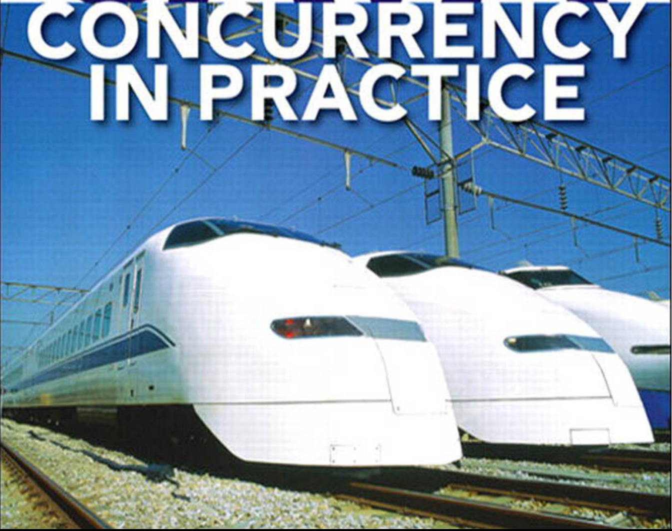
BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA

JAVA CONCURRENCY IN PRACTICE



FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Advance praise for *Java Concurrency in Practice*

I was fortunate indeed to have worked with a fantastic team on the design and implementation of the concurrency features added to the Java platform in Java 5.0 and Java 6. Now this same team provides the best explanation yet of these new features, and of concurrency in general. Concurrency is no longer a subject for advanced users only. Every Java developer should read this book.

—Martin Buchholz
JDK Concurrency Czar, Sun Microsystems

For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. Writing code that effectively exploits multiple processors can be very challenging. *Java Concurrency in Practice* provides you with the concepts and techniques needed to write safe and scalable Java programs for today's—and tomorrow's—systems.

—Doron Rajwan
Research Scientist, Intel Corp

This is the book you need if you're writing—or designing, or debugging, or maintaining, or contemplating—multithreaded Java programs. If you've ever had to synchronize a method and you weren't sure why, you owe it to yourself and your users to read this book, cover to cover.

—Ted Neward
Author of Effective Enterprise Java

Brian addresses the fundamental issues and complexities of concurrency with uncommon clarity. This book is a must-read for anyone who uses threads and cares about performance.

—Kirk Pepperdine
CTO, JavaPerformanceTuning.com

This book covers a very deep and subtle topic in a very clear and concise way, making it the perfect Java Concurrency reference manual. Each page is filled with the problems (and solutions!) that programmers struggle with every day. Effectively exploiting concurrency is becoming more and more important now that Moore's Law is delivering more cores but not faster cores, and this book will show you how to do it.

—Dr. Cliff Click
Senior Software Engineer, Azul Systems

I have a strong interest in concurrency, and have probably written more thread deadlocks and made more synchronization mistakes than most programmers. Brian's book is the most readable on the topic of threading and concurrency in Java, and deals with this difficult subject with a wonderful hands-on approach. This is a book I am recommending to all my readers of The Java Specialists' Newsletter, because it is interesting, useful, and relevant to the problems facing Java developers today.

—Dr. Heinz Kabutz
The Java Specialists' Newsletter

I've focused a career on simplifying simple problems, but this book ambitiously and effectively works to simplify a complex but critical subject: concurrency. *Java Concurrency in Practice* is revolutionary in its approach, smooth and easy in style, and timely in its delivery—it's destined to be a very important book.

—Bruce Tate
Author of Beyond Java

Java Concurrency in Practice is an invaluable compilation of threading know-how for Java developers. I found reading this book intellectually exciting, in part because it is an excellent introduction to Java's concurrency API, but mostly because it captures in a thorough and accessible way expert knowledge on threading not easily found elsewhere.

—Bill Venners
Author of Inside the Java Virtual Machine

Java Concurrency in Practice

This page intentionally left blank

Java Concurrency in Practice

Brian Goetz
with
Tim Peierls
Joshua Bloch
Joseph Bowbeer
David Holmes
and Doug Lea

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to http://www.awprofessional.com/safarienabled
- Complete the brief registration form
- Enter the coupon code

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Library of Congress Cataloging-in-Publication Data

Goetz, Brian.
Java Concurrency in Practice / Brian Goetz, with Tim Peierls. . . [et al.]
p. cm.
Includes bibliographical references and index.
ISBN 0-321-34960-1 (pbk. : alk. paper)
1. Java (Computer program language) 2. Parallel programming (Computer science) 3. Threads (Computer programs) I. Title.

QA76.73.J38G588 2006
005.13'3--dc22 2006012205

Copyright © 2006 Pearson Education, Inc.

ISBN 0-321-34960-1
Text printed in the United States on recycled paper at Courier Stoughton in Stoughton, Massachusetts.
13th Printing

To Jessica

This page intentionally left blank

Contents

Listings xii

Preface xvii

1 Introduction 1

1.1 A (very) brief history of concurrency . . . 1

1.2 Benefits of threads . . . 3

1.3 Risks of threads . . . 5

1.4 Threads are everywhere . . . 9

I Fundamentals 13

2 Thread Safety 15

2.1 What is thread safety? . . . 17

2.2 Atomicity . . . 19

2.3 Locking . . . 23

2.4 Guarding state with locks . . . 27

2.5 Liveness and performance . . . 29

3 Sharing Objects 33

3.1 Visibility . . . 33

3.2 Publication and escape . . . 39

3.3 Thread confinement . . . 42

3.4 Immutability . . . 46

3.5 Safe publication . . . 49

4 Composing Objects 55

4.1 Designing a thread-safe class . . . 55

4.2 Instance confinement . . . 58

4.3 Delegating thread safety . . . 62

4.4 Adding functionality to existing thread-safe classes . . . 71

4.5 Documenting synchronization policies . . . 74

5 **Building Blocks** **79**
5.1 Synchronized collections 79
5.2 Concurrent collections 84
5.3 Blocking queues and the producer-consumer pattern 87
5.4 Blocking and interruptible methods 92
5.5 Synchronizers 94
5.6 Building an efficient, scalable result cache 101

II **Structuring Concurrent Applications** **111**

6 **Task Execution** **113**
6.1 Executing tasks in threads 113
6.2 The `Executor` framework 117
6.3 Finding exploitable parallelism 123

7 **Cancellation and Shutdown** **135**
7.1 Task cancellation 135
7.2 Stopping a thread-based service 150
7.3 Handling abnormal thread termination 161
7.4 JVM shutdown 164

8 **Applying Thread Pools** **167**
8.1 Implicit couplings between tasks and execution policies 167
8.2 Sizing thread pools 170
8.3 Configuring `ThreadPoolExecutor` 171
8.4 Extending `ThreadPoolExecutor` 179
8.5 Parallelizing recursive algorithms 181

9 **GUI Applications** **189**
9.1 Why are GUIs single-threaded? 189
9.2 Short-running GUI tasks 192
9.3 Long-running GUI tasks 195
9.4 Shared data models 198
9.5 Other forms of single-threaded subsystems 202

III **Liveness, Performance, and Testing** **203**

10 **Avoiding Liveness Hazards** **205**
10.1 Deadlock 205
10.2 Avoiding and diagnosing deadlocks 215
10.3 Other liveness hazards 218

11 **Performance and Scalability** **221**
11.1 Thinking about performance 221
11.2 Amdahl's law 225
11.3 Costs introduced by threads 229
11.4 Reducing lock contention 232

11.5 Example: Comparing Map performance . . . 242
11.6 Reducing context switch overhead . . . 243

12 Testing Concurrent Programs **247**
12.1 Testing for correctness . . . 248
12.2 Testing for performance . . . 260
12.3 Avoiding performance testing pitfalls . . . 266
12.4 Complementary testing approaches . . . 270

IV Advanced Topics **275**

13 Explicit Locks **277**
13.1 Lock and ReentrantLock . . . 277
13.2 Performance considerations . . . 282
13.3 Fairness . . . 283
13.4 Choosing between synchronized and ReentrantLock . . . 285
13.5 Read-write locks . . . 286

14 Building Custom Synchronizers **291**
14.1 Managing state dependence . . . 291
14.2 Using condition queues . . . 298
14.3 Explicit condition objects . . . 306
14.4 Anatomy of a synchronizer . . . 308
14.5 AbstractQueuedSynchronizer . . . 311
14.6 AQS in java.util.concurrent synchronizer classes . . . 314

15 Atomic Variables and Nonblocking Synchronization **319**
15.1 Disadvantages of locking . . . 319
15.2 Hardware support for concurrency . . . 321
15.3 Atomic variable classes . . . 324
15.4 Nonblocking algorithms . . . 329

16 The Java Memory Model **337**
16.1 What is a memory model, and why would I want one? . . . 337
16.2 Publication . . . 344
16.3 Initialization safety . . . 349

A Annotations for Concurrency **353**
A.1 Class annotations . . . 353
A.2 Field and method annotations . . . 353

Bibliography **355**

Index **359**

Listings

1 Bad way to sort a list. *Don't do this.* xix
2 Less than optimal way to sort a list. xx
1.1 Non-thread-safe sequence generator. 6
1.2 Thread-safe sequence generator. 7
2.1 A stateless servlet. 18
2.2 Servlet that counts requests without the necessary synchronization. *Don't do this.* 19
2.3 Race condition in lazy initialization. *Don't do this.* 21
2.4 Servlet that counts requests using `AtomicLong`. 23
2.5 Servlet that attempts to cache its last result without adequate atomicity. *Don't do this.* 24
2.6 Servlet that caches last result, but with unnacceptably poor concurrency. *Don't do this.* 26
2.7 Code that would deadlock if intrinsic locks were not reentrant. 27
2.8 Servlet that caches its last request and result. 31
3.1 Sharing variables without synchronization. *Don't do this.* 34
3.2 Non-thread-safe mutable integer holder. 36
3.3 Thread-safe mutable integer holder. 36
3.4 Counting sheep. 39
3.5 Publishing an object. 40
3.6 Allowing internal mutable state to escape. *Don't do this.* 40
3.7 Implicitly allowing the `this` reference to escape. *Don't do this.* 41
3.8 Using a factory method to prevent the `this` reference from escaping during construction. 42
3.9 Thread confinement of local primitive and reference variables. 44
3.10 Using `ThreadLocal` to ensure thread confinement. 45
3.11 Immutable class built out of mutable underlying objects. 47
3.12 Immutable holder for caching a number and its factors. 49
3.13 Caching the last result using a volatile reference to an immutable holder object. 50
3.14 Publishing an object without adequate synchronization. *Don't do this.* 50
3.15 Class at risk of failure if not properly published. 51
4.1 Simple thread-safe counter using the Java monitor pattern. 56
4.2 Using confinement to ensure thread safety. 59
4.3 Guarding state with a private lock. 61

4.4 Monitor-based vehicle tracker implementation. 63
4.5 Mutable point class similar to `java.awt.Point`. 64
4.6 Immutable `Point` class used by `DelegatingVehicleTracker`. 64
4.7 Delegating thread safety to a `ConcurrentHashMap`. 65
4.8 Returning a static copy of the location set instead of a "live" one. . 66
4.9 Delegating thread safety to multiple underlying state variables. . . 66
4.10 Number range class that does not sufficiently protect its invariants. *Don't do this.* 67
4.11 Thread-safe mutable point class. 69
4.12 Vehicle tracker that safely publishes underlying state. 70
4.13 Extending `Vector` to have a put-if-absent method. 72
4.14 Non-thread-safe attempt to implement put-if-absent. *Don't do this.* . 72
4.15 Implementing put-if-absent with client-side locking. 73
4.16 Implementing put-if-absent using composition. 74
5.1 Compound actions on a `Vector` that may produce confusing results. 80
5.2 Compound actions on `Vector` using client-side locking. 81
5.3 Iteration that may throw `ArrayIndexOutOfBoundsException`. 81
5.4 Iteration with client-side locking. 82
5.5 Iterating a `List` with an `Iterator`. 82
5.6 Iteration hidden within string concatenation. *Don't do this.* 84
5.7 `ConcurrentMap` interface. 87
5.8 Producer and consumer tasks in a desktop search application. . . . 91
5.9 Starting the desktop search. 92
5.10 Restoring the interrupted status so as not to swallow the interrupt. 94
5.11 Using `CountDownLatch` for starting and stopping threads in timing tests. 96
5.12 Using `FutureTask` to preload data that is needed later. 97
5.13 Coercing an unchecked `Throwable` to a `RuntimeException`. 98
5.14 Using `Semaphore` to bound a collection. 100
5.15 Coordinating computation in a cellular automaton with `CyclicBarrier`. 102
5.16 Initial cache attempt using `HashMap` and synchronization. 103
5.17 Replacing `HashMap` with `ConcurrentHashMap`. 105
5.18 Memoizing wrapper using `FutureTask`. 106
5.19 Final implementation of `Memoizer`. 108
5.20 Factorizing servlet that caches results using `Memoizer`. 109
6.1 Sequential web server. 114
6.2 Web server that starts a new thread for each request. 115
6.3 `Executor` interface. 117
6.4 Web server using a thread pool. 118
6.5 `Executor` that starts a new thread for each task. 118
6.6 `Executor` that executes tasks synchronously in the calling thread. . 119
6.7 Lifecycle methods in `ExecutorService`. 121
6.8 Web server with shutdown support. 122
6.9 Class illustrating confusing `Timer` behavior. 124
6.10 Rendering page elements sequentially. 125
6.11 `Callable` and `Future` interfaces. 126

6.12 Default implementation of newTaskFor in ThreadPoolExecutor. . . 126
6.13 Waiting for image download with Future. 128
6.14 QueueingFuture class used by ExecutorCompletionService. 129
6.15 Using CompletionService to render page elements as they become available. 130
6.16 Fetching an advertisement with a time budget. 132
6.17 Requesting travel quotes under a time budget. 134
7.1 Using a volatile field to hold cancellation state. 137
7.2 Generating a second's worth of prime numbers. 137
7.3 Unreliable cancellation that can leave producers stuck in a blocking operation. *Don't do this.* 139
7.4 Interruption methods in Thread. 139
7.5 Using interruption for cancellation. 141
7.6 Propagating InterruptedException to callers. 143
7.7 Noncancelable task that restores interruption before exit. 144
7.8 Scheduling an interrupt on a borrowed thread. *Don't do this.* 145
7.9 Interrupting a task in a dedicated thread. 146
7.10 Cancelling a task using Future. 147
7.11 Encapsulating nonstandard cancellation in a Thread by overriding interrupt. 149
7.12 Encapsulating nonstandard cancellation in a task with newTaskFor. 151
7.13 Producer-consumer logging service with no shutdown support. . . 152
7.14 Unreliable way to add shutdown support to the logging service. . . 153
7.15 Adding reliable cancellation to LogWriter. 154
7.16 Logging service that uses an ExecutorService. 155
7.17 Shutdown with poison pill. 156
7.18 Producer thread for IndexingService. 157
7.19 Consumer thread for IndexingService. 157
7.20 Using a private Executor whose lifetime is bounded by a method call. 158
7.21 ExecutorService that keeps track of cancelled tasks after shutdown.159
7.22 Using TrackingExecutorService to save unfinished tasks for later execution. 160
7.23 Typical thread-pool worker thread structure. 162
7.24 UncaughtExceptionHandler interface. 163
7.25 UncaughtExceptionHandler that logs the exception. 163
7.26 Registering a shutdown hook to stop the logging service. 165
8.1 Task that deadlocks in a single-threaded Executor. *Don't do this.* . . 169
8.2 General constructor for ThreadPoolExecutor. 172
8.3 Creating a fixed-sized thread pool with a bounded queue and the caller-runs saturation policy. 175
8.4 Using a Semaphore to throttle task submission. 176
8.5 ThreadFactory interface. 176
8.6 Custom thread factory. 177
8.7 Custom thread base class. 178
8.8 Modifying an Executor created with the standard factories. 179
8.9 Thread pool extended with logging and timing. 180

8.10 Transforming sequential execution into parallel execution. 181
8.11 Transforming sequential tail-recursion into parallelized recursion. . 182
8.12 Waiting for results to be calculated in parallel. 182
8.13 Abstraction for puzzles like the "sliding blocks puzzle". 183
8.14 Link node for the puzzle solver framework. 184
8.15 Sequential puzzle solver. 185
8.16 Concurrent version of puzzle solver. 186
8.17 Result-bearing latch used by `ConcurrentPuzzleSolver`. 187
8.18 Solver that recognizes when no solution exists. 188
9.1 Implementing `SwingUtilities` using an `Executor`. 193
9.2 `Executor` built atop `SwingUtilities`. 194
9.3 Simple event listener. 194
9.4 Binding a long-running task to a visual component. 196
9.5 Long-running task with user feedback. 196
9.6 Cancelling a long-running task. 197
9.7 Background task class supporting cancellation, completion notification, and progress notification. 199
9.8 Initiating a long-running, cancellable task with `BackgroundTask`. . 200
10.1 Simple lock-ordering deadlock. *Don't do this.* 207
10.2 Dynamic lock-ordering deadlock. *Don't do this.* 208
10.3 Inducing a lock ordering to avoid deadlock. 209
10.4 Driver loop that induces deadlock under typical conditions. 210
10.5 Lock-ordering deadlock between cooperating objects. *Don't do this.* 212
10.6 Using open calls to avoiding deadlock between cooperating objects. 214
10.7 Portion of thread dump after deadlock. 217
11.1 Serialized access to a task queue. 227
11.2 Synchronization that has no effect. *Don't do this.* 230
11.3 Candidate for lock elision. 231
11.4 Holding a lock longer than necessary. 233
11.5 Reducing lock duration. 234
11.6 Candidate for lock splitting. 236
11.7 `ServerStatus` refactored to use split locks. 236
11.8 Hash-based map using lock striping. 238
12.1 Bounded buffer using `Semaphore`. 249
12.2 Basic unit tests for `BoundedBuffer`. 250
12.3 Testing blocking and responsiveness to interruption. 252
12.4 Medium-quality random number generator suitable for testing. . . 253
12.5 Producer-consumer test program for `BoundedBuffer`. 255
12.6 Producer and consumer classes used in `PutTakeTest`. 256
12.7 Testing for resource leaks. 258
12.8 Thread factory for testing `ThreadPoolExecutor`. 258
12.9 Test method to verify thread pool expansion. 259
12.10 Using `Thread.yield` to generate more interleavings. 260
12.11 Barrier-based timer. 261
12.12 Testing with a barrier-based timer. 262
12.13 Driver program for `TimedPutTakeTest`. 262
13.1 `Lock` interface. 277

13.2 Guarding object state using ReentrantLock. 278
13.3 Avoiding lock-ordering deadlock using tryLock. 280
13.4 Locking with a time budget. 281
13.5 Interruptible lock acquisition. 281
13.6 ReadWriteLock interface. 286
13.7 Wrapping a Map with a read-write lock. 288
14.1 Structure of blocking state-dependent actions. 292
14.2 Base class for bounded buffer implementations. 293
14.3 Bounded buffer that balks when preconditions are not met. 294
14.4 Client logic for calling GrumpyBoundedBuffer. 294
14.5 Bounded buffer using crude blocking. 296
14.6 Bounded buffer using condition queues. 298
14.7 Canonical form for state-dependent methods. 301
14.8 Using conditional notification in BoundedBuffer.put. 304
14.9 Recloseable gate using wait and notifyAll. 305
14.10 Condition interface. 307
14.11 Bounded buffer using explicit condition variables. 309
14.12 Counting semaphore implemented using Lock. 310
14.13 Canonical forms for acquisition and release in AQS. 312
14.14 Binary latch using AbstractQueuedSynchronizer. 313
14.15 tryAcquire implementation from nonfair ReentrantLock. 315
14.16 tryAcquireShared and tryReleaseShared from Semaphore. 316
15.1 Simulated CAS operation. 322
15.2 Nonblocking counter using CAS. 323
15.3 Preserving multivariable invariants using CAS. 326
15.4 Random number generator using ReentrantLock. 327
15.5 Random number generator using AtomicInteger. 327
15.6 Nonblocking stack using Treiber's algorithm (Treiber, 1986). 331
15.7 Insertion in the Michael-Scott nonblocking queue algorithm (Michael and Scott, 1996). 334
15.8 Using atomic field updaters in ConcurrentLinkedQueue. 335
16.1 Insufficiently synchronized program that can have surprising results. *Don't do this.* 340
16.2 Inner class of FutureTask illustrating synchronization piggybacking. 343
16.3 Unsafe lazy initialization. *Don't do this.* 345
16.4 Thread-safe lazy initialization. 347
16.5 Eager initialization. 347
16.6 Lazy initialization holder class idiom. 348
16.7 Double-checked-locking antipattern. *Don't do this.* 349
16.8 Initialization safety for immutable objects. 350

Preface

At this writing, multicore processors are just now becoming inexpensive enough for midrange desktop systems. Not coincidentally, many development teams are noticing more and more threading-related bug reports in their projects. In a recent post on the NetBeans developer site, one of the core maintainers observed that a single class had been patched over 14 times to fix threading-related problems. Dion Almaer, former editor of TheServerSide, recently blogged (after a painful debugging session that ultimately revealed a threading bug) that most Java programs are so rife with concurrency bugs that they work only "by accident".

Indeed, developing, testing and debugging multithreaded programs can be extremely difficult because concurrency bugs do not manifest themselves predictably. And when they do surface, it is often at the worst possible time—in production, under heavy load.

