:+DisableExplicitGC` to ignore `System.gc()`
 - `System.gc()` does a Stop the World full GC
 - When using JDK 6 and the concurrent collector, you can use `-XX:+ExplicitGCInvokesConcurrent`.

ORACLE

Do not call `java.lang.System.gc()`. The garbage collector generally does a much better job than `System.gc()` in deciding when to perform garbage collection. In fact, performance is likely to decrease if your application repeatedly calls `System.gc()`. If you are having problems with memory usage, pause times for garbage collection, or similar issues, you should configure the memory management system appropriately.

A common use of explicit GC is RMI distributed garbage collection (dgc). -
`XX:+DisableExplicitGC` will disable RMI dgc. Consider tuning RMI dgc if needed rather than disabling explicit GC.

Default RMI distributed GC interval is once per minute (60000 ms). To change this, use `-Dsun.rmi.dgc.client.gcInterval` and `-Dsun.rmi.dgc.server.gcInterval`. The maximum value that it can take is `Long.MAX_VALUE`.

Data Structure Sizing

Data Structure Sizing

- Avoid frequent resizing of array-based data structures.

```
ArrayList<String> list = new ArrayList<String>();  
list.ensureCapacity(1024);
```

- Try to size data structures as realistically as possible.

```
ArrayList<String> list = new ArrayList<String>(1024);
```

ORACLE

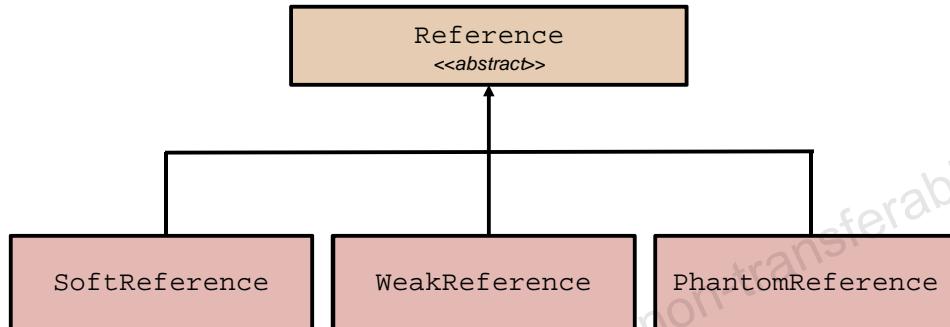
It is important to consider the following issues if large strings are added to the ArrayList:

- Several array-resizing operations will take place.
- They will allocate several large arrays.
- They will cause a large amount of array copying.
- They might cause fragmentation issues on noncompacting GCs.

Garbage Collectors: Reference Objects

Garbage Collectors: Reference Objects

```
package java.lang.ref;
```



ORACLE

Java has three kinds of references, called soft references, weak references, and phantom references, in order of increasing weakness.

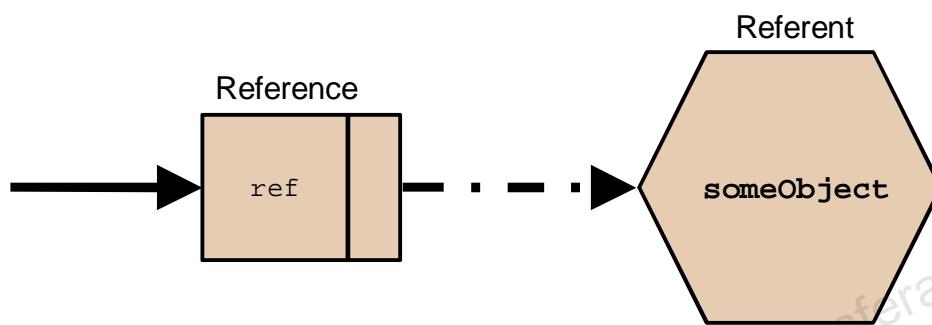
Java has three orders of strength in holding onto objects.

1. **Soft references** can be deleted from a container if the clients are no longer referencing them and memory is tight.
2. **Weak references** are automatically deleted from a container as soon as clients stop referencing them.
3. **Phantom references** point to objects that are already dead and have been finalized.

The JVM holds onto regular objects until they are no longer reachable by either clients or any container. In other words, objects are garbage collected when there are no more live references to them. Dead references do not count.

Reference Objects, Illustration (1/2)

Reference Objects, Illustration (1/2)



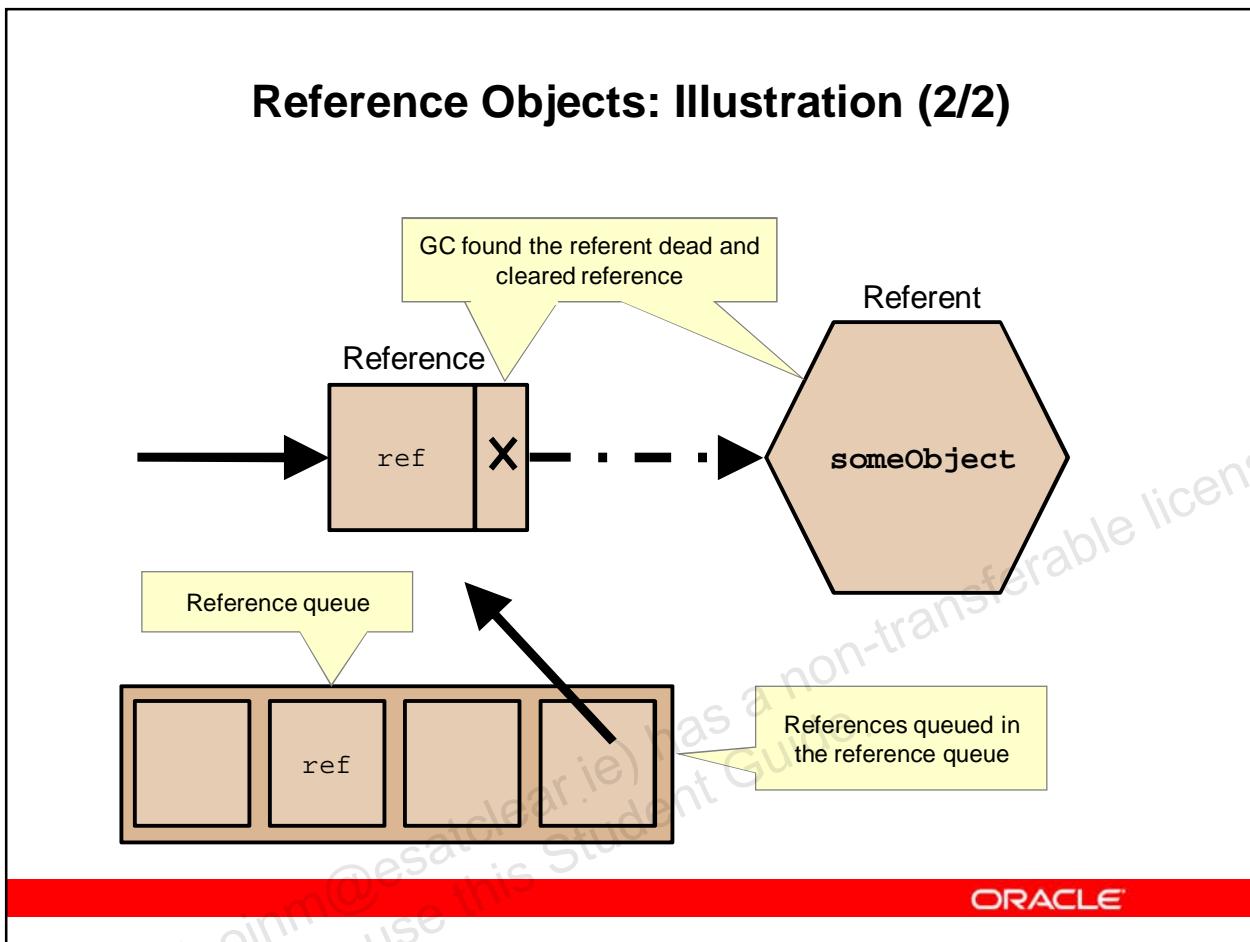
```
ref = new WeakReference(someObject, refQueue);
```

ORACLE

Reference Objects (continued)

The JVM holds onto regular objects until they are no longer reachable by either clients or any container. In other words, objects are garbage collected when there are no more live references to them. Dead references do not count.

Reference Objects: Illustration (2/2)



Once a reference object is discovered by the garbage collector, it is queued for reference processing, which can extend the lifetime of a reference object until the reference processing is completed for that reference object.

Note: If there are many reference objects, the number of reference processing threads can have an impact on the latency of retiring the reference objects.

In addition, many reference objects also give the garbage collector more work to do because unreachable reference objects need to be discovered and queued during garbage collection. Reference object processing can extend the time it takes to perform garbage collections, especially if there are consistently many unreachable reference objects to process.

Reference Objects: Soft Reference

Reference Objects: Soft Reference

- Soft references use “Only reclaim this object if there is memory pressure.”
- `get()` returns the referent, if not reclaimed; else `null`.
- The referent is cleared by the garbage collector.
- Here is an example:

```
...
SoftReference<Object> sr = new SoftReference<Object>(anObject);

Object o = sr.get();
if (o != null) {
    System.out.println(o);
} else {
    // collected or has been reclaimed
}
```

ORACLE

Soft Reference

Soft references are kept alive longer in HotSpot Server JVM.

Use `-XX:SoftRefLRUPolicyMSPerMB=<n>` to control the clearing rate; the default is 1000 ms. This specifies the number of ms a soft reference will be kept alive for each megabyte of free heap space after it is no longer strongly reachable. Keep in mind that soft references are cleared only during garbage collection, which may not occur as frequently as the value set of `SoftRefLRUPolicyMSPerMB`.

Soft references are commonly used for caching.

