

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; measure-
 ment; 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 docu-
 mentation 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 struc-
 ture(s);

atomic variable; 325

asymmetric two-party tasks

 Exchanger management of; 101

asynchrony/asynchronous

 events, handling; 4

 I/O, and non-interruptable block-
 ing; 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); synchroniza-
 tion; 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_{ii}

condition queue use; 297

test case development for; 248

BoundedBufferTest example; 250_{ii}

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_{ii}

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 im-

plementation 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; concur-
rent/concurrency, collec-
tions; 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; synchroniza-
tion/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, 87ⁱⁱ
 - safe publication use; 52
- ConcurrentModificationException**
 - avoiding; 85
 - fail-fast iterators use; 82–83
- ConcurrentSkipListMap**; 85
- ConcurrentSkipListSet**; 85
- Condition**; 307ⁱⁱ
 - 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; 301ⁱⁱ
 - interruptible, as feature of **Condition**; 307
 - uninterruptible, 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; 304ⁱⁱ
 - 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/contented**
 - 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_{ii}
 deadlock, possibilities; 211
 livelock possibilities; 218
thread
 concurrency mechanisms for; 79

coordination

See also synchronization/synchro-
nized;
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_{fi}

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; encapsu-
lation; 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_{ii}

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_i
 - testing use; 255_{ii}, 260_{ii}

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 advances; 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;

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_{ii}**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; ro-

bustness; 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 struc-

ture(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_{ii}

cancellation

of long-running GUI task; 197

strategy using; 145–147

characteristics of; 95

encapsulation of non-standard can-

cellation 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 block-
ing reason; 148

advantages of; 285

explicit locks vs.; 277–278

intrinsic condition queue relation-
ship 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 require-
ments; 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 synchroniza-
tion 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

ConcurrentModificationExcept-
ion 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; synchroniza-
tion; 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 Concurrent-

HashMap 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 variables; 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; multi-threaded 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 im-
 plementation strategy; **140**

execution

design, influencing factors; **167**
 Executors, for ThreadPoolExec-
 utor configuration; **171**
 implicit couplings between tasks
 and; **167–170**
 parallelism analysis for; **123–133**
 task; **118–119**
 task, application performance
 importance; **113**

interruption; **141**, 141–142saturation; **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 synchroniza-
 tion policy role; **55–56**
 thread safety role; **17**

precondition(s)

See also dependencies, state; invari-
 ant(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-sensitive 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
dequeues 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; representa-
 tion;
 atomic arrays; 325
 binary semaphores; 99
 final fields; 48
 of interruption; 93
 of multithreaded environments
 ThreadLocal variable considera-
 tions; 46
 reentrant locking; 26–27
 ReentrantLock capabilities; 278
 ReentrantReadWriteLock capa-
 bilities; 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 manage-
 ment 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 Concurrent-HashMap; 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
 - lifecycle
 - 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*i*
- 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
 - and execution 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_{fi}
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_{ii}

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_{fi}
deadlock avoidance; 280_{ii}

trySendOnSharedLine example; 281_{ii}

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_{ii}

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_{ii}
 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_{fi}

 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 prohib-
 ited from; 68

variance

 service time; 264

Vector

 as safe publication use; 52
 as synchronized collection; 79
 check-then-act operations; 80_{ii}, 79–
 80
 client-side locking management of
 compound actions; 81_{ii}

vehicle tracking example

 delegation strategy; 64
 monitor strategy; 61
 state variable publication strategy;
 69–71
 thread-safe object composition de-
 sign; 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 han-
 dling; 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 prohib-
 ited 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_{fi}
 errors, as concurrency bug pat-
 tern; 272
 interruptible, as feature of Con-
 dition; 307
 uninterruptable, as feature of
 Condition; 307
 waking up from, condition
 queue handling; 300–301
 sets; 297

multiple, as feature of Condi-
 tion; 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 tun-
 ing 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