One of the challenges of developing concurrent programs in Java is the mismatch between the concurrency features offered by the platform and how developers need to think about concurrency in their programs. The language provides low-level *mechanisms* such as synchronization and condition waits, but these mechanisms must be used consistently to implement application-level protocols or *policies*. Without such policies, it is all too easy to create programs that compile and appear to work but are nevertheless broken. Many otherwise excellent books on concurrency fall short of their goal by focusing excessively on low-level mechanisms and APIs rather than design-level policies and patterns.

Java 5.0 is a huge step forward for the development of concurrent applications in Java, providing new higher-level components and additional low-level mechanisms that make it easier for novices and experts alike to build concurrent applications. The authors are the primary members of the JCP Expert Group that created these facilities; in addition to describing their behavior and features, we present the underlying design patterns and anticipated usage scenarios that motivated their inclusion in the platform libraries.

Our goal is to give readers a set of design rules and mental models that make it easier—and more fun—to build correct, performant concurrent classes and applications in Java.

We hope you enjoy *Java Concurrency in Practice*.

Brian Goetz
Williston, VT
March 2006

How to use this book

To address the abstraction mismatch between Java's low-level mechanisms and the necessary design-level policies, we present a *simplified* set of rules for writing concurrent programs. Experts may look at these rules and say "Hmm, that's not entirely true: class *C* is thread-safe even though it violates rule *R*." While it is possible to write correct programs that break our rules, doing so requires a deep understanding of the low-level details of the Java Memory Model, and we want developers to be able to write correct concurrent programs *without* having to master these details. Consistently following our simplified rules will produce correct and maintainable concurrent programs.

We assume the reader already has some familiarity with the basic mechanisms for concurrency in Java. *Java Concurrency in Practice* is not an introduction to concurrency—for that, see the threading chapter of any decent introductory volume, such as *The Java Programming Language* (Arnold et al., 2005). Nor is it an encyclopedic reference for All Things Concurrency—for that, see *Concurrent Programming in Java* (Lea, 2000). Rather, it offers practical design rules to assist developers in the difficult process of creating safe and performant concurrent classes. Where appropriate, we cross-reference relevant sections of *The Java Programming Language*, *Concurrent Programming in Java*, *The Java Language Specification* (Gosling et al., 2005), and *Effective Java* (Bloch, 2001) using the conventions [JPL n.m], [CPJ n.m], [JLS n.m], and [EJ Item n].

After the introduction (Chapter 1), the book is divided into four parts:

Fundamentals. Part I (Chapters 2-5) focuses on the basic concepts of concurrency and thread safety, and how to compose thread-safe classes out of the concurrent building blocks provided by the class library. A "cheat sheet" summarizing the most important of the rules presented in Part I appears on page 110.

Chapters 2 (Thread Safety) and 3 (Sharing Objects) form the foundation for the book. Nearly all of the rules on avoiding concurrency hazards, constructing thread-safe classes, and verifying thread safety are here. Readers who prefer "practice" to "theory" may be tempted to skip ahead to Part II, but make sure to come back and read Chapters 2 and 3 before writing any concurrent code!

Chapter 4 (Composing Objects) covers techniques for composing thread-safe classes into larger thread-safe classes. Chapter 5 (Building Blocks) covers the concurrent building blocks—thread-safe collections and synchronizers—provided by the platform libraries.

Structuring Concurrent Applications. Part II (Chapters 6-9) describes how to exploit threads to improve the throughput or responsiveness of concurrent applications. Chapter 6 (Task Execution) covers identifying parallelizable tasks and executing them within the task-execution framework. Chapter 7 (Cancellation and Shutdown) deals with techniques for convincing tasks and threads to terminate before they would normally do so; how programs deal with cancellation and shutdown is often one of the factors that separates truly robust concurrent applications from those that merely work. Chapter 8 (Applying Thread Pools) addresses some of the more advanced features of the task-execution framework.

Chapter 9 (GUI Applications) focuses on techniques for improving responsiveness in single-threaded subsystems.

Liveness, Performance, and Testing. Part III (Chapters 10-12) concerns itself with ensuring that concurrent programs actually do what you want them to do and do so with acceptable performance. Chapter 10 (Avoiding Liveness Hazards) describes how to avoid liveness failures that can prevent programs from making forward progress. Chapter 11 (Performance and Scalability) covers techniques for improving the performance and scalability of concurrent code. Chapter 12 (Testing Concurrent Programs) covers techniques for testing concurrent code for both correctness and performance.

Advanced Topics. Part IV (Chapters 13-16) covers topics that are likely to be of interest only to experienced developers: explicit locks, atomic variables, nonblocking algorithms, and developing custom synchronizers.

Code examples

While many of the general concepts in this book are applicable to versions of Java prior to Java 5.0 and even to non-Java environments, most of the code examples (and all the statements about the Java Memory Model) assume Java 5.0 or later. Some of the code examples may use library features added in Java 6.

The code examples have been compressed to reduce their size and to highlight the relevant portions. The full versions of the code examples, as well as supplementary examples and errata, are available from the book's website, `http://www.javaconcurrencyinpractice.com`.

The code examples are of three sorts: "good" examples, "not so good" examples, and "bad" examples. Good examples illustrate techniques that should be emulated. Bad examples illustrate techniques that should definitely *not* be emulated, and are identified with a "Mr. Yuk" icon[1] to make it clear that this is "toxic" code (see Listing 1). Not-so-good examples illustrate techniques that are not *necessarily* wrong but are fragile, risky, or perform poorly, and are decorated with a "Mr. Could Be Happier" icon as in Listing 2.

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    // Never returns the wrong answer!
    System.exit(0);
}
```

Listing 1. Bad way to sort a list. *Don't do this.*

Some readers may question the role of the "bad" examples in this book; after all, a book should show how to do things right, not wrong. The bad examples have two purposes. They illustrate common pitfalls, but more importantly they demonstrate how to analyze a program for thread safety—and the best way to do that is to see the ways in which thread safety is compromised.

[1] Mr. Yuk is a registered trademark of the Children's Hospital of Pittsburgh and appears by permission.

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    for (int i=0; i<1000000; i++)
        doNothing();
    Collections.sort(list);
}
```

LISTING 2. Less than optimal way to sort a list.

Acknowledgments

This book grew out of the development process for the java.util.concurrent package that was created by the Java Community Process JSR 166 for inclusion in Java 5.0. Many others contributed to JSR 166; in particular we thank Martin Buchholz for doing all the work related to getting the code into the JDK, and all the readers of the concurrency-interest mailing list who offered their suggestions and feedback on the draft APIs.

This book has been tremendously improved by the suggestions and assistance of a small army of reviewers, advisors, cheerleaders, and armchair critics. We would like to thank Dion Almaer, Tracy Bialik, Cindy Bloch, Martin Buchholz, Paul Christmann, Cliff Click, Stuart Halloway, David Hovemeyer, Jason Hunter, Michael Hunter, Jeremy Hylton, Heinz Kabutz, Robert Kuhar, Ramnivas Laddad, Jared Levy, Nicole Lewis, Victor Luchangco, Jeremy Manson, Paul Martin, Berna Massingill, Michael Maurer, Ted Neward, Kirk Pepperdine, Bill Pugh, Sam Pullara, Russ Rufer, Bill Scherer, Jeffrey Siegal, Bruce Tate, Gil Tene, Paul Tyma, and members of the Silicon Valley Patterns Group who, through many interesting technical conversations, offered guidance and made suggestions that helped make this book better.

We are especially grateful to Cliff Biffle, Barry Hayes, Dawid Kurzyniec, Angelika Langer, Doron Rajwan, and Bill Venners, who reviewed the entire manuscript in excruciating detail, found bugs in the code examples, and suggested numerous improvements.

We thank Katrina Avery for a great copy-editing job and Rosemary Simpson for producing the index under unreasonable time pressure. We thank Ami Dewar for doing the illustrations.

Thanks to the whole team at Addison-Wesley who helped make this book a reality. Ann Sellers got the project launched and Greg Doench shepherded it to a smooth completion; Elizabeth Ryan guided it through the production process.

We would also like to thank the thousands of software engineers who contributed indirectly by creating the software used to create this book, including TeX, LaTeX, Adobe Acrobat, pic, grap, Adobe Illustrator, Perl, Apache Ant, IntelliJ IDEA, GNU emacs, Subversion, TortoiseSVN, and of course, the Java platform and class libraries.

This page intentionally left blank

Chapter 6

Task Execution

Most concurrent applications are organized around the execution of *tasks*: abstract, discrete units of work. Dividing the work of an application into tasks simplifies program organization, facilitates error recovery by providing natural transaction boundaries, and promotes concurrency by providing a natural structure for parallelizing work.

6.1 Executing tasks in threads

The first step in organizing a program around task execution is identifying sensible *task boundaries*. Ideally, tasks are *independent* activities: work that doesn't depend on the state, result, or side effects of other tasks. Independence facilitates concurrency, as independent tasks can be executed in parallel if there are adequate processing resources. For greater flexibility in scheduling and load balancing tasks, each task should also represent a small fraction of your application's processing capacity.

Server applications should exhibit both *good throughput* and *good responsiveness* under normal load. Application providers want applications to support as many users as possible, so as to reduce provisioning costs per user; users want to get their response quickly. Further, applications should exhibit *graceful degradation* as they become overloaded, rather than simply falling over under heavy load. Choosing good task boundaries, coupled with a sensible *task execution policy* (see Section 6.2.2), can help achieve these goals.

Most server applications offer a natural choice of task boundary: individual client requests. Web servers, mail servers, file servers, EJB containers, and database servers all accept requests via network connections from remote clients. Using individual requests as task boundaries usually offers both independence and appropriate task sizing. For example, the result of submitting a message to a mail server is not affected by the other messages being processed at the same time, and handling a single message usually requires a very small percentage of the server's total capacity.

6.1.1 Executing tasks sequentially

There are a number of possible policies for scheduling tasks within an application, some of which exploit the potential for concurrency better than others. The simplest is to execute tasks sequentially in a single thread. `SingleThreadWebServer` in Listing 6.1 processes its tasks—HTTP requests arriving on port 80—sequentially. The details of the request processing aren't important; we're interested in characterizing the concurrency of various scheduling policies.

```
class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

Listing 6.1. Sequential web server.

`SingleThreadedWebServer` is simple and theoretically correct, but would perform poorly in production because it can handle only one request at a time. The main thread alternates between accepting connections and processing the associated request. While the server is handling a request, new connections must wait until it finishes the current request and calls `accept` again. This might work if request processing were so fast that `handleRequest` effectively returned immediately, but this doesn't describe any web server in the real world.

Processing a web request involves a mix of computation and I/O. The server must perform socket I/O to read the request and write the response, which can block due to network congestion or connectivity problems. It may also perform file I/O or make database requests, which can also block. In a single-threaded server, blocking not only delays completing the current request, but prevents pending requests from being processed at all. If one request blocks for an unusually long time, users might think the server is unavailable because it appears unresponsive. At the same time, resource utilization is poor, since the CPU sits idle while the single thread waits for its I/O to complete.

In server applications, sequential processing rarely provides either good throughput or good responsiveness. There are exceptions—such as when tasks are few and long-lived, or when the server serves a single client that makes only a single request at a time—but most server applications do not work this way.[1]

[1] In some situations, sequential processing may offer a simplicity or safety advantage; most GUI frameworks process tasks sequentially using a single thread. We return to the sequential model in Chapter 9.

6.1.2 Explicitly creating threads for tasks

A more responsive approach is to create a new thread for servicing each request, as shown in ThreadPerTaskWebServer in Listing 6.2.

```
class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                    public void run() {
                        handleRequest(connection);
                    }
                };
            new Thread(task).start();
        }
    }
}
```

Listing 6.2. Web server that starts a new thread for each request.

ThreadPerTaskWebServer is similar in structure to the single-threaded version—the main thread still alternates between accepting an incoming connection and dispatching the request. The difference is that for each connection, the main loop creates a new thread to process the request instead of processing it within the main thread. This has three main consequences:

- Task processing is offloaded from the main thread, enabling the main loop to resume waiting for the next incoming connection more quickly. This enables new connections to be accepted before previous requests complete, improving responsiveness.
- Tasks can be processed in parallel, enabling multiple requests to be serviced simultaneously. This may improve throughput if there are multiple processors, or if tasks need to block for any reason such as I/O completion, lock acquisition, or resource availability.
- Task-handling code must be thread-safe, because it may be invoked concurrently for multiple tasks.

Under light to moderate load, the thread-per-task approach is an improvement over sequential execution. As long as the request arrival rate does not exceed the server's capacity to handle requests, this approach offers better responsiveness and throughput.

6.1.3 Disadvantages of unbounded thread creation

For production use, however, the thread-per-task approach has some practical drawbacks, especially when a large number of threads may be created:

Thread lifecycle overhead. Thread creation and teardown are not free. The actual overhead varies across platforms, but thread creation takes time, introducing latency into request processing, and requires some processing activity by the JVM and OS. If requests are frequent and lightweight, as in most server applications, creating a new thread for each request can consume significant computing resources.

Resource consumption. Active threads consume system resources, especially memory. When there are more runnable threads than available processors, threads sit idle. Having many idle threads can tie up a lot of memory, putting pressure on the garbage collector, and having many threads competing for the CPUs can impose other performance costs as well. If you have enough threads to keep all the CPUs busy, creating more threads won't help and may even hurt.

Stability. There is a limit on how many threads can be created. The limit varies by platform and is affected by factors including JVM invocation parameters, the requested stack size in the `Thread` constructor, and limits on threads placed by the underlying operating system.[2] When you hit this limit, the most likely result is an `OutOfMemoryError`. Trying to recover from such an error is very risky; it is far easier to structure your program to avoid hitting this limit.

Up to a certain point, more threads can improve throughput, but beyond that point creating more threads just slows down your application, and creating one thread too many can cause your entire application to crash horribly. The way to stay out of danger is to place some bound on how many threads your application creates, and to test your application thoroughly to ensure that, even when this bound is reached, it does not run out of resources.

The problem with the thread-per-task approach is that nothing places any limit on the number of threads created except the rate at which remote users can throw HTTP requests at it. Like other concurrency hazards, unbounded thread creation may *appear* to work just fine during prototyping and development, with problems surfacing only when the application is deployed and under heavy load. So a malicious user, or enough ordinary users, can make your web server crash if the traffic load ever reaches a certain threshold. For a server application that is supposed to provide high availability and graceful degradation under load, this is a serious failing.

[2] On 32-bit machines, a major limiting factor is address space for thread stacks. Each thread maintains two execution stacks, one for Java code and one for native code. Typical JVM defaults yield a combined stack size of around half a megabyte. (You can change this with the `-Xss` JVM flag or through the `Thread` constructor.) If you divide the per-thread stack size into 2^{32}, you get a limit of a few thousands or tens of thousands of threads. Other factors, such as OS limitations, may impose stricter limits.

6.2 The Executor framework

Tasks are logical units of work, and threads are a mechanism by which tasks can run asynchronously. We've examined two policies for executing tasks using threads—execute tasks sequentially in a single thread, and execute each task in its own thread. Both have serious limitations: the sequential approach suffers from poor responsiveness and throughput, and the thread-per-task approach suffers from poor resource management.

In Chapter 5, we saw how to use *bounded queues* to prevent an overloaded application from running out of memory. *Thread pools* offer the same benefit for thread management, and `java.util.concurrent` provides a flexible thread pool implementation as part of the `Executor` framework. The primary abstraction for task execution in the Java class libraries is *not* `Thread`, but `Executor`, shown in Listing 6.3.

```
public interface Executor {
    void execute(Runnable command);
}
```

Listing 6.3. `Executor` interface.

`Executor` may be a simple interface, but it forms the basis for a flexible and powerful framework for asynchronous task execution that supports a wide variety of task execution policies. It provides a standard means of decoupling *task submission* from *task execution*, describing tasks with `Runnable`. The `Executor` implementations also provide lifecycle support and hooks for adding statistics gathering, application management, and monitoring.

`Executor` is based on the producer-consumer pattern, where activities that submit tasks are the producers (producing units of work to be done) and the threads that execute tasks are the consumers (consuming those units of work). *Using an `Executor` is usually the easiest path to implementing a producer-consumer design in your application.*

6.2.1 Example: web server using Executor

Building a web server with an `Executor` is easy. `TaskExecutionWebServer` in Listing 6.4 replaces the hard-coded thread creation with an `Executor`. In this case, we use one of the standard `Executor` implementations, a fixed-size thread pool with 100 threads.

In `TaskExecutionWebServer`, submission of the request-handling task is decoupled from its execution using an `Executor`, and its behavior can be changed merely by substituting a different `Executor` implementation. Changing `Executor` implementations or configuration is far less invasive than changing the way tasks are submitted; `Executor` configuration is generally a one-time event and can easily be exposed for deployment-time configuration, whereas task submission code tends to be strewn throughout the program and harder to expose.

```
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

Listing 6.4. Web server using a thread pool.

We can easily modify `TaskExecutionWebServer` to behave like `ThreadPerTaskWebServer` by substituting an `Executor` that creates a new thread for each request. Writing such an `Executor` is trivial, as shown in `ThreadPerTaskExecutor` in Listing 6.5.

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    };
}
```

Listing 6.5. `Executor` that starts a new thread for each task.

Similarly, it is also easy to write an `Executor` that would make `TaskExecutionWebServer` behave like the single-threaded version, executing each task synchronously before returning from `execute`, as shown in `WithinThreadExecutor` in Listing 6.6.

6.2.2 Execution policies

The value of decoupling submission from execution is that it lets you easily specify, and subsequently change without great difficulty, the *execution policy* for a given class of tasks. An execution policy specifies the "what, where, when, and how" of task execution, including:

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    };
}
```

Listing 6.6. `Executor` that executes tasks synchronously in the calling thread.

- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO, LIFO, priority order)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected as the victim, and how should the application be notified?
- What actions should be taken before or after executing a task?

Execution policies are a resource management tool, and the optimal policy depends on the available computing resources and your quality-of-service requirements. By limiting the number of concurrent tasks, you can ensure that the application does not fail due to resource exhaustion or suffer performance problems due to contention for scarce resources.[3] Separating the specification of execution policy from task submission makes it practical to select an execution policy at deployment time that is matched to the available hardware.

Whenever you see code of the form:

```
    new Thread(runnable).start()
```

and you think you might at some point want a more flexible execution policy, seriously consider replacing it with the use of an `Executor`.

6.2.3 Thread pools

A thread pool, as its name suggests, manages a homogeneous pool of worker threads. A thread pool is tightly bound to a *work queue* holding tasks waiting to be executed. Worker threads have a simple life: request the next task from the work queue, execute it, and go back to waiting for another task.

[3] This is analogous to one of the roles of a transaction monitor in an enterprise application: it can throttle the rate at which transactions are allowed to proceed so as not to exhaust or overstress limited resources.

Executing tasks in pool threads has a number of advantages over the thread-per-task approach. Reusing an existing thread instead of creating a new one amortizes thread creation and teardown costs over multiple requests. As an added bonus, since the worker thread often already exists at the time the request arrives, the latency associated with thread creation does not delay task execution, thus improving responsiveness. By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources.

The class library provides a flexible thread pool implementation along with some useful predefined configurations. You can create a thread pool by calling one of the static factory methods in `Executors`:

`newFixedThreadPool`. A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected `Exception`).

`newCachedThreadPool`. A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

`newSingleThreadExecutor`. A single-threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO, priority order).[4]

`newScheduledThreadPool`. A fixed-size thread pool that supports delayed and periodic task execution, similar to `Timer`. (See Section 6.2.5.)

The `newFixedThreadPool` and `newCachedThreadPool` factories return instances of the general-purpose `ThreadPoolExecutor`, which can also be used directly to construct more specialized executors. We discuss thread pool configuration options in depth in Chapter 8.

The web server in `TaskExecutionWebServer` uses an `Executor` with a bounded pool of worker threads. Submitting a task with `execute` adds the task to the work queue, and the worker threads repeatedly dequeue tasks from the work queue and execute them.

Switching from a thread-per-task policy to a pool-based policy has a big effect on application stability: the web server will no longer fail under heavy load.[5]

[4] Single-threaded executors also provide sufficient internal synchronization to guarantee that any memory writes made by tasks are visible to subsequent tasks; this means that objects can be safely confined to the "task thread" even though that thread may be replaced with another from time to time.

[5] While the server may not fail due to the creation of too many threads, if the task arrival rate exceeds the task service rate for long enough it is still possible (just harder) to run out of memory because of the growing queue of `Runnable`s awaiting execution. This can be addressed within the `Executor` framework by using a bounded work queue—see Section 8.3.2.

It also degrades more gracefully, since it does not create thousands of threads that compete for limited CPU and memory resources. And using an Executor opens the door to all sorts of additional opportunities for tuning, management, monitoring, logging, error reporting, and other possibilities that would have been far more difficult to add without a task execution framework.

6.2.4 Executor lifecycle

We've seen how to create an Executor but not how to shut one down. An Executor implementation is likely to create threads for processing tasks. But the JVM can't exit until all the (nondaemon) threads have terminated, so failing to shut down an Executor could prevent the JVM from exiting.

Because an Executor processes tasks asynchronously, at any given time the state of previously submitted tasks is not immediately obvious. Some may have completed, some may be currently running, and others may be queued awaiting execution. In shutting down an application, there is a spectrum from graceful shutdown (finish what you've started but don't accept any new work) to abrupt shutdown (turn off the power to the machine room), and various points in between. Since Executors provide a service to applications, they should be able to be shut down as well, both gracefully and abruptly, and feed back information to the application about the status of tasks that were affected by the shutdown.

To address the issue of execution service lifecycle, the ExecutorService interface extends Executor, adding a number of methods for lifecycle management (as well as some convenience methods for task submission). The lifecycle management methods of ExecutorService are shown in Listing 6.7.