Reference Objects: Weak Reference

- Weak references use “Tell me if the object has been reclaimed by the GC. Do not retain this object because of this reference.”
- `get()` returns the referent, if not reclaimed; else `null`
- The referent is cleared by the garbage collector.
- Here is an example:

```
Map<Object, Object> weakMap = new WeakHashMap<Object, Object>();
weakMap.put(keyObject, valueObject);

WeakReference weakSelf = new WeakReference<Object>(valueObject);
weakMap.put(keyObject, weakSelf);
...
weakSelf = (WeakReference) weakMap.get(key);

if (weakSelf == null) { // collected or has been reclaimed }
else { valueObject = weakSelf.get(); }
```

ORACLE

Reference Objects: Phantom Reference

Reference Objects: Phantom Reference

- Phantom references uses “Keep some data around after the object becomes unreachable so that I can use that data to clean up after the object.”
- `get()` returns `null` always.
- The referent is *not* cleared by the GC automatically.
- Here is an example:

```
...
ReferenceQueue referenceQueue = new ReferenceQueue();
HashMap map = new HashMap();

Reference reference = new PhantomReference(object, referenceQueue);
map.put(reference, data);

System.out.println(reference.get());
System.out.println(map.get(reference));
System.out.println(reference.isEnqueued());
```

ORACLE

Garbage Collectors: Soft References

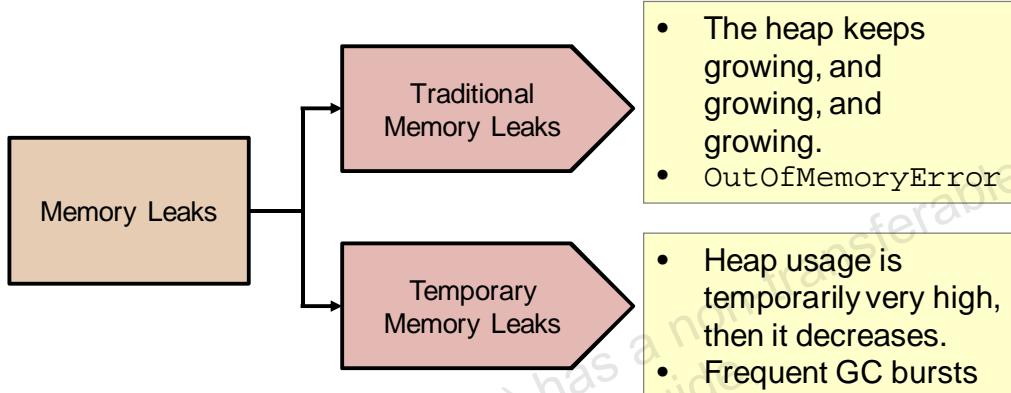
Garbage Collectors: Soft References

- Soft references are kept alive longer in HotSpot Server JVM.
 - `-XX:SoftRefLRUPolicyMSPerMB=<n>` to control the clearing rate; the default is 1000 ms
 - This specifies the number of ms a soft reference will be kept alive for each megabyte of free heap space after it is no longer strongly reachable.
 - Keep in mind that soft references are cleared only during garbage collection, which may not occur as frequently as the value set of `SoftRefLRUPolicyMSPerMB`.

ORACLE

Memory Leaks

- Refers to a condition in which an incremental but steady loss of heap seems to be taking place



ORACLE

A memory leak means the garbage collector is not able to reclaim a certain amount of memory, as the portion of the memory is still being referenced.

You can prevent memory leaks by watching for some common problems. Collection classes, such as hashtables and vectors, are common places to find the cause of a memory leak. This is particularly true if the class has been declared static and exists for the life of the application.

Memory Leaks

Memory Leaks

- Here is an example: OutOfMemoryError

```
...
private static List<Integer> someIntegers = new ArrayList<Integer>();

void someMethod(){
    int count = 0;

    while(true){
        someIntegers.add(new Integer(count));
        count++;
    }
}
```

- When no more memory remains, an OutOfMemoryError alert will be thrown.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at Test.main(Test.java:10)
```

ORACLE

java.lang.OutOfMemoryError is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

What Does OutOfMemoryError Mean?

The meaning is simple: JVM needs more memory and it can no longer acquire it. The reasons behind this memory shortfall can be many. A few important and common reasons are:

- Not enough memory allocated to JVM
- Memory-intensive operations
- Memory leaks

Garbage Collectors: Finalizers

- Objects with finalizers have significant overhead compared to objects without finalizers.
- Finalizers create more pressure on the garbage collector.
- Finalizable objects are retained longer.
- No guarantee:
 - When a finalizer is called
 - Whether they will be called
 - The order in which they will be called

ORACLE

There is a severe performance penalty for using finalizers because of the way the garbage collector works. When you have finalizers, the GC needs to figure out all the objects in eden that need to be finalized, and queue them on a thread that actually executes the finalizers. The GC cannot finish cleaning up the objects efficiently. So, it either has to keep them alive longer than they should be, or has to delay collecting other objects, or both.

Relying on garbage collection to manage resources other than memory is not a good idea. Here are some tips when using finalizers:

- Try to limit the use of the finalizer as a safety net. Use other mechanisms for releasing resources.
- Either include an explicit method for “final” cleanup, or use an alternative “reference-handling” approach by using WeakReferences or SoftReferences.
- If using a finalizer cannot be avoided, try to keep the work being done as small as possible. For instance, do not rely on a finalizer to close file descriptors.

Finalizers Versus Destructors

Finalizers Versus Destructors



Programmer: Finalizers in Java are like C++ destructors, right?



Java Performance Tuning Engineer: No! Not at all.
Finalizers are not like C++ destructors.

The closest concept to C++ destructors is a `finally` clause in Java.

ORACLE

`finalize()` in Java cannot be compared to a C++ destructor. In fact `finalize()` is useless as an object destructor. The `finalize()` method of a class is called only during garbage collection. However, Java does not guarantee when or in what order it will call `finalize()` methods. An object may never be garbage collected, so `finalize()` is not called even when the application terminates.

The unpredictable nature of `finalize()` is not acceptable, except in rare cases. Classes that require cleanup after all access is complete should use explicit destructor methods. Possible actions by destructor methods include the following:

- Removing the object from screen display
- Flushing file buffers
- Closing external file access, and so on

Summary

Summary

In this lesson, you should have learned:

- The best practices for object allocation
- Explicit invocation of the garbage collector and its pitfalls
- To use the reference types in Java
- How and when to use finalizers in Java

ORACLE

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

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

Performance Tuning at the Language Level

Chapter 9

9

Performance Tuning at the Language Level

ORACLE

Objectives

Objectives

After completing this lesson, you should be able to:

- Write string-efficient Java applications
- Use the collection classes efficiently
- Use threads efficiently in a Java application
- Write efficient Java I/O code

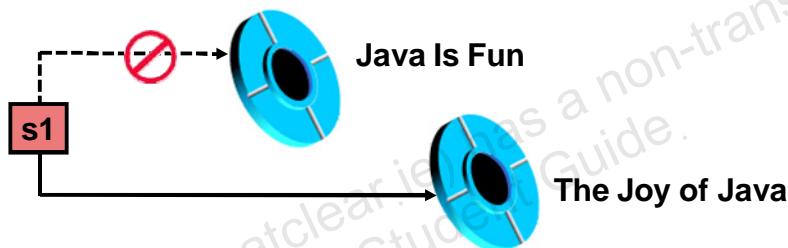
ORACLE

Strings: An Introduction

Strings: An Introduction

- Strings are the only objects with:
 - Their own operators (+ and +=)
 - A literal form
 - Their own externally accessible collection in the VM and class files
- Strings are immutable in Java.

```
String s1 = "Java Is Fun";
s1 = "The Joy of Java";
```



ORACLE

Strings have a special status in the Java programming language. They are the only objects with their own operators, such as + and +=. Characters surrounded by double quotes are also Strings.

A String cannot be altered once created. Although a number of methods, such as .toLowerCase(), .substring(), .trim(), and so on, appear to manipulate Strings, they are unable to do so. Instead, the method returns an altered copy of the String.

Compile Time Versus Runtime Strings

Compile Time Versus Runtime Strings

At compile time, Strings are resolved to eliminate the concatenation operator.

1
...
String s1 = "Java is Fun";
String s2 = "Java" + " is " + "Fun";

boolean result = s1 == s2;

S1 is a compile time string.
S2 is a compile time string.

2
...
String s1 = "Java is Fun";
String s2 = "Java" + " is " +
new StringBuffer("Fun");

boolean result = s1 == s2;

S1 is a compile time string.
S2 is a runtime string.

3
...
public String sayHello(String name){
 return "Hello " + name;
}

An expression involving String concatenation cannot be resolved at compile time.

ORACLE

At compile time, Strings are resolved to eliminate the concatenation operator if possible as in point one in the slide and therefore, the boolean expression evaluated to true.

If you create a String using a StringBuffer, the compiler cannot resolve the String during compile time. Compile time Strings are always more efficient than Strings being created during runtime. In short, when a String can be fully resolved at compile time, the concatenation operator is more efficient than using a StringBuffer. But when the String cannot be resolved at compile time, the concatenation operator is less efficient than using a StringBuffer. Therefore, the boolean expression (2) evaluates to false.

The String generated by the method in (3) cannot be resolved at compile time because the variable name can have any value. The compiler is free to generate code to optimize the String creation, but it does not have to. Consequently, the String-creation line could be compiled as:

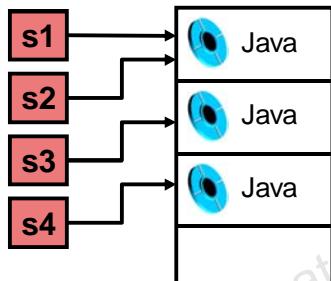
```
return (new StringBuffer( ))  
.append("Hello").append(name).toString( );
```

How JVM Works with Strings

How JVM Works with Strings

JVM maintains an internal list of references for interned Strings to avoid duplicate String objects in heap memory.

```
...
String s1 = "Java";
String s2 = "Java";
String s3 = new String("Java");
String s4 = new String("Java");
...
creation of String Objects
without using the intern() method.
```



```
...
System.out.println(s1 == s2);
String s2 = "Java";
String s3 = new String("Java");
String s4 = new String("Java");
...
...
```

ORACLE

Java Virtual Machine maintains an internal list of references for interned Strings to avoid duplicate String objects in heap memory. Whenever the JVM loads String literal from a class file and executes, it checks whether that String exists in the internal list. If it already exists in the list, it does not create a new String and it uses the reference to the existing String Object.

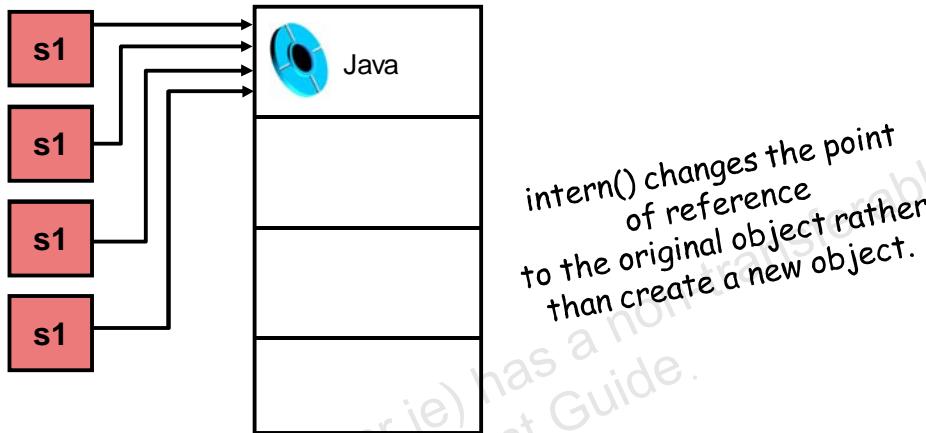
JVM does this type of checking internally for String literal but not for String object, which it creates through the new keyword. You can explicitly force JVM to do this type of checking for String objects that are created through the new keyword by using the `String.intern()` method. This forces JVM to check the internal list and use the existing String object if it is already present.

The diagram in the slide shows the creation of String Objects without using the `intern()` method.

Optimization by Interning Strings

Optimization by Interning Strings

`String.intern()` method avoids duplicating String objects.



ORACLE

In situations where String objects are duplicated unnecessarily, the `String.intern()` method avoids duplicating String objects. The figure shows how the `String.intern()` method works. The `String.intern()` method checks the object existence and if the object exists already, it changes the point of reference to the original object rather than create a new object.

String Versus StringBuffer Versus StringBuilder

String Versus StringBuffer Versus StringBuilder

- A String concatenation creates multiple, intermediate representations and therefore, is not efficient.
- Use the mutable StringBuilder for all cases if no synchronization is needed.
- Use the mutable StringBuffer if synchronization is necessary.

```
...
startTime = System.nanoTime();
for(int i = 0;i < 50000;i++){
    stringObject += i;
}
endTime = System.nanoTime();

System.out.println(endTime - startTime);
...
```

ORACLE

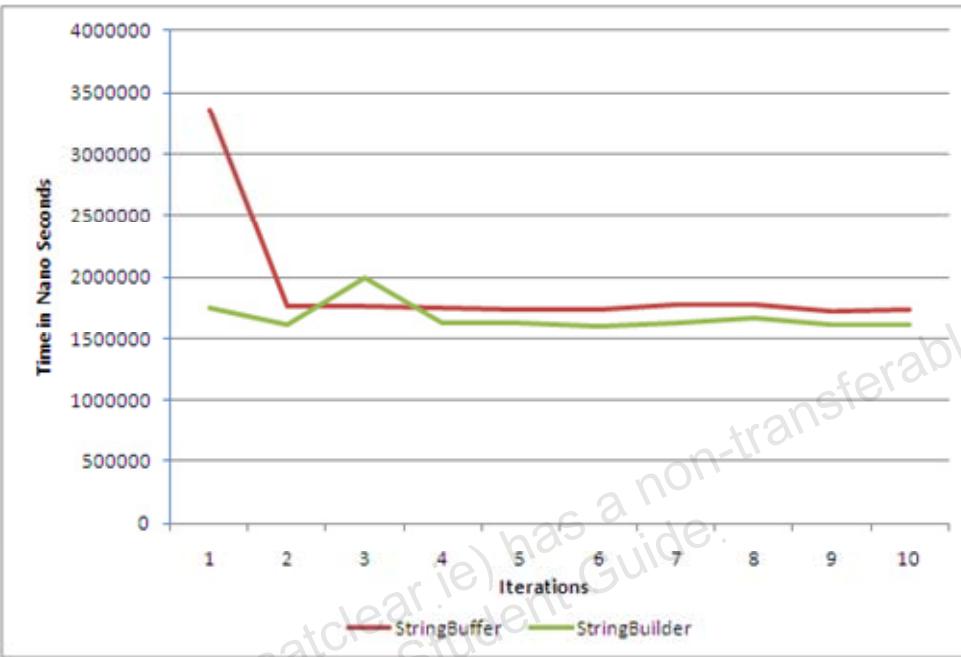
The String class is used to manipulate character strings that cannot be changed. Simply stated, objects of the String type are read only and immutable. The StringBuffer class is used to represent characters that can be modified.

According to javadoc, StringBuilder is designed as a replacement for StringBuffer in single-threaded usage. Their key differences in simple terms are:

- StringBuffer is designed to be thread-safe and all public methods in StringBuffer are synchronized. StringBuilder does not handle the thread-safety issue and none of its methods are synchronized.
- StringBuilder has better performance than StringBuffer under most circumstances.
- Use the new StringBuilder wherever possible.

StringBuffer Versus StringBuilder (in Nanoseconds)

StringBuffer Versus StringBuilder (in Nanoseconds)



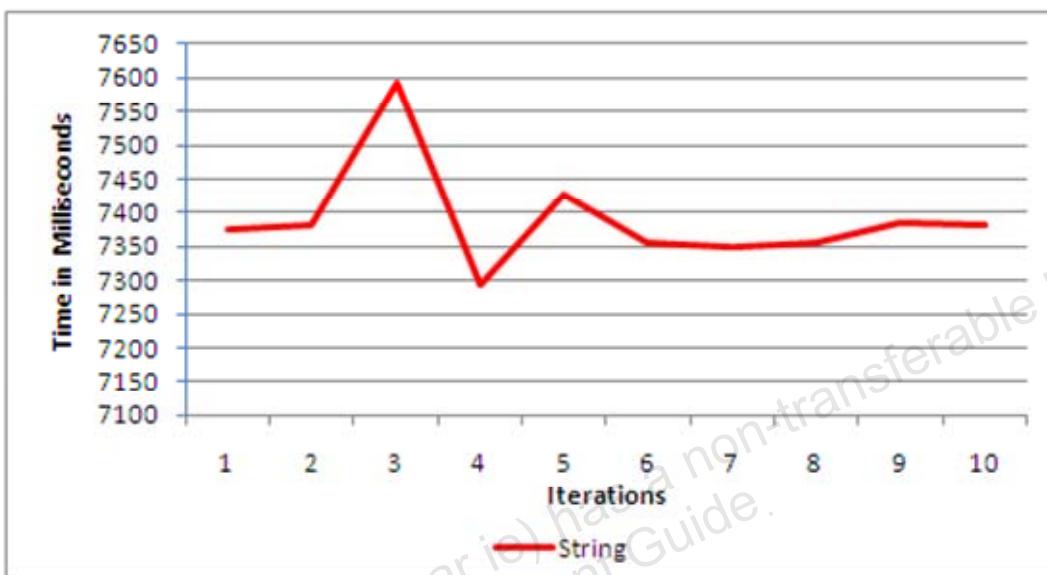
ORACLE

Even though the StringBuilder has been around for a while now, many people still use StringBuffer in single-threaded applications. StringBuilder provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class was designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer, as it will be faster under most implementations.

So StringBuilder is supposed to be faster. But how much faster? To test this, you can run the iterations for about 25,000 times on StringBuffer and StringBuilder and note that StringBuilder is significantly faster than StringBuffer.

The comparison is shown in the above chart.

Execution of "+" Operator in String (in Milliseconds)



ORACLE

There is significant performance difference between String, StringBuffer , and StringBuilder. StringBuffer is faster than String when performing simple concatenations.

StringBuffer is significantly faster than String. Obviously, StringBuffers should be used when possible. If the functionality of the String class is desired, consider using a StringBuffer for major operations and then performing one conversion to String.

Exception Handling and Performance

- Use exceptions only for exceptional situations.
- Never use exceptions for ordinary control flow.

```
try{
    while(true){
        employees[i++].doSomething();
    }
} catch(ArrayIndexOutOfBoundsException e){
    // ...
}
```



- Here is a better replacement for the above code:

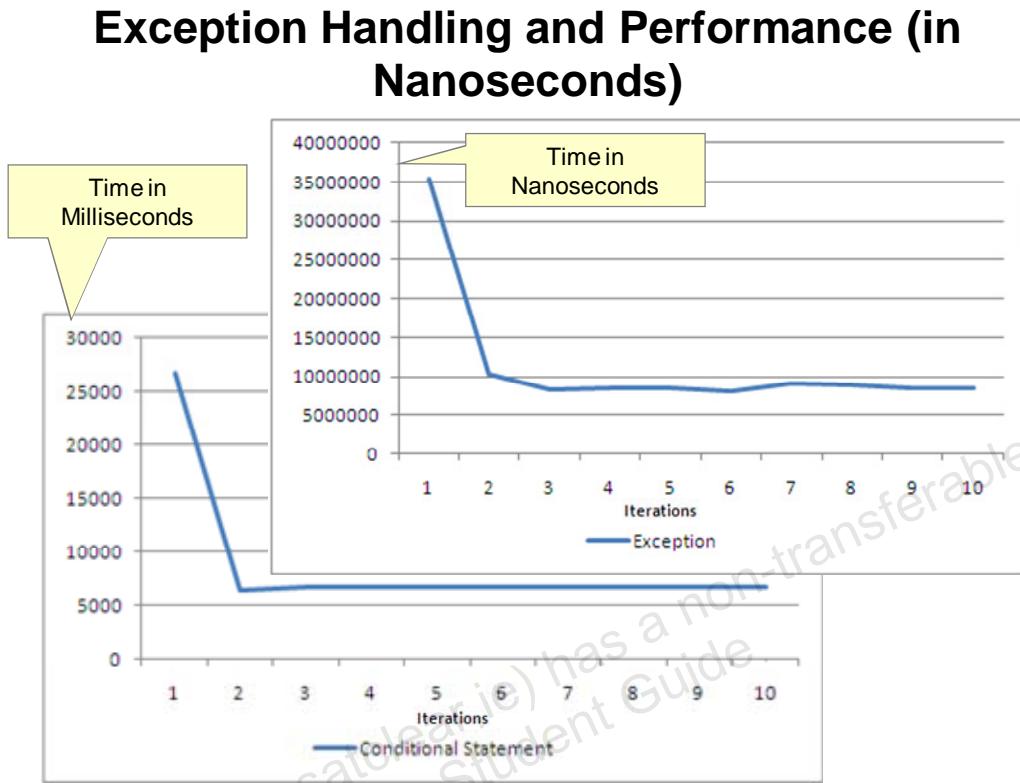
```
for (int i = 0; i < employees.length; i++) {
    employees[i].doSomething();
}
```



ORACLE

The proper use of exceptions can make your programs easier to develop and maintain, freer from bugs, and simpler to use. When exceptions are misused, the opposite situation prevails: programs perform poorly, confuse users, and are harder to maintain. Preventing exception scenarios is one strategy: Use good programming habits such as checking object references for null before accessing a member/method.

Exception Handling and Performance (in Nanoseconds)



ORACLE

Exceptions are, as their name implies, to be used only for exceptional conditions. They should never be used for ordinary control flow. The above graph clearly shows that it is expensive to create, throw, and catch an exception.

Primitives Versus Objects

Primitives Versus Objects

- Use primitives wherever possible.
- Boxing and Unboxing have an associated cost.