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    // ... additional convenience methods for task submission
}
```

Listing 6.7. Lifecycle methods in ExecutorService.

The lifecycle implied by ExecutorService has three states—*running*, *shutting down*, and *terminated*. ExecutorServices are initially created in the *running* state. The shutdown method initiates a graceful shutdown: no new tasks are accepted but previously submitted tasks are allowed to complete—including those that have not yet begun execution. The shutdownNow method initiates an abrupt shutdown: it attempts to cancel outstanding tasks and does not start any tasks that are queued but not begun.

Tasks submitted to an ExecutorService after it has been shut down are handled by the *rejected execution handler* (see Section 8.3.3), which might silently discard the task or might cause execute to throw the unchecked RejectedExecutionException. Once all tasks have completed, the ExecutorService transitions to the *terminated* state. You can wait for an ExecutorService to reach the terminated state with awaitTermination, or poll for whether it has yet terminated with isTerminated. It is common to follow shutdown immediately by awaitTermination, creating the effect of synchronously shutting down the ExecutorService. (Executor shutdown and task cancellation are covered in more detail in Chapter 7.)

LifecycleWebServer in Listing 6.8 extends our web server with lifecycle support. It can be shut down in two ways: programmatically by calling stop, and through a client request by sending the web server a specially formatted HTTP request.

```
class LifecycleWebServer {
    private final ExecutorService exec = ...;

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

Listing 6.8. Web server with shutdown support.

6.2.5 Delayed and periodic tasks

The Timer facility manages the execution of deferred ("run this task in 100 ms") and periodic ("run this task every 10 ms") tasks. However, Timer has some drawbacks, and ScheduledThreadPoolExecutor should be thought of as its replacement.[6] You can construct a ScheduledThreadPoolExecutor through its constructor or through the newScheduledThreadPool factory.

A Timer creates only a single thread for executing timer tasks. If a timer task takes too long to run, the timing accuracy of other TimerTasks can suffer. If a recurring TimerTask is scheduled to run every 10 ms and another TimerTask takes 40 ms to run, the recurring task either (depending on whether it was scheduled at fixed rate or fixed delay) gets called four times in rapid succession after the long-running task completes, or "misses" four invocations completely. Scheduled thread pools address this limitation by letting you provide multiple threads for executing deferred and periodic tasks.

Another problem with Timer is that it behaves poorly if a TimerTask throws an unchecked exception. The Timer thread doesn't catch the exception, so an unchecked exception thrown from a TimerTask terminates the timer thread. Timer also doesn't resurrect the thread in this situation; instead, it erroneously assumes the entire Timer was cancelled. In this case, TimerTasks that are already scheduled but not yet executed are never run, and new tasks cannot be scheduled. (This problem, called "thread leakage" is described in Section 7.3, along with techniques for avoiding it.)

OutOfTime in Listing 6.9 illustrates how a Timer can become confused in this manner and, as confusion loves company, how the Timer shares its confusion with the next hapless caller that tries to submit a TimerTask. You might expect the program to run for six seconds and exit, but what actually happens is that it terminates after one second with an IllegalStateException whose message text is "Timer already cancelled". ScheduledThreadPoolExecutor deals properly with ill-behaved tasks; there is little reason to use Timer in Java 5.0 or later.

If you need to build your own scheduling service, you may still be able to take advantage of the library by using a DelayQueue, a BlockingQueue implementation that provides the scheduling functionality of ScheduledThreadPoolExecutor. A DelayQueue manages a collection of Delayed objects. A Delayed has a delay time associated with it: DelayQueue lets you take an element only if its delay has expired. Objects are returned from a DelayQueue ordered by the time associated with their delay.

6.3 Finding exploitable parallelism

The Executor framework makes it easy to specify an execution policy, but in order to use an Executor, you have to be able to describe your task as a Runnable. In most server applications, there is an obvious task boundary: a single client request. But sometimes good task boundaries are not quite so obvious, as

[6] Timer does have support for scheduling based on absolute, not relative time, so that tasks can be sensitive to changes in the system clock; ScheduledThreadPoolExecutor supports only relative time.

```
public class OutOfTime {
    public static void main(String[] args) throws Exception {
        Timer timer = new Timer();
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(1);
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(5);
    }

    static class ThrowTask extends TimerTask {
        public void run() { throw new RuntimeException(); }
    }
}
```

Listing 6.9. Class illustrating confusing `Timer` behavior.

in many desktop applications. There may also be exploitable parallelism within a single client request in server applications, as is sometimes the case in database servers. (For a further discussion of the competing design forces in choosing task boundaries, see [CPJ 4.4.1.1].)

In this section we develop several versions of a component that admit varying degrees of concurrency. Our sample component is the page-rendering portion of a browser application, which takes a page of HTML and renders it into an image buffer. To keep it simple, we assume that the HTML consists only of marked up text interspersed with image elements with pre-specified dimensions and URLs.

6.3.1 Example: sequential page renderer

The simplest approach is to process the HTML document sequentially. As text markup is encountered, render it into the image buffer; as image references are encountered, fetch the image over the network and draw it into the image buffer as well. This is easy to implement and requires touching each element of the input only once (it doesn't even require buffering the document), but is likely to annoy the user, who may have to wait a long time before all the text is rendered.

A less annoying but still sequential approach involves rendering the text elements first, leaving rectangular placeholders for the images, and after completing the initial pass on the document, going back and downloading the images and drawing them into the associated placeholder. This approach is shown in `SingleThreadRenderer` in Listing 6.10.

Downloading an image mostly involves waiting for I/O to complete, and during this time the CPU does little work. So the sequential approach may underutilize the CPU, and also makes the user wait longer than necessary to see the finished page. We can achieve better utilization and responsiveness by breaking the problem into independent tasks that can execute concurrently.

```
public class SingleThreadRenderer {
    void renderPage(CharSequence source) {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : scanForImageInfo(source))
            imageData.add(imageInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }
}
```

Listing 6.10. Rendering page elements sequentially.

6.3.2 Result-bearing tasks: `Callable` and `Future`

The `Executor` framework uses `Runnable` as its basic task representation. `Runnable` is a fairly limiting abstraction; `run` cannot return a value or throw checked exceptions, although it can have side effects such as writing to a log file or placing a result in a shared data structure.

Many tasks are effectively deferred computations—executing a database query, fetching a resource over the network, or computing a complicated function. For these types of tasks, `Callable` is a better abstraction: it expects that the main entry point, `call`, will return a value and anticipates that it might throw an exception.[7] `Executors` includes several utility methods for wrapping other types of tasks, including `Runnable` and `java.security.PrivilegedAction`, with a `Callable`.

`Runnable` and `Callable` describe abstract computational tasks. Tasks are usually finite: they have a clear starting point and they eventually terminate. The lifecycle of a task executed by an `Executor` has four phases: *created*, *submitted*, *started*, and *completed*. Since tasks can take a long time to run, we also want to be able to cancel a task. In the `Executor` framework, tasks that have been submitted but not yet started can always be cancelled, and tasks that have started can sometimes be cancelled if they are responsive to interruption. Cancelling a task that has already completed has no effect. (Cancellation is covered in greater detail in Chapter 7.)

`Future` represents the lifecycle of a task and provides methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task. `Callable` and `Future` are shown in Listing 6.11. Implicit in the specification of `Future` is that task lifecycle can only move forwards, not backwards—just like the `ExecutorService` lifecycle. Once a task is completed, it stays in that state forever.

The behavior of `get` varies depending on the task state (not yet started, running, completed). It returns immediately or throws an `Exception` if the task has already completed, but if not it blocks until the task completes. If the task completes by throwing an exception, `get` rethrows it wrapped in an `Execution-`

[7] To express a non-value-returning task with `Callable`, use `Callable<Void>`.

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
                   CancellationException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
               CancellationException, TimeoutException;
}
```

Listing 6.11. Callable and Future interfaces.

Exception; if it was cancelled, get throws CancellationException. If get throws ExecutionException, the underlying exception can be retrieved with getCause.

There are several ways to create a Future to describe a task. The submit methods in ExecutorService all return a Future, so that you can submit a Runnable or a Callable to an executor and get back a Future that can be used to retrieve the result or cancel the task. You can also explicitly instantiate a FutureTask for a given Runnable or Callable. (Because FutureTask implements Runnable, it can be submitted to an Executor for execution or executed directly by calling its run method.)

As of Java 6, ExecutorService implementations can override newTaskFor in AbstractExecutorService to control instantiation of the Future corresponding to a submitted Callable or Runnable. The default implementation just creates a new FutureTask, as shown in Listing 6.12.

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
    return new FutureTask<T>(task);
}
```

Listing 6.12. Default implementation of newTaskFor in ThreadPoolExecutor.

Submitting a Runnable or Callable to an Executor constitutes a safe publication (see Section 3.5) of the Runnable or Callable from the submitting thread to the thread that will eventually execute the task. Similarly, setting the result value for a Future constitutes a safe publication of the result from the thread in which it was computed to any thread that retrieves it via get.

6.3.3 Example: page renderer with `Future`

As a first step towards making the page renderer more concurrent, let's divide it into two tasks, one that renders the text and one that downloads all the images. (Because one task is largely CPU-bound and the other is largely I/O-bound, this approach may yield improvements even on single-CPU systems.)

`Callable` and `Future` can help us express the interaction between these cooperating tasks. In `FutureRenderer` in Listing 6.13, we create a `Callable` to download all the images, and submit it to an `ExecutorService`. This returns a `Future` describing the task's execution; when the main task gets to the point where it needs the images, it waits for the result by calling `Future.get`. If we're lucky, the results will already be ready by the time we ask; otherwise, at least we got a head start on downloading the images.

The state-dependent nature of `get` means that the caller need not be aware of the state of the task, and the safe publication properties of task submission and result retrieval make this approach thread-safe. The exception handling code surrounding `Future.get` deals with two possible problems: that the task encountered an `Exception`, or the thread calling `get` was interrupted before the results were available. (See Sections 5.5.2 and 5.4.)

`FutureRenderer` allows the text to be rendered concurrently with downloading the image data. When all the images are downloaded, they are rendered onto the page. This is an improvement in that the user sees a result quickly and it exploits some parallelism, but we can do considerably better. There is no need for users to wait for *all* the images to be downloaded; they would probably prefer to see individual images drawn as they become available.

6.3.4 Limitations of parallelizing heterogeneous tasks

In the last example, we tried to execute two different types of tasks in parallel—downloading the images and rendering the page. But obtaining significant performance improvements by trying to parallelize sequential heterogeneous tasks can be tricky.

Two people can divide the work of cleaning the dinner dishes fairly effectively: one person washes while the other dries. However, assigning a different type of task to each worker does not scale well; if several more people show up, it is not obvious how they can help without getting in the way or significantly restructuring the division of labor. Without finding finer-grained parallelism among similar tasks, this approach will yield diminishing returns.

A further problem with dividing heterogeneous tasks among multiple workers is that the tasks may have disparate sizes. If you divide tasks *A* and *B* between two workers but *A* takes ten times as long as *B*, you've only speeded up the total process by 9%. Finally, dividing a task among multiple workers always involves some amount of coordination overhead; for the division to be worthwhile, this overhead must be more than compensated by productivity improvements due to parallelism.

`FutureRenderer` uses two tasks: one for rendering text and one for downloading the images. If rendering the text is much faster than downloading the images,

```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result
                                = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // Re-assert the thread's interrupted status
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Listing 6.13. Waiting for image download with Future.

as is entirely possible, the resulting performance is not much different from the sequential version, but the code is a lot more complicated. And the best we can do with two threads is speed things up by a factor of two. Thus, trying to increase concurrency by parallelizing heterogeneous activities can be a lot of work, and there is a limit to how much additional concurrency you can get out of it. (See Sections 11.4.2 and 11.4.3 for another example of the same phenomenon.)

> The real performance payoff of dividing a program's workload into tasks comes when there are a large number of independent, *homogeneous* tasks that can be processed concurrently.

6.3.5 `CompletionService`: `Executor` meets `BlockingQueue`

If you have a batch of computations to submit to an `Executor` and you want to retrieve their results as they become available, you could retain the `Future` associated with each task and repeatedly poll for completion by calling `get` with a timeout of zero. This is possible, but tedious. Fortunately there is a better way: a *completion service.*

`CompletionService` combines the functionality of an `Executor` and a `BlockingQueue`. You can submit `Callable` tasks to it for execution and use the queue-like methods `take` and `poll` to retrieve completed results, packaged as `Futures`, as they become available. `ExecutorCompletionService` implements `CompletionService`, delegating the computation to an `Executor`.

The implementation of `ExecutorCompletionService` is quite straightforward. The constructor creates a `BlockingQueue` to hold the completed results. `FutureTask` has a `done` method that is called when the computation completes. When a task is submitted, it is wrapped with a `QueueingFuture`, a subclass of `FutureTask` that overrides `done` to place the result on the `BlockingQueue`, as shown in Listing 6.14. The `take` and `poll` methods delegate to the `BlockingQueue`, blocking if results are not yet available.

```
private class QueueingFuture<V> extends FutureTask<V> {
    QueueingFuture(Callable<V> c) { super(c); }
    QueueingFuture(Runnable t, V r) { super(t, r); }

    protected void done() {
        completionQueue.add(this);
    }
}
```

LISTING 6.14. `QueueingFuture` class used by `ExecutorCompletionService`.

6.3.6 Example: page renderer with `CompletionService`

We can use a `CompletionService` to improve the performance of the page renderer in two ways: shorter total runtime and improved responsiveness. We can create a separate task for downloading *each* image and execute them in a thread pool, turning the sequential download into a parallel one: this reduces the amount of time to download all the images. And by fetching results from the `CompletionService` and rendering each image as soon as it is available, we can give the user a more dynamic and responsive user interface. This implementation is shown in `Renderer` in Listing 6.15.

```
public class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) { this.executor = executor; }

    void renderPage(CharSequence source) {
        List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                 public ImageData call() {
                     return imageInfo.downloadImage();
                 }
            });

        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Listing 6.15. Using `CompletionService` to render page elements as they become available.

Multiple `ExecutorCompletionServices` can share a single `Executor`, so it is perfectly sensible to create an ExecutorCompletionService that is private to a particular computation while sharing a common Executor. When used in this way, a CompletionService acts as a handle for a batch of computations in much the same way that a Future acts as a handle for a single computation. By remembering how many tasks were submitted to the CompletionService and counting how many completed results are retrieved, you can know when all the results for a given batch have been retrieved, even if you use a shared Executor.

6.3.7 Placing time limits on tasks

Sometimes, if an activity does not complete within a certain amount of time, the result is no longer needed and the activity can be abandoned. For example, a web application may fetch its advertisements from an external ad server, but if the ad is not available within two seconds, it instead displays a default advertisement so that ad unavailability does not undermine the site's responsiveness requirements. Similarly, a portal site may fetch data in parallel from multiple data sources, but may be willing to wait only a certain amount of time for data to be available before rendering the page without it.

The primary challenge in executing tasks within a time budget is making sure that you don't wait longer than the time budget to get an answer or find out that one is not forthcoming. The timed version of Future.get supports this requirement: it returns as soon as the result is ready, but throws TimeoutException if the result is not ready within the timeout period.

A secondary problem when using timed tasks is to stop them when they run out of time, so they do not waste computing resources by continuing to compute a result that will not be used. This can be accomplished by having the task strictly manage its own time budget and abort if it runs out of time, or by cancelling the task if the timeout expires. Again, Future can help; if a timed get completes with a TimeoutException, you can cancel the task through the Future. If the task is written to be cancellable (see Chapter 7), it can be terminated early so as not to consume excessive resources. This technique is used in Listings 6.13 and 6.16.

Listing 6.16 shows a typical application of a timed Future.get. It generates a composite web page that contains the requested content plus an advertisement fetched from an ad server. It submits the ad-fetching task to an executor, computes the rest of the page content, and then waits for the ad until its time budget runs out.[8] If the get times out, it cancels[9] the ad-fetching task and uses a default advertisement instead.

6.3.8 Example: a travel reservations portal

The time-budgeting approach in the previous section can be easily generalized to an arbitrary number of tasks. Consider a travel reservation portal: the user en-

[8] The timeout passed to get is computed by subtracting the current time from the deadline; this may in fact yield a negative number, but all the timed methods in java.util.concurrent treat negative timeouts as zero, so no extra code is needed to deal with this case.

[9] The true parameter to Future.cancel means that the task thread can be interrupted if the task is currently running; see Chapter 7.

```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // Render the page while waiting for the ad
    Page page = renderPageBody();
    Ad ad;
    try {
        // Only wait for the remaining time budget
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}
```

Listing 6.16. Fetching an advertisement with a time budget.

ters travel dates and requirements and the portal fetches and displays bids from a number of airlines, hotels or car rental companies. Depending on the company, fetching a bid might involve invoking a web service, consulting a database, performing an EDI transaction, or some other mechanism. Rather than have the response time for the page be driven by the slowest response, it may be preferable to present only the information available within a given time budget. For providers that do not respond in time, the page could either omit them completely or display a placeholder such as "Did not hear from Air Java in time."

Fetching a bid from one company is independent of fetching bids from another, so fetching a single bid is a sensible task boundary that allows bid retrieval to proceed concurrently. It would be easy enough to create *n* tasks, submit them to a thread pool, retain the `Future`s, and use a timed `get` to fetch each result sequentially via its `Future`, but there is an even easier way—`invokeAll`.

Listing 6.17 uses the timed version of `invokeAll` to submit multiple tasks to an `ExecutorService` and retrieve the results. The `invokeAll` method takes a collection of tasks and returns a collection of `Future`s. The two collections have identical structures; `invokeAll` adds the `Future`s to the returned collection in the order imposed by the task collection's iterator, thus allowing the caller to associate a `Future` with the `Callable` it represents. The timed version of `invokeAll` will return when all the tasks have completed, the calling thread is interrupted, or the timeout expires. Any tasks that are not complete when the timeout expires are cancelled. On return from `invokeAll`, each task will have either completed normally or been cancelled; the client code can call `get` or `isCancelled` to find out which.

Summary

Structuring applications around the execution of *tasks* can simplify development and facilitate concurrency. The `Executor` framework permits you to decouple task submission from execution policy and supports a rich variety of execution policies; whenever you find yourself creating threads to perform tasks, consider using an `Executor` instead. To maximize the benefit of decomposing an application into tasks, you must identify sensible task boundaries. In some applications, the obvious task boundaries work well, whereas in others some analysis may be required to uncover finer-grained exploitable parallelism.

```java
private class QuoteTask implements Callable<TravelQuote> {
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes(
        TravelInfo travelInfo, Set<TravelCompany> companies,
        Comparator<TravelQuote> ranking, long time, TimeUnit unit)
        throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    List<Future<TravelQuote>> futures =
        exec.invokeAll(tasks, time, unit);

    List<TravelQuote> quotes =
        new ArrayList<TravelQuote>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();
    for (Future<TravelQuote> f : futures) {
        QuoteTask task = taskIter.next();
        try {
            quotes.add(f.get());
        } catch (ExecutionException e) {
            quotes.add(task.getFailureQuote(e.getCause()));
        } catch (CancellationException e) {
            quotes.add(task.getTimeoutQuote(e));
        }
    }

    Collections.sort(quotes, ranking);
    return quotes;
}
```

Listing 6.17. Requesting travel quotes under a time budget.