```
Integer[] objectArray = new Integer[n];
Long objectSum = 0;

for (int i = 0; i < n; i++) {
    objectArray[i] = i;
    objectSum += objectArray[i];
}
```



- Here is a better replacement of the above code:

```
int[] primitiveArray = new int[n];
long primitiveSum = 0;

for (int i = 0; i < n; i++) {
    primitiveArray[i] = i;
    primitiveSum += primitiveArray[i];
}
```



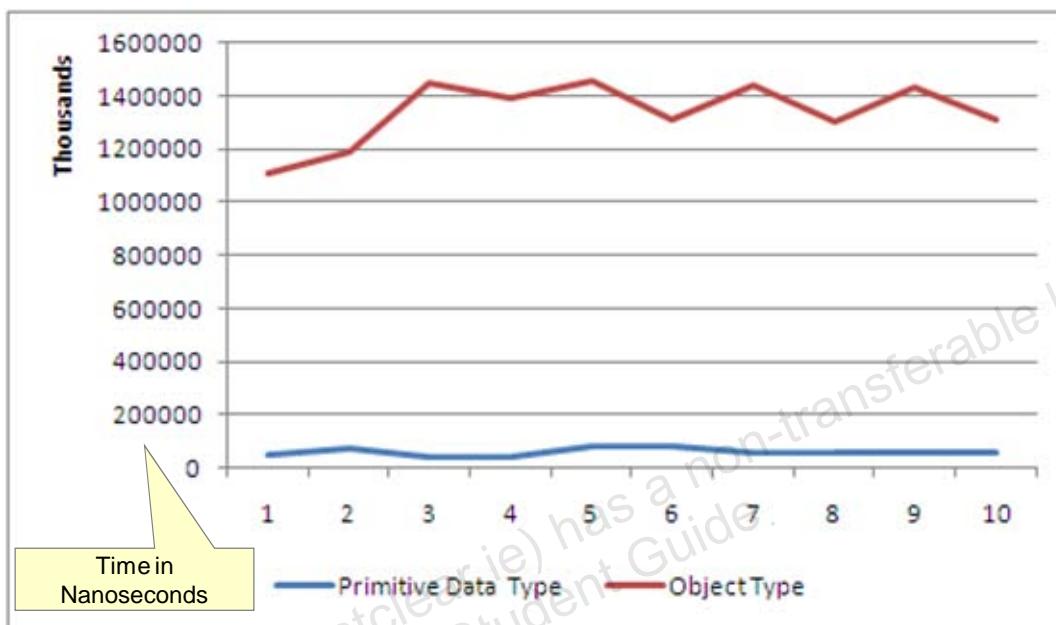
ORACLE

A best practice is to use primitive data types wherever possible, because it would be faster than the object wrapper in all circumstances. Some unavoidable situations where you have to box a primitive data type as an object would be:

- While storing the data in the collections
- If it is a requirement to pass the data as a reference type to a method

Primitives Versus Objects [Benchmark Results]

Primitives Versus Objects [Benchmark Results]



ORACLE

The program was tested on JDK 1.5.1 for running the loops 1,000,000 times. The result very clearly shows that primitive data type is faster than using an object type, and boxing/unboxing has a cost.

Prefer Reusing Objects

- Mutable objects can be changed.
- JDK methods return copies of mutable objects to ensure encapsulation.

```
...
public void paint(Graphics g){
    g.fillRect(0, 0, getSize().getWidth(), getSize().getHeight());
}
```

- Preferably, reuse the existing mutable object to retrieve data.

```
...
public void paint(Graphics g){
    Dimension d = getSize();
    g.fillRect(0, 0, d.getWidth(), d.getHeight());
}
```

ORACLE

Instead of accessing the method that returns a mutable object several times, it is always better that you have the object captured and use the data related to the object. This will avoid unnecessary creation of objects.

Thread Synchronization

Thread Synchronization

- Spraying around the `synchronized` keyword does not ensure that your code is thread-safe.
- Synchronization has a cost:
 - Methods execute more slowly, because acquiring and releasing a monitor lock is expensive.
 - Synchronization may cause a deadlock.
- Use the `synchronized` keyword:
 - Only for a critical section
 - Hold the lock for as short a time as possible.

ORACLE

The main issue with thread synchronization is a liveness problem. A concurrent application's ability to execute in a timely manner is known as its liveness. The most common types of liveness problems are deadlock, starvation, and livelock.

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.
- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result.

Thread Synchronization

Thread Synchronization

- Objective: Hold monitor lock for as short a time as possible
- Account is a shared resource.

```
1 synchronized int getBalance(){  
    Account account = // ...  
    return account.balance;  
}  
  
2 int getBalance(){  
    synchronized(this){  
        Account account = // ...  
        return account.balance;  
    }  
}  
  
3 int getBalance(){  
    Account account = // ...  
    synchronized(account){  
        return account.balance;  
    }  
}
```

The lock is held for a long time and the entire method is synchronized.

Same as above. The current object is locked.

Only the account object is locked for a shorter time - a better solution for the requirement.

ORACLE

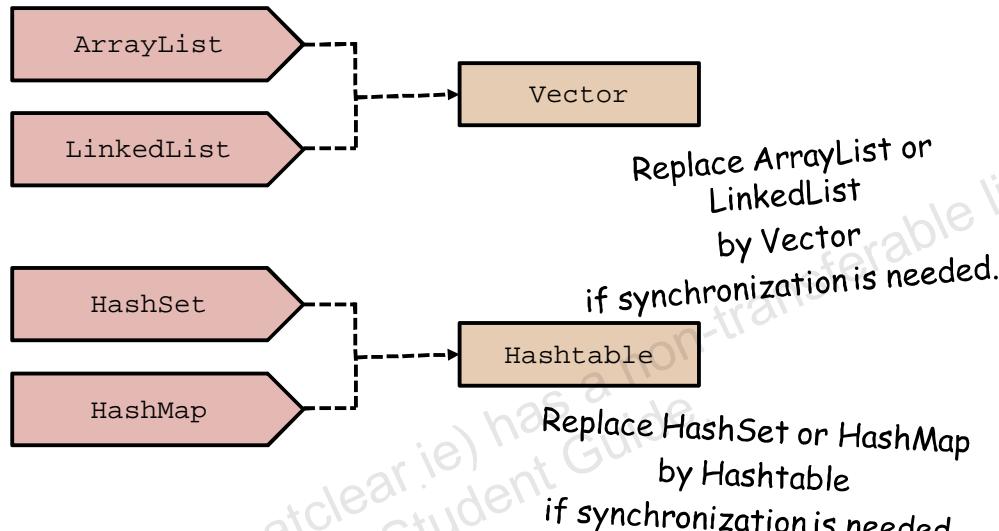
Using synchronization has a high performance cost. Improper synchronization can also cause a deadlock, which can result in complete loss of service because the system usually has to be shut down and restarted. But performance overhead cost is not a sufficient reason to avoid synchronization completely. Failing to make sure that your application is thread-safe in a multithreaded environment can cause data corruption, which can be much worse than losing performance.

The following are some practices that you can consider to minimize the overhead:

- Synchronize critical sections only
- Use private fields
- Use a thread-safe wrapper
- Use immutable objects
- Know which Java objects already have synchronization built-in
- Do not undersynchronize

Collections

All the methods of `Vector` and `Hashtable` are synchronized.



ORACLE

Some Java collection classes like `ArrayList`, `LinkedList`, `HashSet`, and `HashMap` are not synchronized by default. If you are working on a multithreaded environment, the best practice is to use `Vector` as a replacement for `ArrayList` or `LinkedList` and use `Hashtable` as a replacement for classes like `HashSet` or `HashMap`.

Collections

Collections

- Use static factory methods of the Collections class.

```
...  
Collections.synchronizedList(new ArrayList());
```

- It is imperative that the user manually synchronize on the returned list when iterating over it.

```
List list = Collections.synchronizedList(new ArrayList());  
...  
synchronized(list) {  
    Iterator i = list.iterator(); // Must be in synchronized block  
    while (i.hasNext())  
        i.next();  
}
```

- Failure to follow this advice may result in nondeterministic behavior.

ORACLE

The SynchronizedList synchronizes all access methods to the list by providing a synchronized wrapper on top of the List. You must, however, always use the wrapper to access the list elements.

The Iterator is not synchronized, because the moment anything changes in the List you get a ConcurrentModificationException. This means that simply synchronizing individual access methods (next, remove) will not be useful, because if the List is modified externally, the Iterator becomes useless.

When using an Iterator, you actually have to synchronize the entire block that iterates over the List.

The correct usage when using Lists in multiple threads is to get a SynchronizedList and use it everywhere. In the code that uses an Iterator over that List, you have to be sure that you synchronize the block on the SynchronizedList object.

Performance of Java Collections

Performance of Java Collections

- Benchmark for ArrayList

```
List<Integer> list = new ArrayList<Integer>();
final int counter = 100000;
int result;

for (int i = 0; i < counter; i++)
    list.add(i);

for (int i = 0; i < counter; i++)
    result = list.get(i);

for (int i = 0; i < counter; i++)
    list.remove(0);
```

- Tested for:

- add()
- get()
- remove()

ORACLE

Benchmark Results

Benchmark Results

- `ArrayList`
 - The `get()` method is very fast.
 - Removing elements is slow.
- `LinkedList`
 - Removing and editing elements is very fast.
 - Retrieving elements is slow.
- `HashSet/HashMap` is overall very fast.
- `Vector` behaves almost like `ArrayList`.

(m.sec)	<code>ArrayList</code>	<code>LinkedList</code>	<code>HashSet</code>	<code>HashMap</code>
Insertion	66	81	120	188
Retrieval	16	11016	18	31
Removal	3641	3	24	16

ORACLE

The benchmark results give a clear understanding about the usage of the collection. If the application has more random lookups, the best practice is to use `ArrayList` instead of `LinkedList`, whereas if an application has more delete operations, you are strongly urged to use `LinkedList`. It would be faster in every case.

Copying Entire Array: Use System.arraycopy

Copying Entire Array: Use System.arraycopy

- When copying the entire array, use `System.arraycopy` instead of using a loop to copy the elements.

```
...
String[] arraySource = new String[100];
...
String[] arrayDestination = new String[100];

for(int i = 0;i < arraySource.length;i++){
    arrayDestination[i] = arraySource[i];
}
```



- Use `System.arraycopy` for better performance.

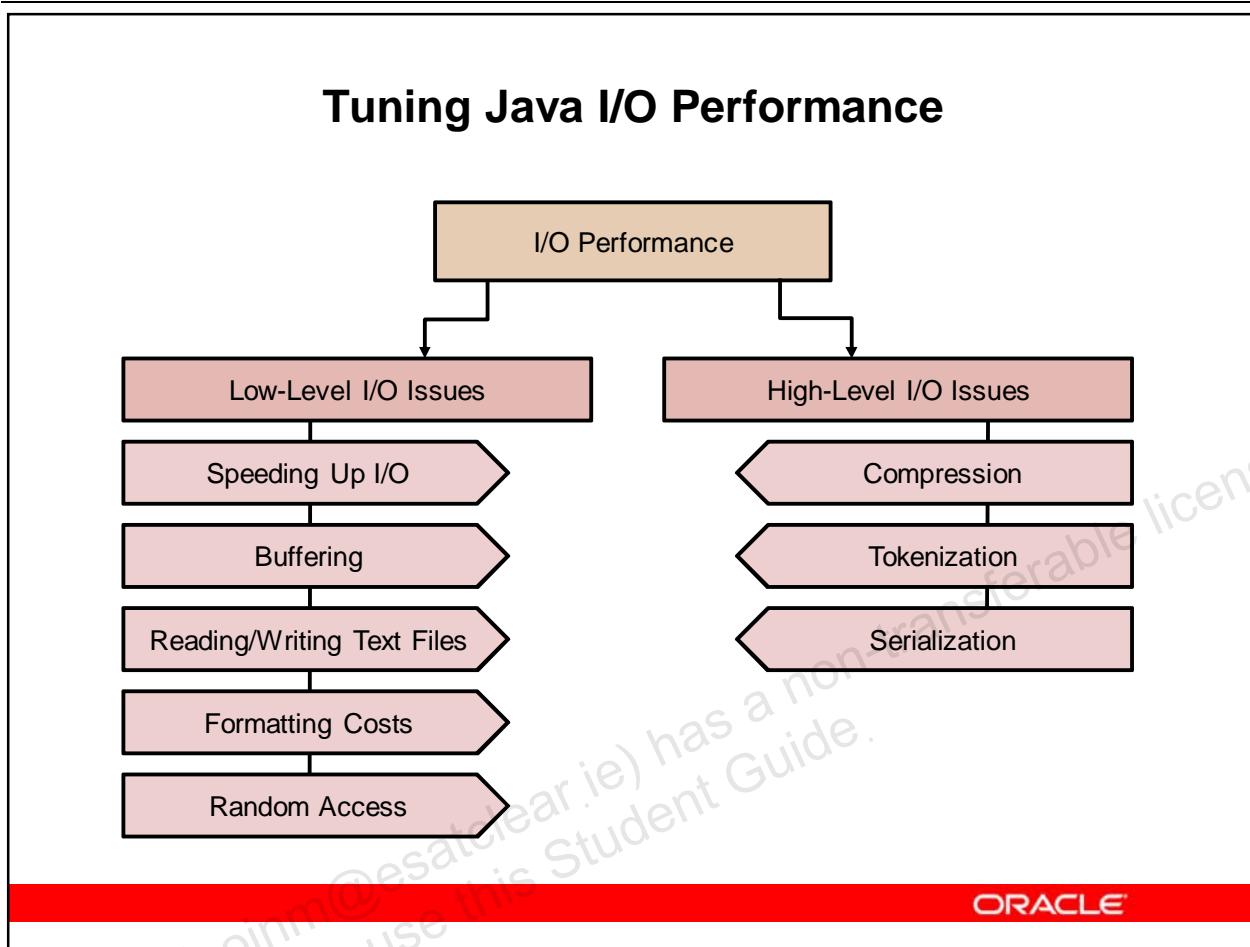
```
...
String[] arraySource = new String[100];
String[] arrayDestination = new String[100];