This page intentionally left blank

Index

Symbols

64-bit operations
nonatomic nature of; 36

A

ABA problem; 336
abnormal thread termination
handling; 161–163
abort saturation policy; **174**
See also lifecycle; termination;
abrupt shutdown
limitations; 158–161
triggers for; 164
vs. graceful shutdown; 153
`AbstractExecutorService`
task representation use; 126
abstractions
See models/modeling; representation;
`AbstractQueuedSynchronizer`
See AQS framework;
access
See also encapsulation; sharing; visibility;
exclusive
and concurrent collections; 86
integrity
nonblocking algorithm use; 319
mutable state
importance of coordinating; 110
remote resource
as long-running GUI task; 195
serialized
`WorkerThread` example; 227_{li}
vs. object serialization; 27_{fn}
visibility role in; 33
`AccessControlContext`
custom thread factory handling; 177
acquisition of locks
See locks, acquisition;
action(s)
See also compound actions; condition, predicate; control flow; task(s);
barrier; **99**
JMM specification; 339–342
listener; 195–197
activity(s)
See also task(s);
cancellation; **135**, 135–150
tasks as representation of; 113
ad-hoc thread confinement; **43**
See also confinement;
algorithm(s)
See also design patterns; idioms; representation;
comparing performance; 263–264
design role of representation; 104
lock-free; **329**
Michael-Scott nonblocking queue; 332
nonblocking; **319**, **329**, 329–336
backoff importance for; 231_{fn}
synchronization; 319–336
`SynchronousQueue`; 174_{fn}
parallel iterative
barrier use in; 99
recursive
parallelizing; 181–188
Treiber's
nonblocking stack; 331_{li}
work stealing
deques and; 92
alien method; **40**
See also untrusted code behavior;
deadlock risks posed by; 211
publication risks; 40

allocation
  advantages vs. synchronization; 242
  immutable objects and; 48_{fn}
  object pool use
    disadvantages of; 241
  scalability advantages; 263
Amdahl's law; 225–229
  See also concurrent/concurrency; performance; resource(s); throughput; utilization;
  lock scope reduction advantage; 234
  qualitative application of; 227
analysis
  See also instrumentation; measurement; static analysis tools;
  deadlock
    thread dump use; 216–217
  escape; **230**
  for exploitable parallelism; 123–133
  lock contention
    thread dump use; 240
  performance; 221–245
annotations; 353–354
  See also documentation;
  for concurrency documentation; 6
  `@GuardedBy`; 28, 354
    synchronization policy documentation use; 75
  `@Immutable`; 353
  `@NotThreadSafe`; 353
  `@ThreadSafe`; 353
AOP (aspect-oriented programming)
  in testing; 259, 273
application(s)
  See also frameworks(s); service(s); tools;
  -scoped objects
    thread safety concerns; 10
  frameworks, and ThreadLocal; 46
  GUI; 189–202
    thread safety concerns; 10–11
  parallelizing
    task decomposition; 113
  shutdown
    and task cancellation; 136
AQS (`AbstractQueuedSynchronizer`) framework; 308, 311–317
  exit protocol use; 306
  `FutureTask` implementation
    piggybacking use; 342
`ArrayBlockingQueue`; 89
  as bounded buffer example; 292
  performance advantages over `BoundedBuffer`; 263
`ArrayDeque`; 92
arrays
  See also collections; data structure(s);
  atomic variable; 325
asymmetric two-party tasks
  `Exchanger` management of; 101
asynchrony/asynchronous
  events, handling; 4
  I/O, and non-interruptable blocking; 148
  sequentiality vs.; 2
  tasks
    execution, `Executor` framework use; 117
    `FutureTask` handling; 95–98
atomic variables; 319–336
  and lock contention; 239–240
  classes; 324–329
  locking vs.; 326–329
  strategies for use; 34
  volatile variables vs.; 39, 324–326
atomic/atomicity; **22**
  See also invariant(s); synchronization; visibility;
  64-bit operations
    nonatomic nature of; 36
  and compound actions; 22–23
  and multivariable invariants; 57, 67–68
  and open call restructuring; 213
  and service shutdown; 153
  and state transition constraints; 56
  caching issues; 24–25
  client-side locking support for; 80
  field updaters; 335–336
  immutable object use for; 48
  in cache implementation design; 106
  intrinsic lock enforcement of; 25–26
  loss
    risk of lock scope reduction; 234
  `Map` operations; 86
  put-if-absent; 71–72
  statistics gathering hooks use; 179
  thread-safety issues
    in servlets with state; 19–23

`AtomicBoolean`; 325

`AtomicInteger`; 324
- nonblocking algorithm use; 319
- random number generator using; 327$_{li}$

`AtomicLong`; 325

`AtomicReference`; 325
- nonblocking algorithm use; 319
- safe publication use; 52

`AtomicReferenceFieldUpdater`; 335

audit(ing)
- *See also* instrumentation;

audit(ing) tools; 28$_{fn}$

AWT (Abstract Window Toolkit)
- *See also* GUI;
- thread use; 9
  - safety concerns and; 10–11

B

backoff
- and nonblocking algorithms; 231$_{fn}$

barging; **283**
- *See also* fairness; ordering; synchronization;
- and read-write locks; 287
- performance advantages of; 284

barrier(s); **99**, 99–101
- *See also* latch(es); semaphores; synchronizers;
- -based timer; 260–261
- action; **99**
- memory; **230**, **338**
- point; **99**

behavior
- *See also* activities; task(s);

bias
- *See* testing, pitfalls;

bibliography; 355–357

binary latch; **304**
- AQS-based; 313–314

binary semaphore
- mutex use; 99

Bloch, Joshua
- (bibliographic reference); 69

block(ing); **92**
- bounded collections
  - semaphore management of; 99
  - testing; 248
- context switching impact of; 230
- interruptible methods and; 92–94
- interruption handling methods; 138
- methods
  - and interruption; 143
- non-interruptable; 147–150
- operations
  - testing; 250–252
  - thread pool size impact; 170
- queues; 87–94
  - *See also* `Semaphore`;
  - and thread pool management; 173
  - cancellation, problems; 138
  - cancellation, solutions; 140
  - `Executor` functionality combined with; 129
  - producer-consumer pattern and; 87–92
- spin-waiting; **232**
- state-dependent actions; 291–308
  - and polling; 295–296
  - and sleeping; 295–296
  - condition queues; 296–308
  - structure; 292$_{li}$
- threads, costs of; 232
- waits
  - timed vs. unbounded; 170

`BlockingQueue`; 84–85
- and state-based preconditions; 57
- safe publication use; 52
- thread pool use of; 173

bound(ed)
- *See also* constraints; encapsulation;
- blocking collections
  - semaphore management of; 99
- buffers
  - blocking operations; 292
  - scalability testing; 261
  - size determination; 261
- queues
  - and producer-consumer pattern; 88
  - saturation policies; 174–175
  - thread pool use; 172
  - thread pool use of; 173
- resource; **221**

boundaries
- *See also* encapsulation;
- task; **113**
  - analysis for parallelism; 123–133

broken multi-threaded programs
- strategies for fixing; 16

- **`BrokenBarrierException`**
  - parallel iterative algorithm use; 99
- **buffer(s)**
  - *See also* cache/caching;
  - bounded
    - blocking state-dependent operations with; 292
    - scalability testing; 261
    - size determination; 261
  - `BoundedBuffer` example; 249$_{li}$
    - condition queue use; 297
    - test case development for; 248
  - `BoundedBufferTest` example; 250$_{li}$
  - capacities
    - comparison testing; 261–263
  - testing; 248
- **bug pattern(s)**; 271, **271**
  - *See also* debugging; design patterns; testing;
  - detector; **271**
- **busy-waiting**; **295**
  - *See also* spin-waiting;

C

- **cache/caching**
  - *See also* performance;
  - atomicity issues; 24–25
  - flushing
    - and memory barriers; 230
  - implementation issues
    - atomic/atomicity; 106
    - safety; 104
  - misses
    - as cost of context switching; 229
  - result
    - building; 101–109
- **`Callable`**; 126$_{li}$
  - `FutureTask` use; 95
  - results handling capabilities; 125
- **callbacks**
  - testing use; 257–259
- **caller-runs saturation policy**; **174**
- **cancellation**; 135–150
  - *See also* interruption; lifecycle; shutdown;
  - activity; **135**
  - as form of completion; 95
  - `Future` use; 145–147
  - interruptible lock acquisition; 279–281
  - interruption relationship to; 138
  - long-running GUI tasks; 197–198
  - non-standard
    - encapsulation of; 148–150
    - reasons and strategies; 147–150
  - points; **140**
  - policy; **136**
    - and thread interruption policy; 141
    - interruption advantages as implementation strategy; 140
  - reasons for; 136
  - shutdown and; 135–166
  - task
    - `Executor` handling; 125
    - in timed task handling; 131
  - timed locks use; 279
- **`CancellationException`**
  - `Callable` handling; 98
- **CAS (compare-and-swap) instructions**; 321–324
  - *See also* atomic/atomicity, variables;
  - Java class support in Java 5.0; 324
  - lock-free algorithm use; 329
  - nonblocking algorithm use; 319, 329
- **cascading effects**
  - of thread safety requirements; 28
- **cellular automata**
  - barrier use for computation of; 101
- **check-then-act operation**
  - *See also* compound actions;
  - as race condition cause; 21
  - atomic variable handling; 325
  - compound action
    - in collection operations; 79
  - multivariable invariant issues; 67–68
  - service shutdown issue; 153
- **checkpoint**
  - state
    - shutdown issues; 158
- **checksums**
  - safety testing use; 253
- **class(es)**
  - as instance confinement context; 59
  - extension
    - strategies and risks; 71
    - with helper classes; 72–73
  - synchronized wrapper
    - client-side locking support; 73
  - thread-safe
    - and object composition; 55–78

- **cleanup**
  - *See also* lifecycle;
  - and interruption handling
    - protecting data integrity; 142
  - in end-of-lifecycle processing; 135
  - JVM shutdown hooks use for; 164
- **client(s)**
  - *See also* server;
  - requests
    - as natural task boundary; 113
- **client-side locking**; 72–73, **73**
  - *See also* lock(ing);
  - and compound actions; 79–82
  - and condition queues; 306
  - class extension relationship to; 73
  - stream class management; 150$_{fn}$
- **coarsening**
  - *See also* lock(ing);
  - lock; **231**, 235$_{fn}$, 286
- **code review**
  - as quality assurance strategy; 271
- **collections**
  - *See also* hashtables; lists; set(s);
  - bounded blocking
    - semaphore management of; 99
  - concurrent; 84–98
    - building block; 79–110
  - copying
    - as alternative to locking; 83
  - lock striping use; 237
  - synchronized; 79–84
    - concurrent collections vs.; 84
- **`Collections.synchronizedList`**
  - safe publication use; 52
- **`Collections.synchronizedXxx`**
  - synchronized collection creation; 79
- **communication**
  - mechanisms for; 1
- **compare-and-swap (CAS) instructions**
  - *See* CAS;
- **comparison**
  - priority-ordered queue use; 89
- **compilation**
  - dynamic
    - and performance testing; 267–268
  - timing and ordering alterations
    - thread safety risks; 7
- **completion**; **95**
  - *See also* lifecycle;
  - notification
    - of long-running GUI task; 198
  - service
    - `Future`; 129
  - task
    - measuring service time variance; 264–266
  - volatile variable use with; 39
- **`CompletionService`**
  - in page rendering example; 129
- **composition**; **73**
  - *See also* delegation; encapsulation;
  - as robust functionality extension mechanism; 73
  - of objects; 55–78
- **compound actions**; **22**
  - *See also* atomic/atomicity; concurrent/concurrency, collections; race conditions;
  - atomicity handling of; 22–23
  - concurrency design rules role; 110
  - concurrent collection support for; 84
  - examples of
    - *See* check-then-act operation; iteration; navigation; put-if-absent operation; read-modify-write; remove-if-equal operation; replace-if-equal operation;
  - in cache implementation; 106
  - in synchronized collection class use
    - mechanisms for handling; 79–82
  - synchronization requirements; 29
- **computation**
  - compute-intensive code
    - impact on locking behavior; 34
    - thread pool size impact; 170
  - deferred
    - design issues; 125
  - thread-local
    - and performance testing; 268
- ***Concurrent Programming in Java***; 42, 57, 59, 87, 94, 95, 98, 99, 101, 124, 201, 211, 279, 282, 304
- **concurrent/concurrency**
  - *See also* parallelizing/parallelism; safety; synchronization/`synchronized`;
  - and synchronized collections; 84
  - and task independence; 113
  - annotations; 353–354
  - brief history; 1–2

  building blocks; 79–110
  cache implementation issues; 103
  collections; 84–98
  `ConcurrentHashMap` locking strategy advantages; 85
  debugging
    costs vs. performance optimization value; 224
  design rules; 110
  errors
    See deadlock; livelock; race conditions; starvation;
  fine-grained
    and thread-safe data models; 201
  modifying
    synchronized collection problems with; 82
  object pool disadvantages; 241
  poor; **30**
  prevention
    See also single-threaded;
    single-threaded executor use; 172, 177–178
  read-write lock advantages; 286–289
  testing; 247–274
`ConcurrentHashMap`; 84–86
  performance advantages of; 242
`ConcurrentLinkedDeque`; 92
`ConcurrentLinkedQueue`; 84–85
  algorithm; 319–336
  reflection use; 335
  safe publication use; 52
`ConcurrentMap`; 84, 87_{li}
  safe publication use; 52
`ConcurrentModificationException`
  avoiding; 85
  fail-fast iterators use; 82–83
`ConcurrentSkipListMap`; 85
`ConcurrentSkipListSet`; 85
`Condition`; 307_{li}
  explicit condition object use; 306
  intrinsic condition queues vs.
    performance considerations; 308
condition
  predicate; **299**, 299–300
    lock and condition variable relationship; 308
  queues; **297**
    See also synchronizers;
    AQS support for; 312
    blocking state-dependent operations use; 296–308
    explicit; 306–308
    intrinsic; 297
    intrinsic, disadvantages of; 306
    using; 298
  variables
    explicit; 306–308
  waits
    and condition predicate; 299
    canonical form; 301_{li}
    interruptible, as feature of `Condition`; 307
    uninterruptable, as feature of `Condition`; 307
    waking up from, condition queue handling; 300–301
conditional
  See also blocking/blocks;
  notification; **303**
    as optimization; 303
    subclassing safety issues; 304
    use; 304_{li}
  read-modify-writer operations
    atomic variable support for; 325
configuration
  of `ThreadPoolExecutor`; 171–179
  thread creation
    and thread factories; 175
  thread pool
    post-construction manipulation; 177–179
confinement
  See also encapsulation; single-thread(ed);
  instance; **59**, 58–60
  stack; **44**, 44–45
  thread; **42**, 42–46
    ad-hoc; 43
    and execution policy; 167
    in Swing; 191–192
    role, synchronization policy specification; 56
    serial; **90**, 90–92
    single-threaded GUI framework use; 190
    `ThreadLocal`; 45–46
`Connection`
  thread confinement use; 43
  `ThreadLocal` variable use with; 45

consistent/consistency
copy timeliness vs.
as design tradeoff; 62
data view timeliness vs.
as design tradeoff; 66, 70
lock ordering
and deadlock avoidance; 206
weakly consistent iterators; **85**
constraints
See also invariant(s); post-conditions; pre-conditions;
state transition; **56**
thread creation
importance of; 116
construction/constructors
See also lifecycle;
object
publication risks; 41–42
thread handling issues; 41–42
partial
unsafe publication influence; 50
private constructor capture idiom; 69_{fn}
starting thread from
as concurrency bug pattern; 272
`ThreadPoolExecutor`; 172_{li}
post-construction customization; 177
consumers
See also blocking, queues; producer-consumer pattern;
blocking queues use; 88
producer-consumer pattern
blocking queues and; 87–92
containers
See also collections;
blocking queues as; 94
scoped
thread safety concerns; 10
contention/contended
as performance inhibiting factor; 263
intrinsic locks vs. `ReentrantLock`
performance considerations; 282–286
lock
costs of; 320
measurement; 240–241
reduction impact; 211
reduction, strategies; 232–242
scalability impact; 232
`signal` method reduction in; 308
locking vs. atomic variables; 328
resource
and task execution policy; 119
deque advantages; 92
scalability under
as AQS advantage; 311
scope
atomic variable limitation of; 324
synchronization; **230**
thread
collision detection help with; 321
latches help with; 95
throughput impact; 228
unrealistic degrees of
as performance testing pitfall; 268–269
context switching; 229
See also performance;
as cost of thread use; 229–230
condition queues advantages; 297
cost(s); 8
message logging
reduction strategies; 243–244
performance impact of; 221
reduction; 243–244
`signal` method reduction in; 308
throughput impact; 228
control flow
See also event(s); lifecycle; MVC (model-view-controller) pattern;
coordination
in producer-consumer pattern; 94
event handling
model-view objects; 195_{fg}
simple; 194_{fg}
latch characteristics; 94
model-view-controller pattern
and inconsistent lock ordering; 190
vehicle tracking example; 61
convenience
See also responsiveness;
as concurrency motivation; 2
conventions
annotations
concurrency documentation; 6
Java monitor pattern; 61

cooperation/cooperating
  See also concurrent/concurrency; synchronization;
  end-of-lifecycle mechanisms
    interruption as; 93, 135
  model, view, and controller objects in GUI applications
    inconsistent lock ordering; 190
  objects
    deadlock, lock-ordering; 212$_{li}$
    deadlock, possibilities; 211
    livelock possibilities; 218
  thread
    concurrency mechanisms for; 79
coordination
  See also synchronization/synchronized;
  control flow
    producer-consumer pattern, blocking queues use; 94
  in multithreaded environments
    performance impact of; 221
  mutable state access
    importance of; 110
copying
  collections
    as alternative to locking; 83
  data
    thread safety consequences; 62
CopyOnWriteArrayList; 84, 86–87
  safe publication use; 52
  versioned data model use
    in GUI applications; 201
CopyOnWriteArraySet
  safe publication use; 52
  synchronized Set replacement; 86
core pool size parameter
  thread creation impact; **171**, 172$_{fn}$
correctly synchronized program; **341**
correctness; **17**
  See also safety;
  testing; 248–260
    goals; 247
  thread safety defined in terms of; 17
corruption
  See also atomic/atomicity; encapsulation; safety; state;
  data
    and interruption handling; 142
    causes, stale data; 35
cost(s)
  See also guidelines; performance; safety; strategies; tradeoffs;
  thread; 229–232
    context switching; 8
    locality loss; 8
  tradeoffs
    in performance optimization strategies; 223
CountDownLatch; 95
  AQS use; 315–316
  puzzle-solving framework use; 184
  TestHarness example use; 96
counting semaphores; **98**
  See also Semaphore;
  permits, thread relationships; 248
  SemaphoreOnLock example; 310$_{li}$
coupling
  See also dependencies;
  behavior
    blocking queue handling; 89
  implicit
    between tasks and execution policies; 167–170
CPU utilization
  See also performance;
  and sequential execution; 124
  condition queues advantages; 297
  impact on performance testing; 261
  monitoring; 240–241
  optimization
    as multithreading goal; 222
  spin-waiting impact on; 295
creation
  See also copying; design; policy(s); representation;
  atomic compound actions; 80
  class
    existing thread-safe class reuse advantages over; 71
  collection copy
    as immutable object strategy; 86
  of immutable objects; 48
  of state-dependent methods; 57
  synchronizer; 94
  thread; 171–172
    explicitly, for tasks; 115
    thread factory use; 175–177
    unbounded, disadvantages; 116
  thread pools; 120
  wrappers

during memoization; 103

customization

thread configuration

ThreadFactory use; 175

thread pool configuration

post-construction; 177–179

CyclicBarrier; 99

parallel iterative algorithm use; 102_{li}

testing use; 255_{li}, 260_{li}

D

daemon threads; 165

data

See also state;

contention avoidance

and scalability; 237

hiding

thread-safety use; 16

nonatomic

64-bit operations; 36

sharing; 33–54

See also page renderer examples;

access coordination; 277–290, 319

advantages of threads; 2

shared data models; 198–202

synchronization costs; 8

split data models; **201**, 201–202

stale; 35–36

versioned data model; **201**

data race; **341**

race condition vs.; 20_{fn}

data structure(s)

See also collections; object(s); queue(s); stack(s); trees;

handling

See atomic/atomicity; confinement; encapsulation; iterators/iteration; recursion;

protection

and interruption handling; 142

shared

as serialization source; 226

testing insertion and removal handling; 248

database(s)

deadlock recovery capabilities; 206

JDBC Connection

thread confinement use; 43

thread pool size impact; 171

Date

effectively immutable use; 53

dead-code elimination

and performance testing; 269–270

deadline-based waits

as feature of Condition; 307

deadlock(s); **205**, 205–217

See also concurrent/concurrency, errors; liveness; safety;

analysis

thread dump use; 216–217

as liveness failure; 8

avoidance

and thread confinement; 43_{fn}

nonblocking algorithm advantages; 319, 329

strategies for; 215–217

cooperating objects; 211

diagnosis

strategies for; 215–217

dynamic lock order; 207–210

in GUI framework; 190

lock splitting as risk factor for; 235

locking during iteration risk of; 83

recovery

database capabilities; 206

polled and timed lock acquisition use; 279, 280

timed locks use; 215

reentrancy avoidance of; 27

resource; 213–215

thread starvation; **169**, 168–169, **215**

deadly embrace

See deadlock;

death, thread

abnormal, handling; 161–163

debugging

See also analysis; design; documentation; recovery; testing;

annotation use; 353

concurrency

costs vs. performance optimization value; 224

custom thread factory as aid for; 175

JVM optimization pitfalls; 38_{fn}

thread dump use; 216_{fn}

thread dumps

intrinsic lock advantage over ReentrantLock; 285–286

unbounded thread creation risks; 116

decomposition
- *See also* composition; delegation; encapsulation;
- producer-consumer pattern; 89
- tasks-related; 113–134

Decorator pattern
- collection class use for wrapper factories; 60

decoupling
- of activities
  - as producer-consumer pattern advantage; 87
  - task decomposition as representation of; 113
- of interrupt notification from handling in `Thread` interruption handling methods; 140
- task submission from execution
  - and `Executor` framework; 117

delayed tasks
- *See also* time/timing;
- handling of; 123

`DelayQueue`
- time management; 123

delegation
- *See also* composition; design; safety;
- advantages
  - class extension vs.; 314
  - for class maintenance safety; 234
- thread safety; **234**
  - failure causes; 67–68
  - management; 62

dependencies
- *See also* atomic/atomicity; invariant(s); postconditions; preconditions; state;
- code
  - as removal, as producer-consumer pattern advantage; 87
- in multiple-variable invariants
  - thread safety issues; 24
- state
  - blocking operations; 291–308
  - classes; **291**
  - classes, building; 291–318
  - managing; 291–298
  - operations; **57**
  - operations, condition queue handling; 296–308
  - task freedom from, importance of; 113
- task
  - and execution policy; 167
  - thread starvation deadlock; 168
- task freedom from
  - importance; 113

`Deque`; 92

deques
- *See also* collections; data structure(s); queue(s);
- work stealing and; 92

design
- *See also* documentation; guidelines; policies; representation; strategies;
- class
  - state ownership as element of; 57–58
- concurrency design rules; 110
- concurrency testing; 250–252
- condition queue encapsulation; 306
- condition queues
  - and condition predicate; 299
- control flow
  - latch characteristics; 94
- execution policy
  - influencing factors; 167
- GUI single-threaded use
  - rationale for; 189–190
- importance
  - in thread-safe programs; 16
- of thread-safe classes
  - guidelines; 55–58
- parallelism
  - application analysis for; 123–133
- parallelization criteria; 181
- performance
  - analysis, monitoring, and improvement; 221–245
- performance tradeoffs
  - evaluation of; 223–225
- principles
  - simplicity of final fields; 48
- producer-consumer pattern
  - decoupling advantages; 117
  - `Executor` framework use; 117
- program
  - and task decomposition; 113–134
- result-bearing tasks
  - representation issues; 125