System.arraycopy(arraySource, 0, arrayDestination, 0, 100);
```



ORACLE

For better performance results, avoid using a linear logic to copy the contents of an array. Instead use `System.arraycopy()` when you have to copy the entire contents from one array to another.



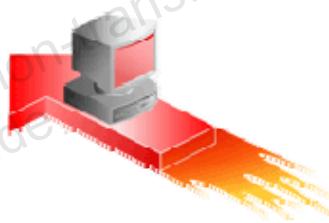
When discussing Java I/O, it is worth noting that the Java programming language assumes two distinct types of disk file organization. One is based on streams of bytes, the other on character sequences. In the Java language a character is represented by using two bytes, not one byte as in other common languages such as C. Because of this, some translation is required to read characters from a file.

Speeding Up I/O

Speeding Up I/O

Here are some basic rules for speeding up I/O:

- Avoid accessing the disk.
- Avoid accessing the underlying operating system.
- Avoid method calls.
- Avoid processing bytes and characters individually.



ORACLE

Speeding Up I/O: Approach 1

Speeding Up I/O: Approach 1

- Use the simple `read` method.

```
...
FileInputStream fis = new FileInputStream("filename");

int cnt = 0;
int b;

while ((b = fis.read()) != -1) {
    if (b == '\n')
        cnt++;
}
```

- `FileInputStream.read` is a native method that returns the next byte of the file.

ORACLE

The first approach simply uses the `read` method on a `FileInputStream`. However, this approach triggers several calls to the underlying runtime system, that is, `FileInputStream.read`, a native method that returns the next byte of the file.

Speeding Up I/O: Approach 2

Speeding Up I/O: Approach 2

- Using a large buffer

```
...
FileInputStream fis = new FileInputStream("filename");
BufferedInputStream bis = new BufferedInputStream(fis);

int cnt = 0;
int b;

while ((b = bis.read()) != -1) {
    if (b == '\n')
        cnt++;
}
```

- `FileInputStream.read` is a native method that returns the next byte of the file.

ORACLE

The second approach avoids the above problem by using a large buffer. `BufferedInputStream.read` takes the next byte from the input buffer, and only rarely accesses the underlying system.

Speeding Up I/O: Approach 3

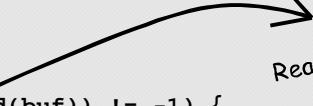
Speeding Up I/O: Approach 3

- Direct Buffering

```
...
FileInputStream fis = new FileInputStream("filename");

byte buf[] = new byte[1024];
int cnt = 0;
int n;

while ((n = fis.read(buf)) != -1) {
    for (int i = 0; i < n; i++) {
        if (buf[i] == '\n')
            cnt++;
    }
}
```



Reads 1024 bytes at a time

- `FileInputStream.read` is a native method that returns the next byte of the file.

ORACLE

The third approach avoids `BufferedInputStream` and performs buffering directly, thereby eliminating the `read` method calls.

Speeding Up I/O: Performance Benchmarking

For a 5-MB input file, the results are as follows:

Approach	Time (in Milliseconds)
Simple Read	33132
Large Buffer	4744
Direct Buffering	2311

ORACLE

The huge speedup does not necessarily prove that you should always emulate the third approach, in which you perform your own buffering. Such an approach may be error-prone, especially in handling end-of-file events, if it is not carefully implemented. It may also be less readable than the alternatives. But it is useful to keep in mind where the time goes, and how it can be reclaimed when necessary.

Buffering

Buffering

- Large chunks of a file are read from disk, and then accessed a byte or character at a time.
- The basic Java classes that support buffering are:
 - `BufferedInputStream` for bytes
 - `BufferedReader` for characters
- Will making the buffer bigger make I/O go faster?



ORACLE

Buffering is a technique where large chunks of a file are read from disk, and then accessed a byte or character at a time. Buffering is a basic and important technique for speeding I/O, and several Java classes support buffering (`BufferedInputStream` for bytes, `BufferedReader` for characters).

An obvious question is: Will making the buffer bigger make I/O go faster? Java buffers typically are by default 1024 or 2048 bytes long. A buffer larger than this may help speed I/O, but often by only a few percent, say 5 to 10%.

Reading/Writing Text Files

Reading/Writing Text Files

- Overhead can be significant when reading data from a file.

1

```
FileInputStream fis = new FileInputStream("filename");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);

while (dis.readLine() != null) { // do something }
```

- DataInputStream.readLine is obsolete.

2

```
FileReader fr = new FileReader("filename");
BufferedReader br = new BufferedReader(fr);

while (br.readLine() != null) { // do something }
```

- readLine does not properly convert bytes to characters
(Java language uses the Unicode character set, not ASCII).

ORACLE

- This program uses the old DataInputStream.readLine method, which is implemented by using read method calls to obtain each character.
- Even though the second program is not faster, there is an important issue to note. The first program evokes a deprecation warning from the Java 2 compiler, because DataInputStream.readLine is obsolete. It does not properly convert bytes to characters, and would not be an appropriate choice for manipulating text files containing anything other than ASCII text byte streams (recall that the Java language uses the Unicode character set, and not ASCII).

Reading/Writing Text Files

Reading/Writing Text Files

- `readLine` is not an appropriate choice for manipulating text files containing anything other than ASCII text.
- The following example writes an output file, but without preserving the Unicode characters that are actually output.

3

```
FileOutputStream fos = new FileOutputStream("filename");
PrintStream ps = new PrintStream(fos);

ps.println("\uffff\u4321\u1143");
```

4

```
FileOutputStream fos = new FileOutputStream("filename");
OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF8");
PrintWriter pw = new PrintWriter(osw);

pw.println("\uffff\u3214\u1243");
```

ORACLE

3. The program in (3) writes an output file, but without preserving the Unicode characters that are actually output. The Reader/Writer I/O classes are character-based, and are designed to resolve this issue. `OutputStreamWriter` is where the encoding of characters to bytes is applied.
4. This program uses the UTF8 encoding, which has the property of encoding ASCII text as itself, and other characters as two or three bytes.

Formatting Costs

Formatting Costs

- Writing data to a file is only part of the cost of output.
- Data formatting has a significant cost.
- Approaches to demonstrate:
 - Simply write out a fixed string to get an idea of the intrinsic I/O cost.
 - Use simple formatting and the “+” character.
 - Use the `MessageFormat` class from the `java.text` package.

ORACLE

Writing data to a file is only part of the cost of output. Another significant cost is data formatting.

Formatting Costs

Formatting Costs

- Output a fixed string to get an idea of the intrinsic I/O cost.

```
...
for (int i = 0; i < 30000; i++) {
    String s = "The cube of 3 is 27\n";
    System.out.print(s);
}
```

- Use simple formatting and the “+” character.

```
...
for (int i = 0; i < 30000; i++) {
    String s = "The cube of " + n + " is " + n * n * n + "\n";
    System.out.print(s);
}
```

ORACLE

- **Approach 1:** The first approach is simply to write out a fixed string to get an idea of the intrinsic I/O cost.
- **Approach 2:** The second approach uses simple formatting with the “+” character.

Formatting Costs: Use MessageFormat Class

Formatting Costs: Use MessageFormat Class

Use the `MessageFormat` class from `java.text` package.

```
...
MessageFormat fmt = new MessageFormat("The cube of {0} is {1}\n");
Object values[] = new Object[2];

values[0] = new Integer(n);
values[1] = new Integer(n * n * n);

for (int i = 0; i < 30000; i++) {
    String s = fmt.format(values);
    System.out.print(s);
}
```

ORACLE

- **Approach 3:** The third approach uses the `MessageFormat` class from the `java.text` package.

Formatting Costs: Benchmarking

Formatting Costs: Benchmarking

The runtime results are as follows:

Approach	Time (in Milliseconds)
Fixed string	14021
Formatting with “+”	19434
Using MessageFormat class	81229

ORACLE

The fact that approach 3 is quite a bit slower than approaches 1 and 2 does not mean that you should not use it. But you need to be aware of the cost in time.

Message formats are quite important in internationalization contexts, and an application concerned about this issue might typically read the format from a resource bundle, and then use it.

Random Access

Random Access

- `RandomAccessFile` is a Java class for doing random access I/O on files.
- The `seek` method that accesses the underlying runtime system is expensive.

```
...
randomAccessFile.seek(blockStart);
n = randomAccessFile.read(inBuffer);
```

- A cheaper alternative is to set up your own buffering on top of a `RandomAccessFile`, and implement a `read` method for bytes directly.

ORACLE

`RandomAccessFile` is a Java class for doing random access I/O (at the byte level) on files. The class provides a `seek` method, similar to that found in C/C++, to move the file pointer to an arbitrary location, from which point bytes can then be read or written.

The `seek` method accesses the underlying runtime system, and as such, tends to be expensive. One cheaper alternative is to set up your own buffering on top of a `RandomAccessFile` and implement a `read` method for bytes directly. The parameter to read is the byte offset ≥ 0 of the desired byte. This technique is helpful if you have locality of access, where nearby bytes in the file are read at about the same time. For example, if you are implementing a binary search scheme on a sorted file, this approach might be useful. It is of less value if you are truly doing random access at arbitrary points in a large file.

Compression

Compression

- Java provides classes for compressing and uncompressing byte streams.
- These are found in the `java.util.zip` package.
- An example is compression.