- strategies
  - for `InterruptedException`; 93
- thread confinement; 43
- thread pool size
  - relevant factors for; 170
- timed tasks; 131–133
- tradeoffs
  - collection copying vs. locking during iteration; 83
  - concurrent vs. synchronized collections; 85
  - copy-on-write collections; 87
  - synchronized block; 34
  - timeliness vs. consistency; 62, 66, 70

design patterns
- antipattern example
  - double-checked locking; 348–349
- examples
  - *See* Decorator pattern; MVC (model-view-controller) pattern; producer-consumer pattern; Singleton pattern;

destruction
- *See* teardown;

dining philosophers problem; 205
- *See also* deadlock;

discard saturation policy; **174**

discard-oldest saturation policy; **174**

documentation
- *See also* debugging; design; good practices; guidelines; policy(s);
- annotation use; 6, 353
- concurrency design rules role; 110
- critical importance for conditional notification use; 304
- importance
  - for special execution policy requirements; 168
  - stack confinement usage; 45
- of synchronization policies; 74–77
- safe publication requirements; 54

double-checked locking (DCL); 348–349
- as concurrency bug pattern; 272

downgrading
- read-write lock implementation strategy; 287

driver program
- for `TimedPutTakeTest` example; 262

dynamic
- *See also* responsiveness;
- compilation
  - as performance testing pitfall; 267–268
- lock order deadlocks; 207–210

E

EDT (event dispatch thread)
- GUI frameworks use; 5
- single-threaded GUI use; **189**
- thread confinement use; 42

Effective Java Programming Language Guide; 46–48, 73, 166, 257, 292, 305, 314, 347

efficiency
- *See also* performance;
- responsiveness vs.
  - polling frequency; 143
- result cache, building; 101–109

elision
- lock; 231_{fn}
  - JVM optimization; 286

encapsulation
- *See also* access; atomic/atomicity; confinement; safety; state; visibility;
- breaking
  - costs of; 16–17
- code
  - as producer-consumer pattern advantage; 87
- composition use; 74
- concurrency design rules role; 110
- implementation
  - class extension violation of; 71
- instance confinement relationship with; 58–60
- invariant management with; 44
- locking behavior
  - reentrancy facilitation of; 27
- non-standard cancellation; 148–150
- of condition queues; 306
- of lifecycle methods; 155
- of synchronization
  - hidden iterator management through; 83
- publication dangers for; 39
- state

- - breaking, costs of; 16–17
  - invariant protection use; 83
  - ownership relationship with; 58
  - synchronizer role; 94
  - thread-safe class use; 23
- synchronization policy
  - and client-side locking; 71
- thread ownership; 150
- thread-safety role; 55
- thread-safety use; 16

end-of-lifecycle
- *See also* thread(s);
- management techniques; 135–166

enforcement
- locking policies, lack of; 28

entry protocols
- state-dependent operations; 306

`Error`
- `Callable` handling; 97

error(s)
- as cancellation reason; 136
- concurrency
  - *See* deadlock; livelock; race conditions;

escape; **39**
- analysis; **230**
- prevention
  - in instance confinement; 59
- publication and; 39–42
- risk factors
  - in instance confinement; 60

Ethernet protocol
- exponential backoff use; 219

evaluation
- *See also* design; measurement; testing;
- of performance tradeoffs; 223–225

event(s); **191**
- as cancellation reason; 136
- dispatch thread
  - GUI frameworks use; 5
- handling
  - control flow, simple; 194_{fg}
  - model-view objects; 195_{fg}
  - threads benefits for; 4
- latch handling based on; 99
- main event loop
  - vs. event dispatch thread; 5
- notification
  - copy-on-write collection advantages; 87
- sequential processing
  - in GUI applications; 191
- timing
  - and liveness failures; 8

example classes
- `AtomicPseudoRandom`; 327_{li}
- `AttributeStore`; 233_{li}
- `BackgroundTask`; 199_{li}
- `BarrierTimer`; 261_{li}
- `BaseBoundedBuffer`; 293_{li}
- `BetterAttributeStore`; 234_{li}
- `BetterVector`; 72_{li}
- `Big`; 258_{li}
- `BoundedBuffer`; 248, 249_{li}, 297, 298_{li}
- `BoundedBufferTest`; 250_{li}
- `BoundedExecutor`; 175
- `BoundedHashSet`; 100_{li}
- `BrokenPrimeProducer`; 139_{li}
- `CachedFactorizer`; 31_{li}
- `CancellableTask`; 151_{li}
- `CasCounter`; 323_{li}
- `CasNumberRange`; 326_{li}
- `CellularAutomata`; 102_{li}
- `Computable`; 103_{li}
- `ConcurrentPuzzleSolver`; 186_{li}
- `ConcurrentStack`; 331_{li}
- `ConditionBoundedBuffer`; 308, 309_{li}
- `Consumer`; 256_{li}
- `Counter`; 56_{li}
- `CountingFactorizer`; 23_{li}
- `CrawlerThread`; 157_{li}
- `DelegatingVehicleTracker`; 65_{li}, 201
- `DemonstrateDeadlock`; 210_{li}
- `Dispatcher`; 212_{li}, 214_{li}
- `DoubleCheckedLocking`; 349_{li}
- `ExpensiveFunction`; 103_{li}
- `Factorizer`; 109_{li}
- `FileCrawler`; 91_{li}
- `FutureRenderer`; 128_{li}
- `GrumpyBoundedBuffer`; 292, 294_{li}
- `GuiExecutor`; 192, 194_{li}
- `HiddenIterator`; 84_{li}
- `ImprovedList`; 74_{li}
- `Indexer`; 91_{li}
- `IndexerThread`; 157_{li}
- `IndexingService`; 156_{li}
- `LazyInitRace`; 21_{li}
- `LeftRightDeadlock`; 207_{li}
- `LifecycleWebServer`; 122_{li}
- `LinkedQueue`; 334_{li}

ListHelper; 73, 74_{li}
LogService; 153, 154_{li}
LogWriter; 152_{li}
Memoizer; 103_{li}, 108_{li}
Memoizer2; 104_{li}
Memoizer3; 106_{li}
MonitorVehicleTracker; 63_{li}
MutableInteger; 36_{li}
MutablePoint; 64_{li}
MyAppThread; 177, 178_{li}
MyThreadFactory; 177_{li}
Node; 184_{li}
NoVisibility; 34_{li}
NumberRange; 67_{li}
OneShotLatch; 313_{li}
OneValueCache; 49_{li}, 51_{li}
OutOfTime; 124_{li}, 161
PersonSet; 59_{li}
Point; 64_{li}
PossibleReordering; 340_{li}
Preloader; 97_{li}
PrimeGenerator; 137_{li}
PrimeProducer; 141_{li}
PrivateLock; 61_{li}
Producer; 256_{li}
PutTakeTest; 255_{li}, 260
Puzzle; 183_{li}
PuzzleSolver; 188_{li}
QueueingFuture; 129_{li}
ReaderThread; 149_{li}
ReadWriteMap; 288_{li}
ReentrantLockPseudoRandom; 327_{li}
Renderer; 130_{li}
SafeListener; 42_{li}
SafePoint; 69_{li}
SafeStates; 350_{li}
ScheduledExecutorService; 145_{li}
SemaphoreOnLock; 310_{li}
Sequence; 7_{li}
SequentialPuzzleSolver; 185_{li}
ServerStatus; 236_{li}
SimulatedCAS; 322_{li}
SingleThreadRenderer; 125_{li}
SingleThreadWebServer; 114_{li}
SleepyBoundedBuffer; 295, 296_{li}
SocketUsingTask; 151_{li}
SolverTask; 186_{li}
StatelessFactorizer; 18_{li}
StripedMap; 238_{li}
SwingUtilities; 191, 192, 193_{li}
Sync; 343_{li}
SynchronizedFactorizer; 26_{li}
SynchronizedInteger; 36_{li}
TaskExecutionWebServer; 118_{li}
TaskRunnable; 94_{li}
Taxi; 212_{li}, 214_{li}
TestHarness; 96_{li}
TestingThreadFactory; 258_{li}
ThisEscape; 41_{li}
ThreadDeadlock; 169_{li}
ThreadGate; 305_{li}
ThreadPerTaskExecutor; 118_{li}
ThreadPerTaskWebServer; 115_{li}
ThreeStooges; 47_{li}
TimedPutTakeTest; 261
TimingThreadPool; 180_{li}
TrackingExecutorService; 159_{li}
UEHLogger; 163_{li}
UnsafeCachingFactorizer; 24_{li}
UnsafeCountingFactorizer; 19_{li}
UnsafeLazyInitialization; 345_{li}
UnsafeStates; 40_{li}
ValueLatch; 184, 187_{li}
VisualComponent; 66_{li}
VolatileCachedFactorizer; 50_{li}
WebCrawler; 160_{li}
Widget; 27_{li}
WithinThreadExecutor; 119_{li}
WorkerThread; 227_{li}

exceptions
See also error(s); interruption; lifecycle;
and precondition failure; 292–295
as form of completion; 95
Callable handling; 97
causes
stale data; 35
handling
Runnable limitations; 125
logging
UEHLogger example; 163_{li}
thread-safe class handling; 82
Timer disadvantages; 123
uncaught exception handler; 162–163
unchecked
catching, disadvantages; 161

Exchanger
See also producer-consumer pattern;
as two-party barrier; 101
safe publication use; 53

`execute`
- `submit` vs., uncaught exception handling; 163

execution
- policies
  - design, influencing factors; 167
  - `Executors` factory methods; 171
  - implicit couplings between tasks and; 167–170
  - parallelism analysis for; 123–133
- task; 113–134
  - policies; 118–119
  - sequential; 114

`ExecutionException`
- `Callable` handling; 98

`Executor` framework; 117_{li}, 117–133
- and GUI event processing; 191, 192
- and long-running GUI tasks; 195
- as producer-consumer pattern; 88
- execution policy design; 167
- `FutureTask` use; 97
- `GuiExecutor` example; 194_{li}
- single-threaded
  - deadlock example; 169_{li}

`ExecutorCompletionService`
- in page rendering example; 129

`Executors`
- factory methods
  - thread pool creation with; 120

`ExecutorService`
- and service shutdown; 153–155
- cancellation strategy using; 146
- checkMail example; 158
- lifecycle methods; 121_{li}, 121–122

exhaustion
- *See* failure; leakage; resource exhaustion;

exit protocols
- state-dependent operations; 306

explicit locks; 277–290
- interruption during acquisition; 148

exponential backoff
- and avoiding livelock; 219

extending
- existing thread-safe classes
  - and client-side locking; 73
  - strategies and risks; 71
- `ThreadPoolExecutor`; 179

external locking; 73

F

factory(s)
- *See also* creation;
- methods
  - constructor use with; 42
  - `newTaskFor`; 148
  - synchronized collections; 79, 171
  - thread pool creation with; 120
- thread; **175**, 175–177

fail-fast iterators; **82**
- *See also* iteration/iterators;

failure
- *See also* exceptions; liveness, failure; recovery; safety;
- causes
  - stale data; 35
- graceful degradation
  - task design importance; 113
- management techniques; 135–166
- modes
  - testing for; 247–274
- precondition
  - bounded buffer handling of; 292
  - propagation to callers; 292–295
- thread
  - uncaught exception handlers; 162–163
- timeout
  - deadlock detection use; 215

fairness
- *See also* responsiveness; synchronization;
- as concurrency motivation; 1
- fair lock; **283**
- nonfair lock; **283**
- nonfair semaphores vs. fair
  - performance measurement; 265
- queuing
  - intrinsic condition queues; 297_{fn}
- `ReentrantLock` options; 283–285
- `ReentrantReadWriteLock`; 287
- scheduling
  - thread priority manipulation risks; 218

'fast path' synchronization
- CAS-based operations vs.; 324
- costs of; 230

feedback
- *See also* GUI;
- user
  - in long-running GUI tasks; 196_{li}

fields
- atomic updaters; 335–336
- hot fields
  - avoiding; 237
  - updating, atomic variable advantages; 239–240
- initialization safety
  - final field guarantees; 48

FIFO queues
- `BlockingQueue` implementations; 89

files
- *See also* data; database(s);
- as communication mechanism; 1

`final`
- and immutable objects; 48
- concurrency design rules role; 110
- immutability not guaranteed by; 47
- safe publication use; 52
- `volatile` vs.; 158_{fn}

finalizers
- JVM orderly shutdown use; 164
- warnings; 165–166

`finally` block
- *See also* interruptions; lock(ing);
- importance with explicit locks; 278

FindBugs code auditing tool
- *See also* tools;
- as static analysis tool example; 271
- locking failures detected by; 28_{fn}
- unreleased lock detector; 278_{fn}

fire-and-forget event handling strategy
- drawbacks of; 195

flag(s)
- *See* mutex;
- cancellation request
  - as cancellation mechanism; 136
- interrupted status; **138**

flexibility
- *See also* responsiveness; scalability;
- and instance confinement; 60
- decoupling task submission from execution, advantages for; 119
- immutable object design for; 47
- in CAS-based algorithms; 322
- interruption policy; 142
- resource management
  - as blocking queue advantage; 88
  - task design guidelines for; 113
- task design role; 113

flow control
- communication networks, thread pool comparison; 173_{fn}

fragility
- *See also* debugging; guidelines; robustness; safety; scalability; testing;
- issues and causes
  - as class extension; 71
  - as client-side locking; 73
  - interruption use for non-standard purposes; 138
  - issue; 43
  - piggybacking; 342
  - state-dependent classes; 304
  - volatile variables; 38
- solutions
  - composition; 73
  - encapsulation; 17
  - stack confinement vs. ad-hoc thread confinement; 44

frameworks
- *See also* AQS framework; data structure(s); `Executor` framework; RMI framework; Servlets framework;
- application
  - and `ThreadLocal`; 46
- serialization hidden in; 227
- thread use; 9
- thread use impact on applications; 9
- threads benefits for; 4

functionality
- extending for existing thread-safe classes
  - strategies and risks; 71
- tests
  - vs. performance tests; 260

`Future`; 126_{li}
- cancellation
  - of long-running GUI task; 197
  - strategy using; 145–147
- characteristics of; 95
- encapsulation of non-standard cancellation use; 148
- results handling capabilities; 125
- safe publication use; 53
- task lifecycle representation by; 125

- task representation
  - implementation strategies; 126
- **`FutureTask`**; 95
  - AQS use; 316
  - as latch; 95–98
  - completion notification
    - of long-running GUI task; 198
  - efficient and scalable cache implementation with; 105
  - example use; 97_{li}, 108_{li}, 151_{li}, 199_{li}
  - task representation use; 126

G

- **garbage collection**
  - as performance testing pitfall; 266
- **gate**
  - *See also* barrier(s); conditional; latch(es);
  - as latch role; 94
  - `ThreadGate` example; 304
- **global variables**
  - ThreadLocal variables use with; 45
- **good practices**
  - *See* design; documentation; encapsulation; guidelines; performance; strategies;
- **graceful**
  - degradation
    - and execution policy; 121
    - and saturation policy; 175
    - limiting task count; 119
    - task design importance; 113
  - shutdown
    - vs. abrupt shutdown; 153
- **granularity**
  - *See also* atomic/atomicity; scope;
  - atomic variable advantages; 239–240
  - lock
    - Amdahl's law insights; 229
    - reduction of; 235–237
  - nonblocking algorithm advantages; 319
  - serialization
    - throughput impact; 228
  - timer
    - measurement impact; 264
- **guarded**
  - objects; **28**, **54**
  - state
    - locks use for; 27–29
- **`@GuardedBy`**; 353–354
  - and documenting synchronization policy; 7_{fn}, 75
- **GUI (Graphical User Interface)**
  - *See also* event(s); single-thread(ed); Swing;
  - applications; 189–202
    - thread safety concerns; 10–11
  - frameworks
    - as single-threaded task execution example; 114_{fn}
  - long-running task handling; 195–198
  - MVC pattern use
    - in vehicle tracking example; 61
  - response-time sensitivity
    - and execution policy; 168
  - single-threaded use
    - rationale for; 189–190
  - threads benefits for; 5
- **guidelines**
  - *See also* design; documentation; policy(s); strategies;
  - allocation vs. synchronization; 242
  - atomicity
    - definitions; 22
  - concurrency design rules; 110
  - `Condition` methods
    - potential confusions; 307
  - condition predicate
    - documentation; 299
    - lock and condition queue relationship; 300
  - condition wait usage; 301
  - confinement; 60
  - deadlock avoidance
    - alien method risks; 211
    - lock ordering; 206
    - open call advantages; 213
    - thread starvation; 169
  - documentation
    - value for safety; 16
  - encapsulation; 59, 83
    - value for safety; 16
  - exception handling; 163
  - execution policy
    - design; 119
    - special case implications; 168
  - final field use; 48
  - finalizer precautions; 166
  - happens-before use; 346
  - immutability

effectively immutable objects; **53**
objects; 46
requirements for; 47
value for safety; 16
initialization safety; 349, 350
interleaving diagrams; 6
interruption handling
cancellation relationship; 138
importance of interruption policy knowledge; 142, 145
interrupt swallowing precautions; 143
intrinsic locks vs. `ReentrantLock`; 285
invariants
locking requirements for; 29
thread safety importance; 57
value for safety; 16
lock
contention, reduction; 233
contention, scalability impact; 231
holding; 32
ordering, deadlock avoidance; 206
measurement
importance; 224
notification; 303
objects
stateless, thread-safety of; 19
operation ordering
synchronization role; 35
optimization
lock contention impact; 231
premature, avoidance of; 223
parallelism analysis; 123–133
performance
optimization questions; 224
simplicity vs.; 32
postconditions; 57
private field use; 48
publication; 52, 54
safety
definition; 18
testing; 252
scalability; 84
attributes; 222
locking impact on; 232
sequential loops
parallelization criteria; 181
serialization sources; 227
sharing
safety strategies; 16
sharing objects; 54
simplicity
performance vs.; 32
starvation avoidance
thread priority precautions; 218
state
consistency preservation; 25
managing; 23
variables, independent; 68
stateless objects
thread-safety of; 19
synchronization
immutable objects as replacement for; 52
shared state requirements for; 28
task cancellation
criteria for; 147
testing
effective performance tests; 270
timing-sensitive data races; 254
`this` reference
publication risks; 41
threads
daemon thread precautions; 165
handling encapsulation; 150
lifecycle methods; 150
pools; 174
safety; 18, 55
volatile variables; 38

H

hand-over-hand locking; **282**
happens-before
JMM definition; 340–342
piggybacking; 342–344
publication consequences; 244–249
hardware
See also CPU utilization;
concurrency support; 321–324
JVM interaction
reordering; 34
platform memory models; 338
timing and ordering alterations by
thread safety risks; 7
hashcodes/hashtables
See also collections;
`ConcurrentHashMap`; 84–86
performance advantages of; 242
`Hashtable`; 79

    - safe publication use; 52
  - inducing lock ordering with; 208
  - lock striping use; 237

heap inspection tools
- *See also* tools;
- measuring memory usage; 257

Heisenbugs; 247$_{fn}$

helper classes
- and extending class functionality; 72–73

heterogeneous tasks
- parallelization limitations; 127–129

hijacked signal
- *See* missed signals;

Hoare, C. A. R.
- Java monitor pattern inspired by (bibliographic reference); 60$_{fn}$

hoisting
- variables
  - as JVM optimization pitfall; 38$_{fn}$

homogeneous tasks
- parallelism advantages; 129

hooks
- *See also* extending;
- completion
  - in `FutureTask`; 198
- shutdown; **164**
  - JVM orderly shutdown; 164–165
- single shutdown
  - orderly shutdown strategy; 164
- `ThreadPoolExecutor` extension; 179

hot fields
- avoidance
  - scalability advantages; 237
- updating
  - atomic variable advantages; 239–240

HotSpot JVM
- dynamic compilation use; 267

'how fast'; 222
- *See also* GUI; latency; responsiveness;
- vs. 'how much'; 222

'how much'; 222
- *See also* capacity; scalability; throughput;
- importance for server applications; 223
- vs. 'how fast'; 222

`HttpSession`
- thread-safety requirements; 58$_{fn}$

I

I/O
- *See also* resource(s);
- asynchronous
  - non-interruptable blocking; 148
- message logging
  - reduction strategies; 243–244
- operations
  - thread pool size impact; 170
- sequential execution limitations; 124
- server applications
  - task execution implications; 114
- synchronous
  - non-interruptable blocking; 148
  - threads use to simulate; 4
- utilization measurement tools; 240

idempotence
- and race condition mitigation; 161

idioms
- *See also* algorithm(s); conventions; design patterns; documentation; policy(s); protocols; strategies;
- double-checked locking (DCL)
  - as bad practice; 348–349
- lazy initialization holder class; 347–348
- private constructor capture; 69$_{fn}$
- safe initialization; 346–348
- safe publication; 52–53

`IllegalStateException`
- `Callable` handling; 98

`@Immutable`; 7, 353

immutable/immutability; 46–49
- *See also* atomic/atomicity; safety;
- concurrency design rules role; 110
- effectively immutable objects; **53**
- initialization safety guarantees; 51
- initialization safety limitation; 350
- objects; **46**
  - publication with volatile; 48–49
  - requirements for; 47
- role in synchronization policy; 56
- thread-safety use; 16

implicit coupling
- *See also* dependencies;
- between tasks and execution policies; 167–170