```
...
fis = new FileInputStream(filein);
fos = new FileOutputStream(fileout);

ZipOutputStream zos = new ZipOutputStream(fos);
ZipEntry ze = new ZipEntry(filein);

zos.putNextEntry(ze);

byte inbuf[] = new byte[4096];
int n;

while ((n = fis.read(inbuf)) != -1) zos.write(inbuf, 0, n);
```

Buffer size is 4096 bytes

ORACLE

Java provides classes for compressing and uncompressing byte streams. These are found in the `java.util.zip` package, and also serve as the basis for `.jar` files.

Whether compression helps or hurts I/O performance depends to a large extent on your local hardware setup; specifically the relative speeds of the processor and disk drives.

Compression using Zip technology implies typically a 50% reduction in data size, but at the cost of some time to compress and decompress.

Tokenization

Tokenization

- Tokenization refers to the process of breaking byte or character sequences into logical chunks.
- Java offers a StreamTokenizer class for tokenization.

```
...
FileReader fr = new FileReader(args[0]);
BufferedReader br = new BufferedReader(fr);
StreamTokenizer st = new StreamTokenizer(br);

st.resetSyntax();
st.wordChars('a', 'z');

int tok;

while ((tok = st.nextToken()) != StreamTokenizer.TT_EOF) {
    if (tok == StreamTokenizer.TT_WORD)
        // st.sval has token
}
```

ORACLE

Tokenization refers to the process of breaking byte or character sequences into logical chunks, for example words. Java offers a StreamTokenizer class to do so.

StreamTokenizer is sort of a hybrid class, in that it will read from character-based streams (like BufferedReader), but at the same time operates in terms of bytes, treating all characters with two-byte values (greater than 0xff) as though they are alphabetic characters. Writing a low-level code would make the code run faster than the one that uses the StreamTokenizer class.

Serialization

Serialization

- Serialization is used to convert arbitrary Java data structures into byte streams, using a standardized format.
- Serialization example:

```
FileOutputStream fos = new FileOutputStream("filename");
BufferedOutputStream bos = new BufferedOutputStream(fos);
ObjectOutputStream oos = new ObjectOutputStream(bos);

oos.writeObject(employeeObject);
```

- Deserialization example:

```
FileInputStream fis = new FileInputStream("filename");
BufferedInputStream bis = new BufferedInputStream(fis);
ObjectInputStream ois = new ObjectInputStream(bis);

Employee e = (Employee) ois.readObject();
```

Note: Buffering is used to speed the I/O operations.

ORACLE

There is probably no faster way than serialization to write out large volumes of data, and then read it back, except in special cases. For example, suppose that you decide to write out a 64-bit long integer as text instead of as a set of 8 bytes. The maximum length of a long integer as text is around 20 characters, or 2.5 times as long as the binary representation. So it seems likely that this format would not be any faster. In some cases, however, such as bitmaps, a special format might be an improvement. However, using your own scheme does work against the standard offered by serialization, so doing so involves some tradeoffs.

Beyond the actual I/O and formatting costs of serialization (using `DataInputStream` and `DataOutputStream`), there are other costs, for example, the need to create new objects when deserializing.

Summary

Summary

In this lesson, you should have learned:

- To write string-efficient Java applications
- To use the collection classes efficiently
- How to use thread synchronization effectively
- To write efficient Java I/O code and about various factors that affect I/O performance

ORACLE

Appendix A: Monitoring Linux Performance

Chapter 10

Monitoring Linux Performance

Unauthorized reproduction or distribution prohibited. Copyright© 2012, Oracle and/or its affiliates.

Monitoring Linux Performance

ORACLE

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

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

Objectives

Objectives

After completing this lesson, you should be able to describe the Linux tools that:

- Monitor CPU usage
- Monitor network I/O
- Monitor disk I/O

ORACLE

Tools For Monitoring Linux

Tools For Monitoring Linux

Tools to monitor CPU utilization

- vmstat (Solaris and Linux)
- mpstat (Solaris and Linux)
- top (Linux; prefer prstat on Solaris)
- pidstat (Linux)
- nicstat (Linux)
- Gnome System Monitor (Linux)

ORACLE

CPU Tools

- Prstat : Similar to top on Linux. On Solaris prstat is less intrusive than top.
- Gnome System Monitor : a graphical representation of CPU utilization on Linux
- Cpubar : A graphical representation of CPU utilization on Solaris
- Iobar : A graphical representation of I/O and CPU utilization

CPU Usage: Linux - vmstat

CPU Usage: Linux - vmstat

CPU Usage: Linux - vmstat															
procs		memory				swap		io		system				cpu	
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
3	0	0	20016	58228	570976	0	0	24	2	15	12	2	1	96	0
1	0	0	20032	58228	570976	0	0	0	0	394	64891	65	35	0	0
2	0	0	20032	58228	570976	0	0	0	0	396	64458	64	36	0	0
2	0	0	20048	58228	570976	0	0	0	0	359	65760	64	36	0	0
1	1	0	13316	58116	572924	0	0	946	0	417	61156	64	36	0	0
2	0	0	13960	55104	561724	0	0	874	17	480	47712	68	32	0	0
3	0	0	12876	54972	559000	0	0	19	0	402	60525	66	34	0	0
2	0	0	12892	54972	559000	0	0	0	0	388	64982	64	36	0	0
2	0	0	12908	54972	559000	0	0	0	0	413	66070	64	36	0	0
2	0	0	12908	54972	559000	0	0	0	0	393	66022	63	37	0	0
2	0	0	12924	54972	559000	0	0	0	0	380	65865	63	37	0	0
2	0	0	12964	54972	559000	0	0	0	0	401	65217	66	34	0	0
2	0	0	12988	54972	559000	0	0	0	0	374	65472	63	37	0	0
2	0	0	13004	54972	559000	0	0	0	0	399	65192	61	39	0	0
2	0	0	13004	54972	559000	0	0	0	0	410	65336	62	38	0	0
2	0	0	12972	54972	559000	0	0	0	0	389	65425	64	36	0	0
2	0	0	12972	54972	559000	0	0	0	0	379	65700	64	36	0	0
2	0	0	12988	54972	559000	0	0	0	0	411	65783	65	35	0	0
2	0	0	12996	54972	559000	0	0	0	0	400	65771	63	37	0	0
4	0	0	11220	54868	556112	0	0	13	0	386	60600	66	34	0	0
2	0	0	14048	54868	555500	0	0	19	2	443	61540	66	34	0	0

Use vmstat to obtain summaries of CPU usage.

ORACLE

Data of interest

- **Us:** user time
- **Sy:** system time
- **Id:** idle time

CPU Usage: Linux mpstat

CPU Usage: Linux mpstat

Time	CPU	%user	%nice	%sys	%iowait	%irq	%soft	%steal	%idle	intr/s
12:58:59 PM	CPU	58.60	0.00	2.99	0.00	0.00	0.25	0.00	38.15	1398.00
12:59:01 PM	all	58.60	0.00	2.99	0.00	0.00	0.25	0.00	38.15	1398.00
12:59:03 PM	all	60.90	0.00	2.26	0.00	0.00	0.00	0.00	36.84	1034.00
12:59:05 PM	all	47.26	0.00	2.99	0.75	0.00	0.00	0.00	49.00	1023.50
12:59:07 PM	all	49.50	0.00	4.00	0.75	0.00	0.00	0.00	45.75	1080.50
12:59:09 PM	all	60.60	0.00	3.99	0.00	0.00	0.00	0.00	35.41	1309.50
12:59:11 PM	all	37.00	0.00	2.25	1.25	0.00	0.00	0.00	59.50	1072.64
12:59:13 PM	all	55.50	0.00	1.50	0.00	0.00	0.00	0.00	43.00	1031.00
12:59:15 PM	all	73.88	0.00	4.73	0.00	0.00	0.00	0.00	21.39	1281.59
12:59:17 PM	all	71.32	0.00	2.24	2.99	0.25	0.00	0.00	23.19	1067.00
12:59:19 PM	all	67.91	0.00	1.49	0.00	0.00	0.00	0.00	30.60	1019.90
12:59:21 PM	all	68.49	0.00	1.74	0.00	0.00	0.00	0.00	29.78	1019.40
12:59:23 PM	all	80.50	0.00	1.00	0.00	0.00	0.25	0.00	18.25	1093.50
12:59:25 PM	all	50.12	0.00	0.00	0.00	0.00	0.00	0.00	49.88	1030.35
12:59:27 PM	all	1.75	0.00	0.00	0.25	0.00	0.00	0.00	98.00	1083.50
12:59:29 PM	all	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	1029.00
12:59:31 PM	all	0.25	0.00	0.00	0.00	0.00	0.00	0.00	99.75	1028.50
12:59:33 PM	all	0.25	0.00	0.00	0.00	0.00	0.00	0.00	99.75	1028.00
12:59:35 PM	all	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	1051.00

ORACLE

Data of interest

- **User:** user time
- **Sys:** system time
- **Idle:** idle time

CPU Usage: pidstat

- %user - % of user level (application) task
- %system - % of system level (kernel) task
- %CPU – total % of CPU time
- cswch/s – total voluntary context switches/second
- nvcswch/s – total involuntary context switches/second

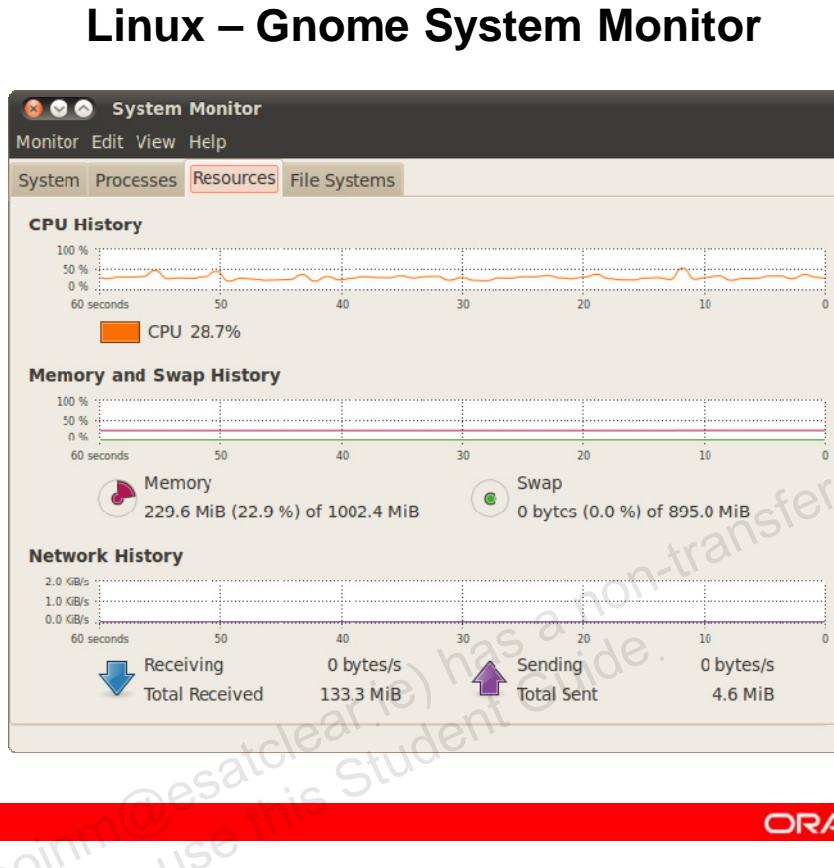
	PID	%user	%system	%CPU	CPU	Command
12:08:00 PM	22326	80.00	12.20	90.20	4	java
12:08:00 PM	22326	cswch/s	nvcswch/s	Command		
		904	18			java
12:08:05 PM	22326	76.00	14.20	90.20	4	java
12:08:05 PM	22326	cswch/s	nvcswch/s	Command		
		706	28			java
12:08:10 PM	22326	84.00	3.10	87.31	4	java
12:08:10 PM	22326	cswch/s	nvcswch/s	Command		
		464	38			java
12:08:15 PM	22326	84.00	3.10	87.31	4	java

To launch:

```
# pidstat -u -w 5
```

ORACLE

Linux – Gnome System Monitor

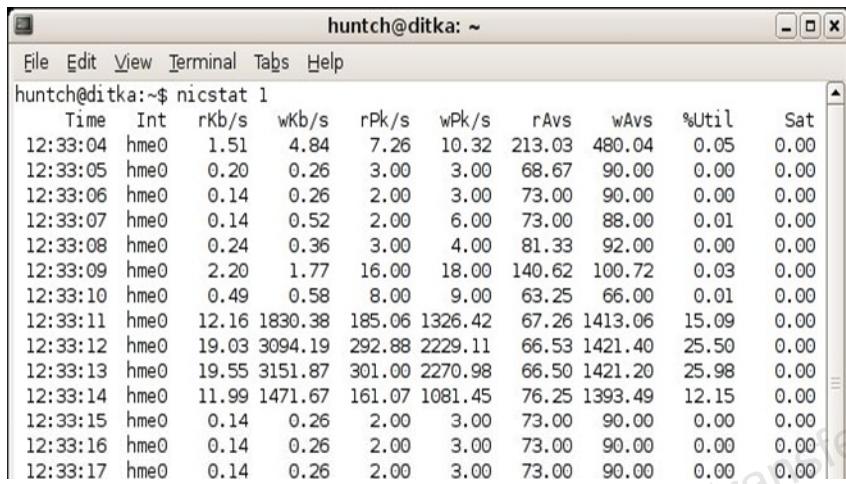


A full featured system monitoring tool for Linux.

ORACLE

Monitoring Network I/O: Using nicstat

Monitoring Network I/O: Using nicstat



A screenshot of a terminal window titled "hutch@ditka: ~". The window displays the output of the "nicstat 1" command. The output shows network statistics for interface "hme0" over 15 intervals of 1 second each. The columns include Time, Int (interval), rKb/s (receive kilobits per second), wKb/s (write kilobits per second), rPk/s (receive packets per second), wPk/s (write packets per second), rAvs (receive average size), wAvs (write average size), %Util (utilization percentage), and Sat (saturation percentage). The "wAvs" column shows values ranging from 8.00 to 185.06, with a significant peak around 1830.38 at interval 12:33:11.

Time	Int	rKb/s	wKb/s	rPk/s	wPk/s	rAvs	wAvs	%Util	Sat
12:33:04	hme0	1.51	4.84	7.26	10.32	213.03	480.04	0.05	0.00
12:33:05	hme0	0.20	0.26	3.00	3.00	68.67	90.00	0.00	0.00
12:33:06	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:07	hme0	0.14	0.52	2.00	6.00	73.00	88.00	0.01	0.00
12:33:08	hme0	0.24	0.36	3.00	4.00	81.33	92.00	0.00	0.00
12:33:09	hme0	2.20	1.77	16.00	18.00	140.62	100.72	0.03	0.00
12:33:10	hme0	0.49	0.58	8.00	9.00	63.25	66.00	0.01	0.00
12:33:11	hme0	12.16	1830.38	185.06	1326.42	67.26	1413.06	15.09	0.00
12:33:12	hme0	19.03	3094.19	292.88	2229.11	66.53	1421.40	25.50	0.00
12:33:13	hme0	19.55	3151.87	301.00	2270.98	66.50	1421.20	25.98	0.00
12:33:14	hme0	11.99	1471.67	161.07	1081.45	76.25	1393.49	12.15	0.00
12:33:15	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:16	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00
12:33:17	hme0	0.14	0.26	2.00	3.00	73.00	90.00	0.00	0.00

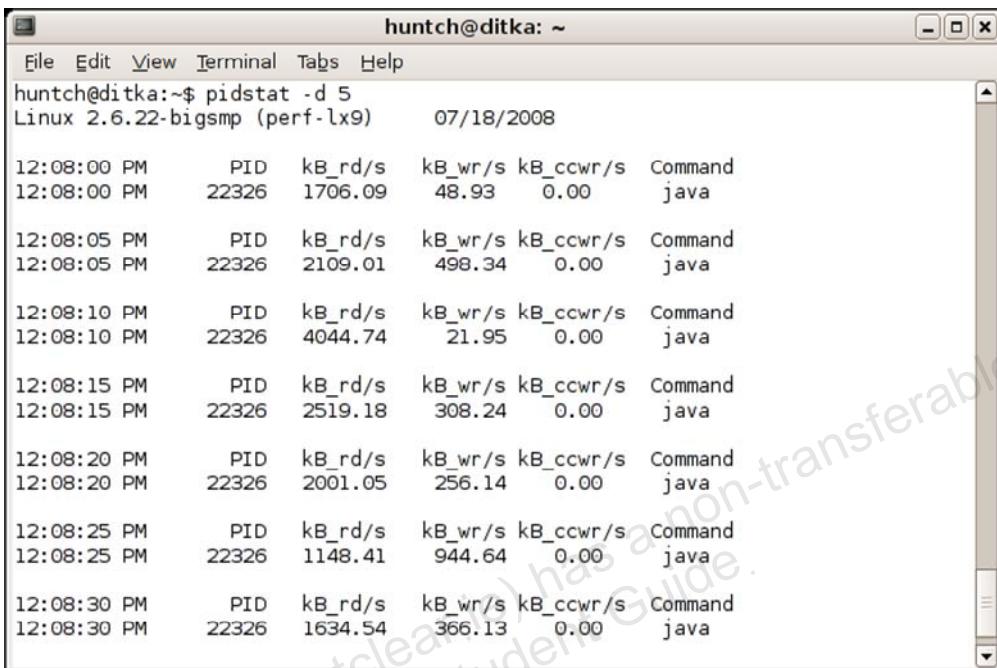
- nicstat displays network statistics
- Notice wAvs, write average size, during four intervals is about 1420 bytes, the Maximum Transmission Unit (MTU) size.

ORACLE

Because 1420 bytes is the maximum MTU for TCP, there was an application using a lot of bandwidth during the middle of this monitoring session.

Monitoring Disk I/O: pidstat Example

Monitoring Disk I/O: pidstat Example



A screenshot of a terminal window titled "huntrch@ditka: ~". The window displays the output of the command "pidstat -d 5". The output shows disk I/O statistics for a process with PID 22326 over a 5-second interval. The statistics include kB_rd/s (read rate), kB_wr/s (write rate), and kB_ccwr/s (cancelled write rate). The process is identified as "Command java". The terminal window has a standard X11 interface with a title bar, menu bar, and scroll bars.

Time	PID	kB_rd/s	kB_wr/s	kB_ccwr/s	Command
12:08:00 PM	22326	1706.09	48.93	0.00	java
12:08:05 PM	22326	2109.01	498.34	0.00	java
12:08:10 PM	22326	4044.74	21.95	0.00	java
12:08:15 PM	22326	2519.18	308.24	0.00	java
12:08:20 PM	22326	2001.05	256.14	0.00	java
12:08:25 PM	22326	1148.41	944.64	0.00	java
12:08:30 PM	22326	1634.54	366.13	0.00	java

ORACLE

This example shows the startup time for an IDE.

- **PID:** The ID of the task being monitored
- **kB_rd/s:** Number of kilobytes the task has caused to be read from disk per second
- **kB_wr/s:** Number of kilobytes the task has caused, or shall cause to be written to disk per second
- **kB_ccwr/s:** Number of kilobytes whose writing to disk has been cancelled by the task. This may occur when the task truncates some dirty pagecache. In this case, some I/O which another task has been accounted for will not be happening.

Kernel Monitoring Using: pidstat

Kernel Monitoring Using: pidstat

- %user - % of user level (application) task
- %system - % of system level (kernel) task
- %CPU – total % of CPU time
- cswch/s – total voluntary context switches/second
- nvcswch/s – total involuntary context switches/second

Time	PID	%user	%system	%CPU	CPU	Command
12:08:00 PM	22326	80.00	12.20	90.20	4	java
12:08:00 PM	22326	cswch/s	nvcswch/s	Command		
		904	18			java
12:08:05 PM	22326	76.00	14.20	90.20	4	java
12:08:05 PM	22326	cswch/s	nvcswch/s	Command		
		706	28			java
12:08:10 PM	22326	84.00	3.10	87.31	4	java
12:08:10 PM	22326	cswch/s	nvcswch/s	Command		
		464	38			java
12:08:15 PM	22326	84.00	3.10	87.31	4	java

ORACLE

Summary

Summary

In this module we focused on monitoring various aspects of the operating system including how to:

- Monitor CPU usage
- Monitor network I/O
- Monitor disk I/O

ORACLE

Appendix B

Chapter 11

Appendix B

Appendix B

ORACLE

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

Objectives

Objectives

After completing this lesson, you should be able to:

- Describe categories of Java HotSpot VM Options
- Use -XX Options

ORACLE

Java HotSpot VM Options

- In this appendix, you find information about typical command-line options and environment variables that can affect the performance characteristics of the Java HotSpot Virtual Machine.
- All information in this appendix pertains to both the Java HotSpot Client VM and the Java HotSpot Server VM.

ORACLE

Categories of Java HotSpot VM Options

Categories of Java HotSpot VM Options

This appendix deals exclusively with nonstandard options recognized by the Java HotSpot VM:

- Options that begin with -X are nonstandard
- Options that are specified with -XX are not stable and are not recommended for casual use.

ORACLE®

Useful -XX Options

Useful -XX Options

Default values are listed for Java SE 6 for Solaris Sparc with the –server option. Some options may vary per architecture/OS/JVM version.

- Boolean options are turned on with -XX:+<option> and turned off with -XX:-<option>.
- Numeric options are set with -XX:<option>=<number>. Numbers can include “m” or “M” for megabytes, “k” or “K” for kilobytes, and “g” or “G” for gigabytes (for example, 32 k is the same as 32768).
- String options are set with -XX:<option>=<string>, and are usually used to specify a file, a path, or a list of commands.

ORACLE

Useful -XX Options

Useful -XX Options

The options below are loosely grouped into three categories:

- Behavioral options change the basic behavior of the VM.
- Performance tuning options are knobs that can be used to tune VM performance.
- Debugging options generally enable tracing, printing, or output of VM information.

ORACLE

Behavioral Options

`-XX:-AllowUserSignalHandlers`

Do not complain if the application installs signal handlers. (relevant to Solaris and Linux only)

-XX:AltStackSize=16384
Alternate signal stack size (in Kbytes) (relevant to Solaris only, removed from 5.0.)

-XX:-DisableExplicitGC
Disable calls to System.gc(), JVM still performs garbage collection when necessary.

-XX:+FailOverToOldVerifier
Fail over to old verifier when the new type checker fails. (introduced in 6)

-XX:+HandlePromotionFailure
The youngest generation collection does not require a guarantee of full promotion of all live objects. (introduced in 1.4.2 update 11) [5.0 and earlier: false]

-XX:+MaxFDLimit
Bump the number of file descriptors to max. (relevant to Solaris only)

-XX:PreBlockSpin=10
Spin count variable for use with -XX:+UseSpinning. Controls the maximum spin iterations allowed before entering operating system thread synchronization code. (introduced in 1.4.2)

-XX:-RelaxAccessControlCheck
Relax the access control checks in the verifier. (introduced in 6)

-XX:+ScavengeBeforeFullGC
Do young generation GC prior to a full GC. (introduced in 1.4.1)

-XX:+UseAltSigs
Use alternate signals instead of SIGUSR1 and SIGUSR2 for VM internal signals. (introduced in 1.3.1 update 9, 1.4.1; relevant to Solaris only)

-XX:+UseBoundThreads
Bind user-level threads to kernel threads. (relevant to Solaris only)

-XX:-UseConcMarkSweepGC
Use concurrent mark-sweep collection for the old generation. (introduced in 1.4.1)

-XX:+UseGCOverheadLimit
Use a policy that limits the proportion of the VM's time that is spent in GC before an OutOfMemory error is thrown. (introduced in 6)

-XX:+UseLWPSynchronization
Use LWP-based instead of thread based synchronization. (introduced in 1.4.0; relevant to Solaris only)

-XX:-UseParallelGC
Use parallel garbage collection for scavenges. (Introduced in 1.4.1)

-XX:-UseParallelOldGC
Use parallel garbage collection for the full collections. Enabling this option automatically sets -XX:+UseParallelGC. (Introduced in 5.0 update 6.)

-XX:-UseSerialGC
Use serial garbage collection. (Introduced in 5.0.)

-XX:-UseSpinning
Enable naive spinning on Java monitor before entering operating system thread synchronization code. (Relevant to 1.4.2 and 5.0 only.) [1.4.2, multi-processor Windows platforms: true]

-XX:+UseTLAB
Use thread-local object allocation (Introduced in 1.4.0, known as UseTLE prior to that.) [1.4.2 and earlier, x86 or with -client: false]

-XX:+UseSplitVerifier
 Use the new type checker with StackMapTable attributes. (Introduced in 5.0.) [5.0: false]

-XX:+UseThreadPriorities
 Use native thread priorities.

-XX:+UseVMIInterruptibleIO
 Thread interrupt before or with EINTR for I/O operations results in OS_INTRPT. (Introduced in 6. Relevant to Solaris only.)

Performance Options

-XX:+AggressiveOpts
 Turn on point performance compiler optimizations that are expected to be default in upcoming releases. (Introduced in 5.0 update 6.)

-XX:CompileThreshold=10000
 Number of method invocations/branches before compiling [-client: 1,500]

-XX:LargePageSizeInBytes=4m
 Sets the large page size used for the Java heap. (Introduced in 1.4.0 update 1.) [amd64: 2m.]

-XX:MaxHeapFreeRatio=70
 Maximum percentage of heap free after GC to avoid shrinking.

-XX:MaxNewSize=size
 Maximum size of new generation (in bytes). Since 1.4, MaxNewSize is computed as a function of NewRatio. [1.3.1 Sparc: 32m; 1.3.1 x86: 2.5m.]

-XX:MaxPermSize=64m
 Size of the Permanent Generation. [5.0 and newer: 64 bit VMs are scaled 30% larger; 1.4 amd64: 96m; 1.3.1 -client: 32m.]

-XX:MinHeapFreeRatio=40
 Minimum percentage of heap free after GC to avoid expansion.

-XX:NewRatio=2
 Ratio of new/old generation sizes. [Sparc -client: 8; x86 -server: 8; x86 -client: 12.] -client: 4 (1.3) 8 (1.3.1+), x86: 12]

-XX:NewSize=2.125m
 Default size of new generation (in bytes) [5.0 and newer: 64 bit VMs are scaled 30% larger; x86: 1m; x86, 5.0 and older: 640k]

-XX:ReservedCodeCacheSize=32m
 Reserved code cache size (in bytes) - maximum code cache size. [Solaris 64-bit, amd64, and -server x86: 48m; in 1.5.0_06 and earlier, Solaris 64-bit and amd64: 1024m.]

-XX:SurvivorRatio=8
 Ratio of eden/survivor space size [Solaris amd64: 6; Sparc in 1.3.1: 25; other Solaris platforms in 5.0 and earlier: 32]

-XX:TargetSurvivorRatio=50
 Desired percentage of survivor space used after scavenge.

-XX:ThreadStackSize=512
 Thread Stack Size (in Kbytes). (0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.]

-XX:+UseBiasedLocking
 Enable biased locking. For more details, see this tuning example. (Introduced in 5.0 update 6.) [5.0: false]

-XX:+UseFastAccessorMethods
Use optimized versions of Get<Primitive>Field.

-XX:-UseISM
Use Intimate Shared Memory. [Not accepted for non-Solaris platforms.] For details, see Intimate Shared Memory.

-XX:+UseLargePages
Use large page memory. (Introduced in 5.0 update 5.) For details, see Java Support for Large Memory Pages.

-XX:+UseMPSS
Use Multiple Page Size Support w/4mb pages for the heap. Do not use with ISM as this replaces the need for ISM. (Introduced in 1.4.0 update 1, Relevant to Solaris 9 and newer.)
[1.4.1 and earlier: false]

-XX:+UseStringCache
Enables caching of commonly allocated strings.

-XX:AllocatePrefetchLines=1
Number of cache lines to load after the last object allocation using prefetch instructions generated in JIT compiled code. Default values are 1 if the last allocated object was an instance and 3 if it was an array.

-XX:AllocatePrefetchStyle=1
Generated code style for prefetch instructions.
0 - no prefetch instructions are generate*d*,
1 - execute prefetch instructions after each allocation,
2 - use TLAB allocation watermark pointer to gate when prefetch instructions are executed.

-XX:+UseCompressedStrings
Use a byte[] for Strings which can be represented as pure ASCII. (Introduced in Java 6 Update 21 Performance Release)

-XX:+OptimizeStringConcat
Optimize String concatenation operations where possible. (Introduced in Java 6 Update 20)

Debugging Options

-XX:-CITime
Prints time spent in JIT Compiler. (Introduced in 1.4.0.)

-XX:ErrorFile=./hs_err_pid<pid>.log
If an error occurs, save the error data to this file. (Introduced in 6.)

-XX:-ExtendedDTraceProbes
Enable performance-impacting dtrace probes. (Introduced in 6. Relevant to Solaris only.)

-XX:HeapDumpPath=./java_pid<pid>.hprof
Path to directory or filename for heap dump. Manageable. (Introduced in 1.4.2 update 12, 5.0 update 7.)

-XX:-HeapDumpOnOutOfMemoryError
Dump heap to file when java.lang.OutOfMemoryError is thrown. Manageable. (Introduced in 1.4.2 update 12, 5.0 update 7.)

-XX:OnError=<cmd args>;<cmd args>
Run user-defined commands on fatal error. (Introduced in 1.4.2 update 9.)

-XX:OnOutOfMemoryError="`<cmd args>; <cmd args>`"
Run user-defined commands when an OutOfMemoryError is first thrown. (Introduced in 1.4.2 update 12, 6)

-XX:-PrintClassHistogram
Print a histogram of class instances on Ctrl-Break. Manageable. (Introduced in 1.4.2.) The jmap -histo command provides equivalent functionality.

-XX:-PrintConcurrentLocks
Print java.util.concurrent locks in Ctrl-Break thread dump. Manageable. (Introduced in 6.) The jstack -l command provides equivalent functionality.

-XX:-PrintCommandLineFlags
Print flags that appeared on the command line. (Introduced in 5.0.)

-XX:-PrintCompilation
Print message when a method is compiled.

-XX:-PrintGC
Print messages at garbage collection. Manageable.

-XX:-PrintGCDetails
Print more details at garbage collection. Manageable. (Introduced in 1.4.0.)

-XX:-PrintGCTimeStamps
Print timestamps at garbage collection. Manageable (Introduced in 1.4.0.)

-XX:-PrintTenuringDistribution
Print tenuring age information.

-XX:-TraceClassLoader
Trace loading of classes.

-XX:-TraceClassLoaderPreorder
Trace all classes loaded in order referenced (not loaded). (Introduced in 1.4.2.)

-XX:-TraceClassResolution
Trace constant pool resolutions. (Introduced in 1.4.2.)

-XX:-TraceClassUnloading
Trace unloading of classes.

-XX:-TraceLoaderConstraints
Trace recording of loader constraints. (Introduced in 6.)

-XX:+PerfSaveDataToFile
Saves jvmstat binary data on exit.

-XX:ParallelGCThreads=`n`
Sets the number of garbage collection threads in the young and old parallel garbage collectors. The default value varies with the platform on which the JVM is running.

-XX:+UseCompressedOops
Enables the use of compressed pointers (object references represented as 32 bit offsets instead of 64-bit pointers) for optimized 64-bit performance with Java heap sizes less than 32gb.

-XX:+AlwaysPreTouch
Pre-touch the Java heap during JVM initialization. Every page of the heap is thus demand-zeroed during initialization rather than incrementally during application execution.

-XX:AllocatePrefetchDistance=`n`
Sets the prefetch distance for object allocation. Memory about to be written with the value of new objects is prefetched into cache at this distance (in bytes) beyond the address of the last

allocated object. Each Java thread has its own allocation point. The default value varies with the platform on which the JVM is running.

-XX:InlineSmallCode=

Inline a previously compiled method only if its generated native code size is less than this. The default value varies with the platform on which the JVM is running.

-XX:MaxInlineSize=35

Maximum bytecode size of a method to be inlined.

-XX:FreqInlineSize=

Maximum bytecode size of a frequently executed method to be inlined. The default value varies with the platform on which the JVM is running.

-XX:LoopUnrollLimit=

Unroll loop bodies with server compiler intermediate representation node count less than this value. The limit used by the server compiler is a function of this value, not the actual value. The default value varies with the platform on which the JVM is running.

-XX:InitialTenuringThreshold=7

Sets the initial tenuring threshold for use in adaptive GC sizing in the parallel young collector. The tenuring threshold is the number of times an object survives a young collection before being promoted to the old, or tenured, generation.

-XX:MaxTenuringThreshold=

Sets the maximum tenuring threshold for use in adaptive GC sizing. The current largest value is 15. The default value is 15 for the parallel collector and 4 for CMS.