- **improper publication**; **51**
  - *See also* safety;
- **increment operation (++)**
  - as non-atomic operation; 19
- **independent/independence**; **25**
  - *See also* dependencies; encapsulation; invariant(s); state;
  - multiple-variable invariant lack of thread safety issues; 24
  - parallelization use; 183
  - state variables; **66**, 66–67
    - lock splitting use with; 235
  - task
    - concurrency advantages; 113
- **inducing lock ordering**
  - for deadlock avoidance; 208–210
- **initialization**
  - *See also* construction/constructors;
  - lazy; **21**
    - as race condition cause; 21–22
    - safe idiom for; 348$_{li}$
    - unsafe publication risks; 345
  - safety
    - and immutable objects; 51
    - final field guarantees; 48
    - idioms for; 346–348
    - JMM support; 349–350
- **inner classes**
  - publication risks; 41
- **instance confinement**; **59**, 58–60
  - *See also* confinement; encapsulation;
- **instrumentation**
  - *See also* analysis; logging; monitoring; resource(s), management; statistics; testing;
  - of thread creation
    - thread pool testing use; 258
  - potential
    - as execution policy advantage; 121
  - service shutdown use; 158
  - support
    - `Executor` framework use; 117
  - thread pool size requirements determination use of; 170
  - `ThreadPoolExecutor` hooks for; 179
- **interfaces**
  - user
    - threads benefits for; 5
- **interleaving**
  - diagram interpretations; 6
  - generating
    - testing use; 259
  - logging output
    - and client-side locking; 150$_{fn}$
  - operation; 81$_{fg}$
  - ordering impact; 339
  - thread
    - dangers of; 5–8
    - timing dependencies impact on race conditions; 20
  - thread execution
    - in thread safety definition; 18
- **`interrupted (Thread)`**
  - usage precautions; 140
- **`InterruptedException`**
  - flexible interruption policy advantages; 142
  - interruption API; 138
  - propagation of; 143$_{li}$
  - strategies for handling; 93
  - task cancellation
    - criteria for; 147
- **interruption(s)**; **93**, **135**, 138–150
  - *See also* completion; errors; lifecycle; notification; termination; triggering;
  - and condition waits; 307
  - blocking and; 92–94
  - blocking test use; 251
  - interruption response strategy
    - exception propagation; 142
    - status restoration; 142
  - lock acquisition use; 279–281
  - non-cancellation uses for; 143
  - non-interruptable blocking
    - handling; 147–150
    - reasons for; 148
  - policies; **141**, 141–142
  - preemptive
    - deprecation reasons; 135$_{fn}$
  - request
    - strategies for handling; 140
  - responding to; 142–150
  - swallowing
    - as discouraged practice; 93
    - bad consequences of; 140
    - when permitted; 143
  - thread; **138**
  - volatile variable use with; 39

intransitivity
encapsulation characterized by; 150
intrinsic condition queues; 297
disadvantages of; 306
intrinsic locks; **25**, 25–26
See also encapsulation; lock(ing); safety; synchronization; thread(s);
acquisition, non-interruptable blocking reason; 148
advantages of; 285
explicit locks vs.; 277–278
intrinsic condition queue relationship to; 297
limitations of; 28
recursion use; 237*fn*
`ReentrantLock` vs.; 282–286
visibility management with; 36
invariant(s)
See also atomic/atomicity; post-conditions; pre-conditions; state;
and state variable publication; 68
`BoundedBuffer` example; 250
callback testing; 257
concurrency design rules role; 110
encapsulation
state, protection of; 83
value for; 44
immutable object use; 49
independent state variables requirements; 66–67
multivariable
and atomic variables; 325–326
atomicity requirements; 57, 67–68
locking requirements for; 29
preservation of, as thread safety requirement; 24
thread safety issues; 24
preservation of
immutable object use; 46
mechanisms and synchronization policy role; 55–56
publication dangers for; 39
specification of
thread-safety use; 16
thread safety role; 17
`iostat` application
See also measurement; tools;
I/O measurement; 240
iterators/iteration
See also concurrent/concurrency; control flow; recursion;
as compound action
in collection operations; 79
atomicity requirements during; 80
fail-fast; **82**
`ConcurrentModificationException` exception with; 82–83
hidden; 83–84
locking
concurrent collection elimination of need for; 85
disadvantages of; 83
parallel iterative algorithms
barrier management of; 99
parallelization of; 181
unreliable
and client-side locking; 81
weakly consistent; **85**

J

Java Language Specification, The; 53, 218*fn*, 259, 358
Java Memory Model (JMM); 337–352
See also design; safety; synchronization; visibility;
initialization safety guarantees for immutable objects; 51
Java monitor pattern; **60**, 60–61
composition use; 74
vehicle tracking example; 61–71
Java Programming Language, The; 346
`java.nio` package
synchronous I/O
non-interruptable blocking; 148
JDBC (Java Database Connectivity)
`Connection`
thread confinement use; 43
JMM (Java Memory Model)
See Java Memory Model (JMM);
`join` (Thread)
timed
problems with; 145
JSPs (JavaServer Pages)
thread safety requirements; 10
JVM (Java Virtual Machine)
See also optimization;
deadlock handling limitations; 206
escape analysis; 230–231
lock contention handling; 320*fn*

  nonblocking algorithm use; 319
  optimization pitfalls; 38_{fn}
  optimizations; 286
  service shutdown issues; 152–153
  shutdown; 164–166
    and daemon threads; 165
    orderly shutdown; 164
  synchronization optimization by; 230
  thread timeout interaction
    and core pool size; 172_{fn}
  thread use; 9
  uncaught exception handling; 162_{fn}

K

keep-alive time
  thread termination impact; 172

L

latch(es); **94**, 94–95
  See also barriers; blocking; semaphores; synchronizers;
  barriers vs.; 99
  binary; **304**
    AQS-based; 313–314
  `FutureTask`; 95–98
  puzzle-solving framework use; 184
  `ThreadGate` example; 304
layering
  three-tier application
    as performance vs. scalability illustration; 223
lazy initialization; **21**
  as race condition cause; 21–22
  safe idiom for; 348_{li}
  unsafe publication risks; 345
leakage
  See also performance;
  resource
    testing for; 257
  thread; **161**
    `Timer` problems with; 123
    `UncaughtExceptionHandler`
      prevention of; 162–163
lexical scope
  as instance confinement context; 59
library
  thread-safe collections
    safe publication guarantees; 52
Life cellular automata game
  barrier use for computation of; 101
lifecycle
  See also cancellation; completion; construction/constructors; `Executor`; interruption; shutdown; termination; thread(s); time/timing;
  encapsulation; 155
  `Executor`
    implementations; 121–122
  management strategies; 135–166
  support
    `Executor` framework use; 117
  task
    and `Future`; 125
    `Executor` phases; 125
  thread
    performance impact; 116
    thread-based service management; 150
lightweight processes
  See threads;
linked lists
  `LinkedBlockingDeque`; 92
  `LinkedBlockingQueue`; 89
    performance advantages; 263
    thread pool use of; 173–174
  `LinkedList`; 85
  Michael-Scott nonblocking queue; 332–335
  nonblocking; 330
`List`
  `CopyOnWriteArrayList` as concurrent collection for; 84, 86
listeners
  See also event(s);
  action; 195–197
  Swing
    single-thread rule exceptions; 190
  Swing event handling; 194
lists
  See also collections;
  `CopyOnWriteArrayList`
    safe publication use; 52
    versioned data model use; 201
  `LinkedList`; 85
  `List`
    `CopyOnWriteArrayList` as concurrent replacement; 84, 86

Little's law
- lock contention corollary; 232$_{fn}$

livelock; 219, **219**
- *See also* concurrent/concurrency, errors; liveness;
- as liveness failure; 8

liveness
- *See also* performance; responsiveness failure;
- causes
  - *See* deadlock; livelock; missed signals; starvation;
- failure
  - avoidance; 205–220
- improper lock acquisition risk of; 61
- nonblocking algorithm advantages; 319–336
- performance and
  - in servlets with state; 29–32
- safety vs.
  - *See* safety;
- term definition; 8
- testing
  - criteria; 248
- thread safety hazards for; 8

local variables
- *See also* encapsulation; state; variables;
- for thread confinement; 43
- stack confinement use; 44

locality, loss of
- as cost of thread use; 8

`Lock`; 277$_{li}$, 277–282
- and `Condition`; 307
- interruptible acquisition; 148
- timed acquisition; 215

lock(ing); 85
- *See also* confinement; encapsulation; `@GuardedBy`; safety; synchronization;
- acquisition
  - AQS-based synchronizer operations; 311–313
  - improper, liveness risk; 61
  - interruptible; 279–281
  - intrinsic, non-interruptable blocking reason; 148
  - nested, as deadlock risk; 208
  - polled; 279
  - protocols, instance confinement use; 60
  - reentrant lock count; 26
  - timed; 279
- and instance confinement; 59
- atomic variables vs.; 326–329
- avoidance
  - immutable objects use; 49
- building
  - AQS use; 311
- client-side; 72–73, **73**
  - and compound actions; 79–82
  - condition queue encapsulation impact on; 306
  - stream class management; 150$_{fn}$
  - vs. class extension; 73
- coarsening; **231**
  - as JVM optimization; 286
  - impact on splitting synchronized blocks; 235$_{fn}$
- concurrency design rules role; 110
- `ConcurrentHashMap` strategy; 85
- `ConcurrentModificationException` avoidance with; 82
- condition variable and condition predicate relationship; 308
- contention
  - measurement; 240–241
  - reduction, guidelines; 233
  - reduction, impact; 211
  - reduction, strategies; 232–242
  - scalability impact of; 232
- coupling; **282**
- cyclic locking dependencies
  - as deadlock cause; 205
- disadvantages of; 319–321
- double-checked
  - as concurrency bug pattern; 272
- elision; 231$_{fn}$
  - as JVM optimization; 286
- encapsulation of
  - reentrancy facilitation; 27
- exclusive
  - alternative to; 239–240
  - alternatives to; 321
  - inability to use, as `ConcurrentHashMap` disadvantage; 86
  - timed lock use; 279
- explicit; 277–290
  - interruption during lock acquisition use; 148
- granularity
  - Amdahl's law insights; 229

    - reduction of; 235–237
  - hand-over-hand; **282**
  - in blocking actions; 292
  - intrinsic; **25**, 25–26
    - acquisition, non-interruptable blocking reason; 148
    - advantages of; 285
    - explicit locks vs.; 277–278
    - intrinsic condition queue relationship to; 297
    - limitations of; 28
    - private locks vs.; 61
    - recursion use; 237$_{fn}$
    - `ReentrantLock` vs., performance considerations; 282–286
  - iteration
    - concurrent collection elimination of need for; 85
    - disadvantages of; 83
  - monitor
    - *See* intrinsic locks;
  - non-block-structured; 281–282
  - nonblocking algorithms vs.; 319
  - open calls
    - for deadlock avoidance; 211–213
  - ordering
    - deadlock risks; 206–213
    - dynamic, deadlocks resulting from; 207–210
    - inconsistent, as multithreaded GUI framework problem; 190
  - private
    - intrinsic locks vs.; 61
  - protocols
    - shared state requirements for; 28
  - read-write; 286–289
    - implementation strategies; 287
  - reentrant
    - semantics; 26–27
    - semantics, `ReentrantLock` capabilities; 278
  - `ReentrantLock` fairness options; 283–285
  - release
    - in hand-over-hand locking; 282
    - intrinsic locking disadvantages; 278
    - preference, in read-write lock implementation; 287
  - role
    - synchronization policy; 56
  - scope
    - *See also* lock(ing), granularity;
    - narrowing, as lock contention reduction strategy; 233–235
  - splitting; 235
    - Amdahl's law insights; 229
    - as lock granularity reduction strategy; 235
    - `ServerStatus` examples; 236$_{li}$
  - state guarding with; 27–29
  - striping; **237**
    - Amdahl's law insights; 229
    - `ConcurrentHashMap` use; 85
  - stripping; 237
  - thread dump information about; 216
  - thread-safety issues
    - in servlets with state; 23–29
  - timed; 215–216
  - unreleased
    - as concurrency bug pattern; 272
  - visibility and; 36–37
  - volatile variables vs.; 39
  - `wait`
    - and condition predicate; 299
- **lock-free algorithms**; **329**
- **logging**
  - *See also* instrumentation;
  - exceptions
    - `UEHLogger` example; 163$_{li}$
  - service
    - as example of stopping a thread-based service; 150–155
  - thread customization example; 177
  - `ThreadPoolExecutor` hooks for; 179
- **logical state**; **58**
- **loops/looping**
  - and interruption; 143

M

- **main event loop**
  - vs. event dispatch thread; 5
- **`Map`**
  - `ConcurrentHashMap` as concurrent replacement; 84
    - performance advantages; 242
  - atomic operations; 86
- **maximum pool size parameter**; **172**
- **measurement**
  - importance for effective optimization; 224
  - performance; 222

  - profiling tools; 225
    - lock contention; 240
  - responsiveness; 264–266
  - strategies and tools
    - profiling tools; 225
  - `ThreadPoolExecutor` hooks for; 179

memoization; **103**
  - *See also* cache/caching;

memory
  - *See also* resource(s);
  - barriers; **230**, **338**
  - depletion
    - avoiding request overload; 173
    - testing for; 257
    - thread-per-task policy issue; 116
  - models
    - hardware architecture; 338
    - JMM; 337–352
  - reordering
    - operations; 339
  - shared memory multiprocessors; 338–339
  - synchronization
    - performance impact of; 230–231
  - thread pool size impact; 171
  - visibility; 33–39
    - `ReentrantLock` effect; 277
    - `synchronized` effect; 33

Michael-Scott nonblocking queue; 332–335

missed signals; 301, **301**
  - *See also* liveness;
  - as single notification risk; 302

model(s)/modeling
  - *See also* Java Memory Model (JMM); MVC (model-view-controller) design pattern; representation; views;
  - event handling
    - model-view objects; 195_{fg}
  - memory
    - hardware architecture; 338
    - JMM; 337–352
  - model-view-controller pattern
    - deadlock risk; 190
    - vehicle tracking example; 61
  - programming
    - sequential; 2
  - shared data
    - *See also* page renderer examples;
    - in GUI applications; 198–202
  - simplicity
    - threads benefit for; 3
  - split data models; **201**, 201–202
  - Swing event handling; 194
  - three-tier application
    - performance vs. scalability; 223
  - versioned data model; **201**

modification
  - concurrent
    - synchronized collection problems with; 82
  - frequent need for
    - copy-on-write collection not suited for; 87

monitor(s)
  - *See also* Java monitor pattern;
  - locks
    - *See* intrinsic locks;

monitoring
  - *See also* instrumentation; performance; scalability; testing; tools;
  - CPU utilization; 240–241
  - performance; 221–245
  - `ThreadPoolExecutor` hooks for; 179
  - tools
    - for quality assurance; 273

monomorphic call transformation
  - JVM use; 268_{fn}

`mpstat` application; 240
  - *See also* measurement; tools;

multiple-reader, single-writer locking
  - and lock contention reduction; 239
  - read-write locks; 286–289

multiprocessor systems
  - *See also* concurrent/concurrency;
  - shared memory
    - memory models; 338–339
  - threads use of; 3

multithreaded
  - *See also* safety; single-thread(ed); thread(s);
  - GUI frameworks
    - issues with; 189–190

multivariable invariants
  - and atomic variables; 325–326
  - atomicity requirements; 57, 67–68
  - dependencies, thread safety issues; 24
  - locking requirements for; 29

- preservation of, as thread safety requirement; 24

mutable; **15**
- objects
  - safe publication of; 54
- state
  - managing access to, as thread safety goal; 15

mutexes (mutual exclusion locks); **25**
- binary semaphore use as; 99
- intrinsic locks as; 25
- ReentrantLock capabilities; 277

MVC (model-view-controller) pattern
- deadlock risks; 190
- vehicle tracking example use of; 61

N

narrowing
- lock scope
  - as lock contention reduction strategy; 233–235

native code
- finalizer use and limitations; 165

navigation
- as compound action
  - in collection operations; 79

`newTaskFor`; 126_{li}
- encapsulating non-standard cancellation; 148

nonatomic 64-bit operations; 36

nonblocking algorithms; **319**, **329**, 329–336
- backoff importance for; 231_{fn}
- synchronization; 319–336
- `SynchronousQueue`; 174_{fn}
- thread-safe counter use; 322–324

nonfair semaphores
- advantages of; 265

notification; 302–304
- *See also* blocking; condition, queues; event(s); listeners; `notify`; `notifyAll`; sleeping; wait(s); waking up;
- completion
  - of long-running GUI task; 198
- conditional; **303**
  - as optimization; 303
  - use; 304_{li}
- errors
  - as concurrency bug pattern; 272
- event notification systems
  - copy-on-write collection advantages; 87

`notify`
- as optimization; 303
- efficiency of; 298_{fn}
- missed signal risk; 302
- `notifyAll` vs.; 302
- subclassing safety issues
  - documentation importance; 304
- usage guidelines; 303

`notifyAll`
- `notify` vs.; 302

`@NotThreadSafe`; 6, 353

NPTL threads package
- Linux use; 4_{fn}

nulling out memory references
- testing use; 257

O

object(s)
- *See also* resource(s);
- composing; 55–78
- condition
  - explicit; 306–308
- effectively immutable; **53**
- guarded; **54**
- immutable; **46**
  - initialization safety; 51
  - publication using volatile; 48–49
- mutable
  - safe publication of; 54
- pools
  - appropriate uses; 241_{fn}
  - bounded, semaphore management of; 99
  - disadvantages of; 241
  - serial thread confinement use; 90
- references
  - and stack confinement; 44
- sharing; 33–54
- state; **55**
  - components of; 55
- Swing
  - thread-confinement; 191–192

objects
- guarded; **28**

open calls; **211**, 211–213
- *See also* encapsulation;

operating systems
- concurrency use
  - historical role; 1

- **operations**
  - 64-bit, nonatomic nature of; 36
  - state-dependent; **57**
- **optimistic concurrency management**
  - *See* atomic variables; CAS; nonblocking algorithms;
- **optimization**
  - compiler
    - as performance testing pitfall; 268–270
  - JVM
    - pitfalls; 38$_{fn}$
    - strategies; 286
  - lock contention
    - impact; 231
    - reduction strategies; 232–242
  - performance
    - Amdahl's law; 225–229
    - premature, avoidance of; 223
    - questions about; 224
    - scalability requirements vs.; 222
  - techniques
    - *See also* atomic variabless; nonblocking synchronization;
    - condition queues use; 297
    - conditional notification; 303
- **order(ing)**
  - *See also* reordering; synchronization;
  - acquisition, in `ReentrantReadWriteLock`; 317$_{fn}$
  - checksums
    - safety testing use; 253
  - FIFO
    - impact of caller state dependence handling on; 294$_{fn}$
  - lock
    - deadlock risks; 206–213
    - dynamic deadlock risks; 207–210
    - inconsistent, as multithreaded GUI framework problem; 190
  - operation
    - synchronization role; 35
  - partial; 340$_{fn}$
    - happens-before, JMM definition; 340–342
    - happens-before, piggybacking; 342–344
    - happens-before, publication consequences; 244–249
  - performance-based alterations in
    - thread safety risks; 7
  - total
    - synchronization actions; 341
- **orderly shutdown**; **164**
- **`OutOfMemoryError`**
  - unbounded thread creation risk; 116
- **overhead**
  - *See also* CPU utilization; measurement; performance;
  - impact of
    - *See* performance; throughput;
  - reduction
    - *See* nonblocking algorithms; optimization; thread(s), pools;
  - sources
    - *See* blocking/blocks; contention; context switching; multithreaded environments; safety; suspension; synchronization; thread(s), lifecycle;
- **ownership**
  - shared; **58**
  - split; **58**
  - state
    - class design issues; 57–58
  - thread; **150**

P

- **page renderer examples**
  - *See also* model(s)/modeling, shared data;
  - heterogenous task partitioning; 127–129
  - parallelism analysis; 124–133
  - sequential execution; 124–127
- **parallelizing/parallelism**
  - *See also* concurrent/concurrency; Decorator pattern;
  - application analysis; 123–133
  - heterogeneous tasks; 127–129
  - iterative algorithms
    - barrier management of; 99
  - puzzle-solving framework; 183–188
  - recursive algorithms; 181–188
  - serialization vs.
    - Amdahl's law; 225–229
  - task-related decomposition; 113
  - thread-per-task policy; 115
- **partial ordering**; 340$_{fn}$
  - happens-before
    - and publication; 244–249
    - JMM definition; 340

    piggybacking; 342–344
partitioning
  as parallelizing strategy; 101
passivation
  impact on `HttpSession` thread-safety requirements; 58$_{fn}$
`perfbar` application
  See also measurement; tools;
  CPU performance measure; 261
  performance measurement use; 225
`perfmon` application; 240
  See also measurement; tools;
  I/O measurement; 240
  performance measurement use; 230
performance; **8**, **221**, 221–245
  See also concurrent/concurrency; liveness; scalability; throughput; utilization;
  and heterogeneous tasks; 127
  and immutable objects; 48$_{fn}$
  and resource management; 119
  atomic variables
    locking vs.; 326–329
  cache implementation issues; 103
  composition functionality extension mechanism; 74$_{fn}$
  costs
    thread-per-task policy; 116
  fair vs. nonfair locking; 284
  hazards
    See also overhead; priority(s), inversion;
  JVM interaction with hardware reordering; 34
  liveness
    in servlets with state; 29–32
  locking
    during iteration impact on; 83
  measurement of; 222
    See also capacity; efficiency; latency; scalability; service time; throughput;
    locks vs. atomic variables; 326–329
  memory barrier impact on; 230
  `notifyAll` impact on; 303
  optimization
    See also CPU utilization; piggybacking;
    Amdahl's law; 225–229
    bad practices; 348–349
    CAS-based operations; 323
    reduction strategies; 232–242
  page renderer example with `CompletionService`
    improvements; 130
  producer-consumer pattern advantages; 90
  read-write lock advantages; 286–289
  `ReentrantLock` vs. intrinsic locks; 282–286
  requirements
    thread-safety impact; 16
  scalability vs.; 222–223
    issues, three-tier application model as illustration; 223
    lock granularity reduction; 239
    object pooling issues; 241
  sequential event processing; 191
  simplicity vs.
    in refactoring `synchronized` blocks; 34
  `synchronized` block scope; 30
  `SynchronousQueue`; 174$_{fn}$
  techniques for improving
    atomic variables; 319–336
    nonblocking algorithms; 319–336
  testing; 247–274
    criteria; 248
    goals; 260
    pitfalls, avoiding; 266–270
  thread pool
    size impact; 170
    tuning; 171–179
  thread safety hazards for; 8
  timing and ordering alterations for thread safety risks; 7
  tradeoffs
    evaluation of; 223–225
permission
  codebase
    and custom thread factory; 177
permits; **98**
  See also semaphores;
pessimistic concurrency management
  See lock(ing), exclusive;
piggybacking; **344**
  on synchronization; 342–344
point(s)
  barrier; **99**
  cancellation; **140**

poison
- message; **219**
  - *See also* livelock;
- pill; **155**, 155–156
  - *See also* lifecycle; shutdown;
  - `CrawlerThread`; 157_{li}
  - `IndexerThread`; 157_{li}
  - `IndexingService`; 156_{li}
  - unbounded queue shutdown with; 155

policy(s)
- *See also* design; documentation; guidelines; protocol(s); strategies;
- application
  - thread pool advantages; 120
- cancellation; **136**
  - for tasks, thread interruption policy relationship to; 141
  - interruption advantages as implementation strategy; 140
- execution
  - design, influencing factors; 167
  - `Executors`, for `ThreadPoolExecutor` configuration; 171
  - implicit couplings between tasks and; 167–170
  - parallelism analysis for; 123–133
  - task; 118–119
  - task, application performance importance; 113
- interruption; **141**, 141–142
- saturation; 174–175
- security
  - custom thread factory handling; 177
- sequential
  - task execution; 114
- sharing objects; 54
- synchronization; **55**
  - requirements, impact on class extension; 71
  - requirements, impact on class modification; 71
  - shared state requirements for; 28
- task scheduling
  - sequential; 114
  - thread pools; 117
  - thread pools advantages over thread-per-task; 121
  - thread-per-task; 115
- thread confinement; 43

polling
- blocking state-dependent actions; 295–296
- for interruption; 143
- lock acquisition; 279

pool(s)
- *See also* resource(s);
- object
  - appropriate uses; 241_{fn}
  - bounded, semaphore use; 99
  - disadvantages of; 241
  - serial thread confinement use; 90
- resource
  - semaphore use; 98–99
  - thread pool size impact; 171
- size
  - core; **171**, 172_{fn}
  - maximum; **172**
- thread; 119–121
  - adding statistics to; 179
  - application; 167–188
  - as producer-consumer design; 88
  - as thread resource management mechanism; 117
  - callback use in testing; 258
  - combined with work queues, in `Executor` framework; 119
  - configuration post-construction manipulation; 177–179
  - configuring task queue; 172–174
  - creating; 120
  - deadlock risks; 215
  - factory methods for; 171
  - sizing; 170–171
  - uncaught exception handling; 163

portal
- timed task example; 131–133

postconditions
- *See also* invariant(s);
- preservation of
  - mechanisms and synchronization policy role; 55–56
- thread safety role; 17

precondition(s)
- *See also* dependencies, state; invariant(s);
- condition predicate as; 299
- failure
  - bounded buffer handling of; 292

        propagation to callers; 292–295
    state-based
        in state-dependent classes; 291
        management; 57
predictability
    See also responsiveness;
    measuring; 264–266
preemptive interruption
    deprecation reasons; 135$_{fn}$
presentation
    See GUI;
primitive
    local variables, safety of; 44
    wrapper classes
        atomic scalar classes vs.; 325
priority(s)
    inversion; **320**
        avoidance, nonblocking algorithm advantages; 329
    thread
        manipulation, liveness hazards; 218
        when to use; 219
`PriorityBlockingQueue`; 89
    thread pool use of; 173–174
`PriorityQueue`; 85
private
    constructor capture idiom; 69$_{fn}$
    locks
        Java monitor pattern vs.; 61
probability
    deadlock avoidance use with timed and polled locks; 279
    determinism vs.
        in concurrent programs; 247
process(es); **1**
    communication mechanisms; 1
    lightweight
        See threads;
    threads vs.; 2
producer-consumer pattern
    and `Executor` functionality
        in `CompletionService`; 129
    blocking queues and; 87–92
    bounded buffer use; 292
    control flow coordination
        blocking queues use; 94
    `Executor` framework use; 117
    pathological waiting conditions; 300$_{fn}$
    performance testing; 261
    safety testing; 252
    work stealing vs.; 92
profiling
    See also measurement;
    JVM use; 320$_{fn}$
    tools
        lock contention detection; 240
        performance measurement; 225
        quality assurance; 273
programming
    models
        sequential; 2
progress indication
    See also GUI;
    in long-running GUI task; 198
propagation
    of interruption exception; 142
protocol(s)
    See also documentation; policy(s); strategies;
    entry and exit
        state-dependent operations; 306
    lock acquisition
        instance confinement use; 60
    locking
        shared state requirements for; 28
    race condition handling; 21
    thread confinement
        atomicity preservation with open calls; 213
pthreads (POSIX threads)
    default locking behavior; 26$_{fn}$
publication; **39**
    See also confinement; documentation; encapsulation; sharing;
    escape and; 39–42
    improper; **51**, 50–51
    JMM support; 244–249
    of immutable objects
        volatile use; 48–49
    safe; 346
        idioms for; 52–53
        in task creation; 126
        of mutable objects; 54
        serial thread confinement use; 90
    safety guidelines; 49–54
    state variables
        safety, requirements for; 68–69
    unsafe; 344–346

put-if-absent operation
- *See also* compound actions;
- as compound action
  - atomicity requirements; 71
- concurrent collection support for; 84

puzzle solving framework
- as parallelization example; 183–188

Q

quality assurance
- *See also* testing;
- strategies; 270–274

quality of service
- measuring; 264
- requirements
  - and task execution policy; 119

Queue; 84–85

queue(s)
- *See also* data structures;
- blocking; 87–94
  - cancellation, problems; 138
  - cancellation, solutions; 140
  - CompletionService as; 129
  - producer-consumer pattern and; 87–92
- bounded
  - saturation policies; 174–175
- condition; **297**
  - blocking state-dependent operations use; 296–308
  - intrinsic; 297
  - intrinsic, disadvantages of; 306
- FIFO; 89
- implementations
  - serialization differences; 227
- priority-ordered; 89
- synchronous
  - design constraints; 89
  - thread pool use of; 173
- task
  - thread pool use of; 172–174
- unbounded
  - poison pill shutdown; 156
- using; 298
- work
  - in thread pools; 88, 119

R

race conditions; **7**, 20–22
- *See also* concurrent/concurrency, errors; data, race; time/timing;
- avoidance
  - immutable object use; 48
  - in thread-based service shutdown; 153
- in GUI frameworks; 189
- in web crawler example
  - idempotence as mitigating circumstance; 161

random(ness)
- livelock resolution use; 219
- pseudorandom number generation
  - scalability; 326–329
- test data generation use; 253

reachability
- publication affected by; 40

read-modify-write operation
- *See also* compound actions;
- as non-atomic operation; 20

read-write locks; 286–289

ReadWriteLock; 286$_{li}$
- exclusive locking vs.; 239

reaping
- *See* termination;

reclosable thread gate; 304

recovery, deadlock
- *See* deadlock, recovery;

recursion
- *See also* control flow; iterators/iteration;
- intrinsic lock acquisition; 237$_{fn}$
- parallelizing; 181–188
  - *See also* Decorator pattern;

reentrant/reentrancy; **26**
- and read-write locks; 287
- locking semantics; 26–27
  - ReentrantLock capabilities; 278
- per-thread lock acquisition; 26–27
- ReentrantLock; 277–282

ReentrantLock
- AQS use; 314–315
- intrinsic locks vs.
  - performance; 282–286
- Lock implementation; 277–282
- random number generator using; 327$_{li}$
- Semaphore relationship with; 308

ReentrantReadWriteLock
- AQS use; 316–317
- reentrant locking semantics; 287

references
- stack confinement precautions; 44

reflection
- atomic field updater use; 335

rejected execution handler
- ExecutorService post-termination task handling; 121
- puzzle-solving framework; 187

RejectedExecutionException
- abort saturation policy use; 174
- post-termination task handling; 122
- puzzle-solving framework use; 187

RejectedExecutionHandler
- and saturation policy; 174

release
- AQS synchronizer operation; 311
- lock
  - in hand-over-hand locking; 282
  - intrinsic locking disadvantages; 278
  - preferences in read-write lock implementation; 287
  - unreleased lock bug pattern; 271
- permit
  - semaphore management; 98

remote objects
- thread safety concerns; 10

remove-if-equal operation
- as atomic collection operation; 86

reordering; **34**
- *See also* deadlock; optimization; order(ing); ordering; synchronization; time/timing;
- initialization safety limitation; 350
- memory
  - barrier impact on; 230
  - operations; 339
- volatile variables warning; 38

replace-if-equal operation
- as atomic collection operation; 86

representation
- *See also* algorithm(s); design; documentation; state(s);
- activities
  - tasks use for; 113
- algorithm design role; 104
- result-bearing tasks; 125
- task
  - lifecycle, Future use for; 125
  - Runnable use for; 125
  - with Future; 126
- thread; 150

request
- interrupt
  - strategies for handling; 140

requirements
- *See also* constraints; design; documentation; performance;
- concrete
  - importance for effective performance optimization; 224
- concurrency testing
  - TCK example; 250
- determination
  - importance of; 223
- independent state variables; 66–67
- performance
  - Amdahl's law insights; 229
  - thread-safety impact; 16
- synchronization
  - synchronization policy component; 56–57
- synchronization policy documentation; 74–77

resource exhaustion, preventing
- bounded queue use; 173
- execution policy as tool for; 119
- testing strategies; 257
- thread pool sizing risks; 170

resource(s)
- *See also* CPU; instrumentation; memory; object(s); pool(s); utilization;
- accessing
  - as long-running GUI task; 195
- bound; **221**
- consumption
  - thread safety hazards for; 8
- deadlocks; 213–215
- depletion
  - thread-per-task policy issue; 116
- increase
  - scalability relationship to; 222
- leakage
  - testing for; 257
- management
  - *See also* instrumentation; testing; dining philosophers problem;

blocking queue advantages; 88
execution policy as tool for; 119
Executor framework use; 117
finalizer use and limitations; 165
graceful degradation, saturation policy advantages; 175
long-running task handling; 170
saturation policies; 174–175
single-threaded task execution disadvantages; 114
testing; 257
thread pools; 117
thread pools, advantages; 121
thread pools, tuning; 171–179
thread-per-task policy disadvantages; 116
threads, keep-alive time impact on; 172
timed task handling; 131
performance
analysis, monitoring, and improvement; 221–245
pools
semaphore use; 98–99
thread pool size impact; 171
utilization
Amdahl's law; 225
as concurrency motivation; 1

response-time-senstive tasks
execution policy implications; 168

responsiveness
See also deadlock; GUI; livelock; liveness; performance;
as performance testing criteria; 248
condition queues advantages; 297
efficiency vs.
polling frequency; 143
interruption policy
InterruptedException advantages; 142
long-running tasks
handling; 170
measuring; 264–266
page renderer example with CompletionService
improvements; 130
performance
analysis, monitoring, and improvement; 221–245
poor
causes and resolution of; 219
safety vs.
graceful vs. abrupt shutdown; 153
sequential execution limitations; 124
server applications
importance of; 113
single-threaded execution disadvantages; 114
sleeping impact on; 295
thread
pool tuning, ThreadPoolExecutor use; 171–179
request overload impact; 173
safety hazards for; 8

restoring interruption status; 142

result(s)
-bearing latches
puzzle framework use; 184
cache
building; 101–109
Callable handling of; 125
Callable use instead of Runnable; 95
dependencies
task freedom from, importance of; 113
Future handling of; 125
handling
as serialization source; 226
irrelevancy
as cancellation reason; 136, 147
non-value-returning tasks; 125
Runnable limitations; 125

retry
randomness, in livelock resolution; 219

return values
Runnable limitations; 125

reuse
existing thread-safe classes
strategies and risks; 71

RMI (Remote Method Invocation)
thread use; 9, 10
safety concerns and; 10
threads benefits for; 4

robustness
See also fragility; safety;
blocking queue advantages; 88
InterruptedException advantages; 142
thread pool advantages; 120

rules
- *See also* guidelines; policy(s); strategies;
- happens-before; 341

Runnable
- handling exceptions in; 143
- task representation limitations; 125

running
- ExecutorService state; 121
- FutureTask state; 95

runtime
- timing and ordering alterations by
  - thread safety risks; 7

RuntimeException
- as thread death cause; 161
- Callable handling; 98
- catching
  - disadvantages of; 161

S

safety
- *See also* encapsulation; immutable objects; synchronization; thread(s), confinement;
- cache implementation issues; 104
- initialization
  - guarantees for immutable objects; 51
  - idioms for; 346–348
  - JMM support; 349–350
- liveness vs.; 205–220
- publication
  - idioms for; 52–53
  - in task creation; 126
  - of mutable objects; 54
- responsiveness vs.
  - as graceful vs. abrupt shutdown; 153
- split ownership concerns; 58
- subclassing issues; 304
- testing; 252–257
  - goals; 247
- tradeoffs
  - in performance optimization strategies; 223–224
- untrusted code behavior
  - protection mechanisms; 161

saturation
- policies; 174–175

scalability; **222**, 221–245
- algorithm
  - comparison testing; 263–264
- Amdahl's law insights; 229
- as performance testing criteria; 248
- client-side locking impact on; 81
- concurrent collections vs. synchronized collections; 84
- ConcurrentHashMap advantages; 85, 242
- CPU utilization monitoring; 240–241
- enhancement
  - reducing lock contention; 232–242
- heterogeneous task issues; 127
- hot field impact on; 237
- intrinsic locks vs. ReentrantLock
  - performance; 282–286
- lock scope impact on; 233
- locking during iteration risk of; 83
- open call strategy impact on; 213
- performance vs.; 222–223
  - lock granularity reduction; 239
  - object pooling issues; 241
  - three-tier application model as illustration; 223
- queue implementations
  - serialization differences; 227
- result cache
  - building; 101–109
- serialization impact on; 228
- techniques for improving
  - atomic variables; 319–336
  - nonblocking algorithms; 319–336
- testing; 261
- thread safety hazards for; 8
- under contention
  - as AQS advantage; 311

ScheduledThreadPoolExecutor
- as Timer replacement; 123

scheduling
- overhead
  - performance impact of; 222
- priority manipulation risks; 218
- tasks
  - sequential policy; 114
  - thread-per-task policy; 115
- threads as basic unit of; 3
- work stealing
  - deques and; 92

scope/scoped
- *See also* granularity;
- containers
  - thread safety concerns; 10
- contention
  - atomic variable limitation of; 324
- escaping
  - publication as mechanism for; 39
- lock
  - narrowing, as lock contention reduction strategy; 233–235
- `synchronized` block; 30

search
- depth-first
  - breadth-first search vs.; 184
  - parallelization of; 181–182

security policies
- and custom thread factory; 177

`Selector`
- non-interruptable blocking; 148

semantics
- *See also* documentation; representation;
- atomic arrays; 325
- binary semaphores; 99
- final fields; 48
- of interruption; 93
- of multithreaded environments
  - `ThreadLocal` variable considerations; 46
- reentrant locking; 26–27
  - `ReentrantLock` capabilities; 278
  - `ReentrantReadWriteLock` capabilities; 287
- undefined
  - of `Thread.yield`; 218
- `volatile`; 39
- weakly consistent iteration; 85
- within-thread-as-if-serial; **337**

`Semaphore`; 98
- AQS use; 315–316
- example use; 100_{li}, 176_{li}, 249_{li}
- in `BoundedBuffer` example; 248
- saturation policy use; 175
- similarities to `ReentrantLock`; 308
- state-based precondition management with; 57

semaphores; **98**, 98–99
- as coordination mechanism; 1
- binary
  - mutex use; 99
- counting; **98**
  - permits, thread relationships; 248_{fn}
  - `SemaphoreOnLock` example; 310_{li}
- fair vs. nonfair
  - performance comparison; 265
- nonfair
  - advantages of; 265

`sendOnSharedLine` example; 281_{li}

sequential/sequentiality
- *See also* concurrent/concurrency;
- asynchrony vs.; 2
- consistency; **338**
- event processing
  - in GUI applications; 191
- execution
  - of tasks; 114
  - parallelization of; 181
- orderly shutdown strategy; 164
- page renderer example; 124–127
- programming model; 2
- task execution policy; 114
- tests, value in concurrency testing; 250
- threads simulation of; 4

serialized/serialization
- access
  - object serialization vs.; 27_{fn}
  - timed lock use; 279
  - `WorkerThread`; 227_{li}
- granularity
  - throughput impact; 228
- impact on `HttpSession` thread-safety requirements; 58_{fn}
- parallelization vs.
  - Amdahl's law; 225–229
- scalability impact; 228
- serial thread confinement; **90**, 90–92
- sources
  - identification of, performance impact; 225

server
- *See also* client;
- applications
  - context switch reduction; 243–244
  - design issues; 113

service(s)
- *See also* applications; frameworks;
- logging

as thread-based service example; 150–155
shutdown
as cancellation reason; 136
thread-based
stopping; 150–161
servlets
framework
thread safety requirements; 10
threads benefits for; 4
stateful, thread-safety issues
atomicity; 19–23
liveness and performance; 29–32
locking; 23–29
stateless
as thread-safety example; 18–19
session-scoped objects
thread safety concerns; 10
set(s)
See also collection(s);
`BoundedHashSet` example; 100$_{li}$
`CopyOnWriteArraySet`
as synchronized `Set` replacement; 86
safe publication use; 52
`PersonSet` example; 59$_{li}$
`SortedSet`
`ConcurrentSkipListSet` as concurrent replacement; 85
`TreeSet`
`ConcurrentSkipListSet` as concurrent replacement; 85
shared/sharing; **15**
See also concurrent/concurrency; publication;
data
See also page renderer examples;
access coordination, explicit lock use; 277–290
models, GUI application handling; 198–202
synchronization costs; 8
threads advantages vs. processes; 2
data structures
as serialization source; 226
memory
as coordination mechanism; 1
memory multiprocessors
memory models; 338–339
mutable objects
guidelines; 54
objects; 33–54
split data models; 201–202
state
managing access to, as thread safety goal; 15
strategies
`ExecutorCompletionService` use; 130
thread
necessities and dangers in GUI applications; 189–190
volatile variables as mechanism for; 38
shutdown
See also lifecycle;
abrupt
JVM, triggers for; 164
limitations; 158–161
as cancellation reason; 136
cancellation and; 135–166
`ExecutorService` state; 121
graceful vs. abrupt tradeoffs; 153
hooks; **164**
in orderly shutdown; 164–165
JVM; 164–166
and daemon threads; 165
of thread-based services; 150–161
orderly; **164**
strategies
lifecycle method encapsulation; 155
logging service example; 150–155
one-shot execution service example; 156–158
support
`LifecycleWebServer` example; 122$_{li}$
`shutdown`; 121
logging service shutdown alternatives; 153
`shutdownNow`; 121
limitations; 158–161
logging service shutdown alternatives; 153
side-effects
as serialization source; 226
freedom from
importance for task independence; 113

- synchronized `Map` implementations
  - not available from `ConcurrentHashMap`; 86

`signal`
- `ConditionBoundedBuffer` example; 308

signal handlers
- as coordination mechanism; 1

simplicity
- *See also* design;
- Java monitor pattern advantage; 61
- of modeling
  - threads benefit for; 3
- performance vs.
  - in refactoring `synchronized` blocks; 34

simulations
- barrier use in; 101

single notification
- *See* `notify`; `signal`;

single shutdown hook
- *See also* hook(s);
- orderly shutdown strategy; 164

single-thread(ed)
- *See also* thread(s); thread(s), confinement;
- as `Timer` restriction; 123
- as synchronization alternative; 42–46
- deadlock avoidance advantages; 43$_{fn}$
- subsystems
  - GUI implementation as; 189–190
- task execution
  - disadvantages of; 114
  - executor use, concurrency prevention; 172, 177–178

Singleton pattern
- `ThreadLocal` variables use with; 45

size(ing)
- *See also* configuration; instrumentation;
- as performance testing goal; 260
- bounded buffers
  - determination of; 261
- heterogeneous tasks; 127
- pool
  - core; **171**, 172$_{fn}$
  - maximum; **172**
- task
  - appropriate; 113
- thread pools; 170–171

sleeping
- blocking state-dependent actions
  - blocking state-dependent actions; 295–296

sockets
- as coordination mechanism; 1
- synchronous I/O
  - non-interruptable blocking reason; 148

solutions
- *See also* interruption; results; search; termination;

`SortedMap`
- `ConcurrentSkipListMap` as concurrent replacement; 85

`SortedSet`
- `ConcurrentSkipListSet` as concurrent replacement; 85

space
- state; **56**

specification
- *See also* documentation;
- correctness defined in terms of; 17

spell checking
- as long-running GUI task; 195

spin-waiting; 232, **295**
- *See also* blocking/blocks; busy-waiting;
- as concurrency bug pattern; 273

split(ing)
- data models; **201**, 201–202
- lock; 235
  - Amdahl's law insights; 229
  - as lock granularity reduction strategy; 235
  - `ServerStatus` examples; 236$_{li}$
- ownership; **58**

stack(s)
- address space
  - thread creation constraint; 116$_{fn}$
- confinement; **44**, 44–45
  - *See also* confinement; encapsulation;
- nonblocking; 330
- size
  - search strategy impact; 184
- trace
  - thread dump use; 216

stale data; 35–36
- improper publication risk; 51
- race condition cause; 20$_{fn}$

- **starvation**; 218, **218**
  - *See also* deadlock; livelock; liveness; performance;
  - as liveness failure; 8
  - locking during iteration risk of; 83
  - thread starvation deadlock; **169**, 168–169
  - thread starvation deadlocks; **215**
- **state(s)**; **15**
  - *See also* atomic/atomicity; encapsulation; lifecycle; representation; safety; visibility;
  - application
    - framework threads impact on; 9
  - code vs.
    - thread-safety focus; 17
  - dependent
    - classes; **291**
    - classes, building; 291–318
    - operations; **57**
    - operations, blocking strategies; 291–308
    - operations, condition queue handling; 296–308
    - operations, managing; 291
    - task freedom from, importance of; 113
  - encapsulation
    - breaking, costs of; 16–17
    - invariant protection use; 83
    - synchronizer role; 94
    - thread-safe class use; 23
  - lifecyle
    - `ExecutorService` methods; 121
  - locks control of; 27–29
  - logical; **58**
  - management
    - AQS-based synchronizer operations; 311
  - managing access to
    - as thread safety goal; 15
  - modification
    - visibility role; 33
  - mutable
    - coordinating access to; 110
  - object; **55**
    - components of; 55
    - remote and thread safety; 10
  - ownership
    - class design issues; 57–58
  - servlets with
    - thread-safety issues, atomicity; 19–23
    - thread-safety issues, liveness and performance concerns; 29–32
    - thread-safety issues, locking; 23–29
  - space; **56**
  - stateless servlet
    - as thread-safety example; 18–19
  - task
    - impact on Future.get; 95
    - intermediate, shutdown issues; 158–161
  - transformations
    - in puzzle-solving framework example; 183–188
  - transition constraints; **56**
  - variables
    - condition predicate use; 299
    - independent; **66**, 66–67
    - independent, lock splitting; 235
    - safe publication requirements; 68–69
- **`stateDependentMethod` example**; 301_{li}
- **static**
  - initializer
    - safe publication mechanism; 53, 347
- **static analysis tools**; 271–273
- **statistics gathering**
  - *See also* instrumentation;
  - adding to thread pools; 179
  - `ThreadPoolExecutor` hooks for; 179
- **status**
  - flag
    - volatile variable use with; 38
  - interrupted; **138**
  - thread
    - shutdown issues; 158
- **strategies**
  - *See also* design; documentation; guidelines; policy(s); representation;
  - atomic variable use; 34
  - cancellation
    - Future use; 145–147
  - deadlock avoidance; 208, 215–217
  - delegation
    - vehicle tracking example; 64
  - design

    - interruption policy; 93
- documentation use
    - annotations value; 6
- end-of-lifecycle management; 135–166
- `InterruptedException` handling; 93
- interruption handling; 140, 142–150
    - `Future` use; 146
- lock splitting; 235
- locking
    - `ConcurrentHashMap` advantages; 85
- monitor
    - vehicle tracking example; 61
- parallelization
    - partitioning; 101
- performance improvement; 30
- program design order
    - correctness then performance; 16
- search
    - stack size impact on; 184
- shutdown
    - lifecycle method encapsulation; 155
    - logging service example; 150–155
    - one-shot execution service example; 156–158
    - poison pill; 155–156
- split ownership safety; 58
- thread safety delegation; 234–235
- thread-safe class extension; 71

stream classes
- client-side locking with; 150$_{fn}$
- thread safety; 150

String
- immutability characteristics; 47$_{fn}$

striping
- *See also* contention;
- lock; 237, **237**
    - Amdahl's law insights; 229
    - ConcurrentHashMap use; 85

structuring
- thread-safe classes
    - object composition use; 55–78

subclassing
- safety issues; 304

`submit`, `execute` vs.
- uncaught exception handling; 163

suspension, thread
- costs of; 232, 320
- elimination by CAS-based concurrency mechanisms; 321
- `Thread.suspend`, deprecation reasons; 135$_{fn}$

swallowing interrupts
- as discouraged practice; 93
- bad consequences of; 140
- when permitted; 143

Swing
- *See also* GUI;
- listeners
    - single-thread rule exceptions; 192
- methods
    - single-thread rule exceptions; 191–192
- thread
    - confinement; 42
    - confinement in; 191–192
    - use; 9
    - use, safety concerns and; 10–11
- untrusted code protection mechanisms in; 162

`SwingWorker`
- long-running GUI task support; 198

synchronization/`synchronized`; 15
- *See also* access; concurrent/concurrency; lock(ing); safety;;
- allocation advantages vs.; 242
- bad practices
    - double-checked locking; 348–349
- blocks; **25**
    - Java objects as; 25
- cache implementation issues; 103
- collections; 79–84
    - concurrent collections vs.; 84
    - problems with; 79–82
- concurrent building blocks; 79–110
- contended; **230**
- correctly synchronized program; **341**
- data sharing requirements for; 33–39
- encapsulation
    - hidden iterator management through; 83
    - requirement for thread-safe classes; 18
- 'fast path'
    - CAS-based operations vs.; 324
    - costs of; 230

immutable objects as replacement; 52
inconsistent
as concurrency bug pattern; 271
memory
performance impact of; 230–231
memory visibility use of; 33–39
operation ordering role; 35
piggybacking; 342–344
policy; **55**
documentation requirements; 74–77
encapsulation, client-side locking violation of; 71
race condition prevention with; 7
requirements, impact on class extension; 71
requirements, impact on class modification; 71
shared state requirements for; 28
ReentrantLock capabilities; 277
requirements
synchronization policy component; 56–57
thread safety need for; 5
types
See barriers; blocking, queues; FutureTask; latches; semaphores;
uncontended; **230**
volatile variables vs.; 38
wrapper
client-side locking support; 73

synchronizedList (Collections)
safe publication use; 52

synchronizer(s); **94**, 94–101
See also Semaphore; CyclicBarrier; FutureTask; Exchanger; CountDownLatch;
behavior and interface; 308–311
building
with AQS; 311
with condition queues; 291–318

synchronous I/O
non-interruptable blocking; 148

SynchronousQueue; 89
performance advantages; 174$_{fn}$
thread pool use of; 173, 174

T

task(s); **113**
See also activities; event(s); lifecycle;
asynchronous
FutureTask handling; 95–98
boundaries; **113**
parallelism analysis; 123–133
using ThreadLocal in; 168
cancellation; 135–150
policy; **136**
thread interruption policy relationship to; 141
completion
as cancellation reason; 136
service time variance relationship to; 264–266
dependencies
execution policy implications; 167
thread starvation deadlock risks; 168
execution; 113–134
in threads; 113–115
policies; 118–119
policies and, implicit couplings between; 167–170
policies, application performance importance; 113
sequential; 114
explicit thread creation for; 115
GUI
long-running tasks; 195–198
short-running tasks; 192–195
heterogeneous tasks
parallelization limitations; 127–129
homogeneous tasks
parallelism advantages; 129
lifecycle
Executor phases; 125
ExecutorService methods; 121
representing with Future; 125
long-running
responsiveness problems; 170
parallelization of
homogeneous vs. heterogeneous; 129
post-termination handling; 121
queues
management, thread pool configuration issues; 172–174

- thread pool use of; 172–174
- representation
  - `Runnable` use for; 125
  - with `Future`; 126
- response-time sensitivity
  - andexecution policy; 168
- scheduling
  - thread-per-task policy; 115
- serialization sources
  - identifying; 225
- state
  - effect on `Future.get`; 95
  - intermediate, shutdown issues; 158–161
- thread(s) vs.
  - interruption handling; 141
- timed
  - handling of; 123
- two-party
  - `Exchanger` management of; 101

TCK (Technology Compatibility Kit)
- concurrency testing requirements; 250

teardown
- thread; 171–172

techniques
- *See also* design; guidelines; strategies;

temporary objects
- and `ThreadLocal` variables; 45

terminated
- `ExecutorService` state; 121

termination
- *See also* cancellation; interruption; lifecycle;
- puzzle-solving framework; 187
- safety test
  - criteria for; 254, 257
- thread
  - abnormal, handling; 161–163
  - keep-alive time impact on; 172
  - reasons for deprecation of; 135_{fn}
- timed locks use; 279

test example method; 262_{li}

testing
- *See also* instrumentation; logging; measurement; monitoring; quality assurance; statistics;
- concurrent programs; 247–274
- deadlock risks; 210_{fn}
- functionality
  - vs. performance tests; 260
- liveness
  - criteria; 248
- performance; 260–266
  - criteria; 248
  - goals; 260
- pitfalls
  - avoiding; 266–270
  - dead code elimination; 269
  - dynamic compilation; 267–268
  - garbage collection; 266
  - progress quantification; 248
  - proving a negative; 248
  - timing and synchronization artifacts; 247
  - unrealistic code path sampling; 268
  - unrealistic contention; 268–269
- program correctness; 248–260
- safety; 252–257
  - criteria; 247
- strategies; 270–274

`testPoolExample` example; 258_{li}

`testTakeBlocksWhenEmpty` example; 252_{li}

`this` reference
- publication risks; 41

`Thread`
- `join`
  - timed, problems with; 145
- `getState`
  - use precautions; 251
- interruption methods; 138, 139_{li}
  - usage precautions; 140

thread safety; **18**, 15–32
- and mutable data; 35
- and shutdown hooks; 164
- characteristics of; 17–19
- data models, GUI application handling; 201
- delegation; 62
- delegation of; **234**
- in puzzle-solving framework; 183
- issues, atomicity; 19–23
- issues, liveness and performance; 29–32
- mechanisms, locking; 23–29
- risks; 5–8

thread(s); 2
- *See also* concurrent/concurrency; safety; synchronization;

- abnormal termination of; 161–163
- as instance confinement context; 59
- benefits of; 3–5
- blocking; **92**
- confinement; **42**, 42–46
  - *See also* confinement; encapsulation;
  - ad-hoc; 43
  - and execution policy; 167
  - in GUI frameworks; 190
  - in Swing; 191–192
  - role, synchronization policy specification; 56
  - stack; **44**, 44–45
  - `ThreadLocal`; 45–46
- cost
  - context locality loss; 8
  - context switching; 8
- costs; 229–232
- creation; 171–172
  - explicit creation for tasks; 115
  - unbounded, disadvantages; 116
- daemon; 165
- dumps; **216**
  - deadlock analysis use; 216–217
  - intrinsic lock advantage over `ReentrantLock`; 285
  - lock contention analysis use; 240
- factories; **175**, 175–177
- failure
  - uncaught exception handlers; 162–163
- forced termination
  - reasons for deprecation of; 135$_{fn}$
- interleaving
  - dangers of; 5–8
- interruption; **138**
  - shutdown issues; 158
  - status flag; **138**
- leakage; **161**
  - testing for; 257
  - `Timer` problems with; 123
  - `UncaughtExceptionHandler` prevention of; 162–163
- lifecycle
  - performance impact; 116
  - thread-based service management; 150
- overhead
  - in safety testing, strategies for mitigating; 254
- ownership; **150**
- pools; 119–121
  - adding statistics to; 179
  - and work queues; 119
  - application; 167–188
  - as producer-consumer design; 88
  - as thread resource management mechanism; 117
  - callback use in testing; 258
  - creating; 120
  - deadlock risks; 215
  - factory methods for; 171
  - post-construction configuration; 177–179
  - sizing; 170–171
  - task queue configuration; 172–174
- priorities
  - manipulation, liveness risks; 218
- priority
  - when to use; 219
- processes vs.; 2
- queued
  - `SynchronousQueue` management of; 89
- risks of; 5–8
- serial thread confinement; **90**, 90–92
- services that own
  - stopping; 150–161
- sharing
  - necessities and dangers in GUI applications; 189–190
- single
  - sequential task execution; 114
- sources of; 9–11
- starvation deadlock; **169**, 168–169
- suspension
  - costs of; 232, 320
  - `Thread.suspend`, deprecation reasons; 135$_{fn}$
- task
  - execution in; 113–115
  - scheduling, thread-per-task policy; 115
  - scheduling, thread-per-task policy disadvantages; 116
  - vs. interruption handling; 141
- teardown; 171–172
- termination
  - keep-alive time impact on; 172
- thread starvation deadlocks; **215**

thread-local
- *See also* stack, confinement;
- computation
  - role in accurate performance testing; 268

`Thread.stop`
- deprecation reasons; 135_{fn}

`Thread.suspend`
- deprecation reasons; 135_{fn}

`ThreadFactory`; 176_{li}
- customizing thread pool with; 175

`ThreadInfo`
- and testing; 273

`ThreadLocal`; 45–46
- and execution policy; 168
- for thread confinement; 43
- risks of; 46

`ThreadPoolExecutor`
- and untrusted code; 162
- configuration of; 171–179
- constructor; 172_{li}
- extension hooks; 179
- `newTaskFor`; 126_{li}, 148

`@ThreadSafe`; 7, 353

throttling
- as overload management mechanism; 88, 173
- saturation policy use; 174
- `Semaphore` use in `BoundedExecutor` example; 176_{li}

throughput
- *See also* performance;
- as performance testing criteria; 248
- locking vs. atomic variables; 328
- producer-consumer handoff
  - testing; 261
- queue implementations
  - serialization differences; 227
- server application
  - importance of; 113
- server applications
  - single-threaded task execution disadvantages; 114
- thread safety hazards for; 8
- threads benefit for; 3

`Throwable`
- `FutureTask` handling; 98

time/timing
- *See also* deadlock; lifecycle; order/ordering; race conditions;
- -based task
  - handling; 123
  - management design issues; 131–133
- barrier handling based on; 99
- constraints
  - as cancellation reason; 136
  - in puzzle-solving framework; 187
  - interruption handling; 144–145
- deadline-based waits
  - as feature of `Condition`; 307
- deferred computations
  - design issues; 125
- dynamic compilation
  - as performance testing pitfall; 267
- granularity
  - measurement impact; 264
- keep-alive
  - thread termination impact; 172
- `LeftRightDeadlock` example; 207_{fg}
- lock acquisition; 279
- lock scope
  - narrowing, as lock contention reduction strategy; 233–235
- long-running GUI tasks; 195–198
- long-running tasks
  - responsiveness problem handling; 170
- measuring
  - in performance testing; 260–263
  - `ThreadPoolExecutor` hooks for; 179
- performance-based alterations in
  - thread safety risks; 7
- periodic tasks
  - handling of; 123
- progress indication
  - for long-running GUI tasks; 198
- relative vs. absolute
  - class choices based on; 123_{fn}
- response
  - task sensitivity to, execution policy implications; 168
- short-running GUI tasks; 192–195
- thread timeout
  - core pool size parameter impact on; 172_{fn}
- timed locks; 215–216

- weakly consistent iteration semantics; 86

`TimeoutException`
- in timed tasks; 131
- task cancellation criteria; 147

`Timer`
- task-handling issues; 123
- thread use; 9

timesharing systems
- as concurrency mechanism; 2

tools
- *See also* instrumentation; measurement;
- annotation use; 353
- code auditing
  - locking failures detected by; 28$_{fn}$
- heap inspection; 257
- measurement
  - I/O utilization; 240
  - importance for effective performance optimization; 224
  - performance; 230
- monitoring
  - quality assurance use; 273
- profiling
  - lock contention detection; 240
  - performance measurement; 225
  - quality assurance use; 273
- static analysis; 271–273

transactions
- *See also* events;
- concurrent atomicity similar to; 25

transformations
- state
  - in puzzle-solving framework example; 183–188

transition
- *See also* state;
- state transition constraints; **56**
  - impact on safe state variable publication; 69

travel reservations portal example
- as timed task example; 131–133

tree(s)
- *See also* collections;
- models
  - GUI application handling; 200
- traversal
  - parallelization of; 181–182

`TreeMap`
- `ConcurrentSkipListMap` as concurrent replacement; 85

`TreeSet`
- `ConcurrentSkipListSet` as concurrent replacement; 85

Treiber's nonblocking stack algorithm; 331$_{li}$

trigger(ing)
- *See also* interruption;
- JVM abrupt shutdown; 164
- thread dumps; 216

`try-catch` block
- *See also* exceptions;
- as protection against untrusted code behavior; 161

`try-finally` block
- *See also* exceptions;
- and uncaught exceptions; 163
- as protection against untrusted code behavior; 161

`tryLock`
- barging use; 283$_{fn}$
- deadlock avoidance; 280$_{li}$

`trySendOnSharedLine` example; 281$_{li}$

tuning
- *See also* optimization;
- thread pools; 171–179

U

unbounded
- *See also* bounded; constraints; queue(s);
- blocking waits
  - timed vs., in long-running task management; 170
- queues
  - nonblocking characteristics; 87
  - poison pill shutdown use; 155
  - thread pool use of; 173
- thread creation
  - disadvantages of; 116

uncaught exception handlers; 162–163
- *See also* exceptions;

`UncaughtExceptionHandler`; 163$_{li}$
- custom thread class use; 175
- thread leakage detection; 162–163

unchecked exceptions
- *See also* exceptions;
- catching
  - disadvantages of; 161

uncontended
 synchronization; **230**
unit tests
 for BoundedBuffer example; 250
 issues; 248
untrusted code behavior
 See also safety;
 ExecutorService code protection strategies; 179
 protection mechanisms; 161
updating
 See also lifecycle;
 atomic fields; 335–336
 immutable objects; 47
 views
  in GUI tasks; 201
upgrading
 read-write locks; 287
usage scenarios
 performance testing use; 260
user
 See also GUI;
 cancellation request
  as cancellation reason; 136
 feedback
  in long-running GUI tasks; 196_{li}
 interfaces
  threads benefits for; 5
utilization; **225**
 See also performance; resource(s);
 CPU
  Amdahl's law; 225, 226_{fg}
  optimization, as multithreading goal; 222
  sequential execution limitations; 124
 hardware
  improvement strategies; 222

V

value(s)
 See result(s);
variables
 See also encapsulation; state;
 atomic
  classes; 324–329
  locking vs.; 326–329
  nonblocking algorithms and; 319–336
  volatile variables vs.; 39, 325–326
 condition
  explicit; 306–308
 hoisting
  as JVM optimization pitfall; 38_{fn}
 local
  stack confinement use; 44
 multivariable invariant requirements for atomicity; 57
 state
  condition predicate use; 299
  independent; **66**, 66–67
  independent, lock splitting use with; 235
  object data stored in; 15
  safe publication requirements; 68–69
 ThreadLocal; 45–46
 volatile; **38**, 37–39
  atomic variable class use; 319
  atomic variable vs.; 39, 325–326
  multivariable invariants prohibited from; 68
variance
 service time; **264**
Vector
 as safe publication use; 52
 as synchronized collection; 79
 check-then-act operations; 80_{li}, 79–80
 client-side locking management of compound actions; 81_{li}
vehicle tracking example
 delegation strategy; 64
 monitor strategy; 61
 state variable publication strategy; 69–71
 thread-safe object composition design; 61–71
versioned data model; **201**
views
 event handling
  model-view objects; 195_{fg}
 model-view-controller pattern
  deadlock risks; 190
  vehicle tracking example; 61
 reflection-based
  by atomic field updaters; 335
 timeliness vs. consistency; 66, 70
 updating
  in long-running GUI task handling; 201
 with split data models; 201

visibility
See also encapsulation; safety; scope;
condition queue
control, explicit Condition and Lock use; 306
guarantees
JMM specification of; 338
lock management of; 36–37
memory; 33–39
ReentrantLock capabilities; 277
synchronization role; 33
volatile reference use; 49
`vmstat` application
See also measurement; tools;
CPU utilization measurement; 240
performance measurement; 230
thread utilization measurement; 241
`Void`
non-value-returning tasks use; 125
`volatile`
cancellation flag use; 136
final vs.; 158_{fn}
publishing immutable objects with; 48–49
safe publication use; 52
variables; **38**, 37–39
atomic variable class use; 319
atomic variable vs.; 39, 325–326
atomicity disadvantages; 320
multivariable invariants prohibited from; 68
thread confinement use with; 43

W

wait(s)
blocking
timed vs. unbounded; 170
busy-waiting; **295**
condition
and condition predicate; 299
canonical form; 301_{li}
errors, as concurrency bug pattern; 272
interruptible, as feature of `Condition`; 307
uninterruptable, as feature of `Condition`; 307
waking up from, condition queue handling; 300–301
sets; **297**
multiple, as feature of `Condition`; 307
spin-waiting; **232**
as concurrency bug pattern; 273
waiting to run
`FutureTask` state; 95
waking up
See also blocking/blocks; condition, queues; `notify`; sleep; wait;
condition queue handling; 300–301
weakly consistent iterators; **85**
See also iterators/iteration;
web crawler example; 159–161
within-thread usage
See stack, confinement;
within-thread-as-if-serial semantics; **337**
work
queues
and thread pools, as producer-consumer design; 88
in `Executor` framework use; 119
thread pool interaction, size tuning requirements; 173
sharing
deques advantages for; 92
stealing scheduling algorithm; **92**
deques and; 92
tasks as representation of; 113
wrapper(s)
factories
Decorator pattern; 60
synchronized wrapper classes
as synchronized collection classes; 79
client-side locking support; 73