## Java Concurrency In Practice

## Chapter 2: Thread safety

* If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken. 3 ways to fix it:
  + Don't share the state variable across threads
  + Make the state variable immutable
  + Use synchronization whenever accessing the state variable.
* When designing thread safe classes, good object oriented techniques encapsulation, immutability, and clear specification of invariants are your best friends.

## 2.1 What is thread safety?

* it is always a good practice first to make your code right, and then make it fast.
* A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.
* Thread safe classes encapsulate any needed synchronization so that clients need not provide their own.
* Stateless objects are always thread safe.

## 2.2. Atomicity

2.2.1 Race Condition

2.2.3 Compound Actions

* Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state.
* Where practical, use existing thread safe objects, like AtomicLong, to manage your class's state. It is simpler to reason about the possible states and state transitions for existing thread safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.

## 2.3 Locking

* To preserve state consistency, update related state variables in a single atomic operation.

2.3.1 Intrinsic locks: synchronized

2.3.2 Reentrancy: intrinsic locks are reentrant, if a thread tries to acquire a lock that it already holds, the request succeeds

## 2.4 Guarding state with locks

* For each mutable state variable that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held. In this case, we say that the variable is guarded by that lock.
* Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.
* For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by the same lock.

## 2.5 Liveness and Performance

* There is frequently a tension between simplicity and performance. When implementing a synchronization policy, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance.
* Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O.

## Chapter 3: Sharing Objects

## 3.1 Visibility

* No guarantee that the reading thread will see a value written by another thread on a timely basis, or even at all
* Reordering: In the absence of synchronization, the compiler, processor, and runtime can do some downright weird things to the order in which operations appear to execute. Attempts to reason about the order in which memory actions "must" happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect.

3.1.1 Stale data

3.1.2 Non-atomic 64-bit operations

* For non-volatile long and double variables, the JVM is permitted to treat a 64 bit read or write as two separate 32 bit operations. If the reads and writes occur in different threads, it is therefore possible to read a nonvolatile long and get back the high 32 bits of one value and the low 32 bits of another.

3.1.3 Locking and visibility

* Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads see the most up to date values of shared mutable variables, the reading and writing threads must synchronize on a common lock.

3.1.4 Volatile variables

* When a field is declared volatile, the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with other memory operations. Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread.
* Use volatile variables only when they simplify implementing and verifying your synchronization policy; avoid using volatile variables when verifying correctness would require subtle reasoning about visibility. Good uses of volatile variables include ensuring the visibility of their own state, that of the object they refer to, or indicating that an important lifecycle event (such as initialization or shutdown) has occurred.
* Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility. Not strong enough to make the increment operation (count++) atomic, unless you can guarantee that the variable is written only from a single thread.
* **You can use volatile variables only when all the following criteria are met:** 
  + Writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value;
  + The variable does not participate in invariants with other state variables; and
  + Locking is not required for any other reason while the variable is being accessed.

## 3.2 Publication and escape

* Publishing an object means making it available to code outside of its current scope
* An object that is published when it should not have been is said to have escaped.

// Allowing Internal Mutable State to Escape. Don't Do this.

class UnsafeStates {

private String[] states = new String[] {

"AK", "AL" ...

};

public String[] getStates() { return states; }

}

// Implicitly Allowing this Reference to Escape. Don't Do this.

public class ThisEscape {

public ThisEscape(EventSource source) {

source.registerListener(

.//// Escaped when construction not finished

new EventListener() {

public void onEvent(Event e) {

doSomething(e);

}

});

} }

3.2.1 Safe construction practices

* Do not allow this reference to escape during construction.
* A common mistake that can let this reference escape during construction is to start a thread from a constructor. There's nothing wrong with creating a thread in a constructor, but it is best not to start the thread immediately.
* Avoid the improper construction by using a private constructor and a public factory method

public class SafeListener {

private final EventListener listener;

private SafeListener() {

listener = new EventListener() {

public void onEvent(Event e) {

doSomething(e);

} };

}

public static SafeListener newInstance(EventSource source) {

SafeListener safe = new SafeListener();

source.registerListener(safe.listener);

return safe;

} }

## 3.3 Thread confinement

* Thread confinement to achieve thread safety: If data is only accessed from a single thread, no synchronization is needed.
* Sample usage: UI thread. JDBC thread pool

3.3.1 Ad-hoc thread confinement

* Maintaining thread confinement falls entirely on the implementation
* Volatile variables: Safe to perform read-modify-write operations on shared volatile variables as long as can ensure that the volatile variable is only written from a single thread. Confining the modification to a single thread to prevent race conditions.

3.3.2 Stack confinement

* an object can only be reached through local variables

3.3.3 ThreadLocal

* Allows to associate a per thread value with a value holding object
* Often used to prevent sharing in designs based on mutable Singletons or global variables.
* Like global variables, thread local variables can detract from reusability and introduce hidden couplings among classes, and should therefore be used with care.

## 3.4 Immutability

* Immutable objects are always thread safe.
* An object is immutable if:
  + Its state cannot be modified after construction;
  + All its fields are final;
  + It is properly constructed (this reference does not escape during construction)

3.4.1 Final fields

* A good practice to make all fields final unless they need to be mutable.

3.4.2 Using volatile to public immutable objects

class Cache { private final BigInteger someData;

Cache(BigInteger data) { someData = data; }

BigInteger getData() { return someData; }

}

private **volatile** Cache cache;BigInteger data = cache.getData();

If (data == null) { // Get some data

cache = new Cache(data);

}

## 3.5 Safe publication

3.5.1 Improper publication: When good objects go bad

3.5.2 Immutable objects and initialization safety

* Immutable objects can be used safely by any thread without additional synchronization, even when synchronization is not used to publish them.

3.5.3 Safe publication idioms

* To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time. A properly constructed object can be safely published by:
  + Initializing an object reference from a static initializer;
  + Storing a reference to it into a volatile field or AtomicReference;
  + Storing a reference to it into a final field of a properly constructed object;
  + Storing a reference to it into a field that is properly guarded by a lock.

3.5.4 Effectively immutable objects

* Objects that are not technically immutable, but whose state will not be modified after publication, are called effectively immutable
* Safely published effectively immutable objects can be used safely by any thread without additional synchronization.

3.5.5 Mutable objects

* The publication requirements for an object depend on its mutability:
  + Immutable objects can be published through any mechanism;
  + Effectively immutable objects must be safely published;
  + Mutable objects must be safely published, and must be either thread safe or guarded by a lock.

3.5.6 Sharing objects safely

* **The most useful policies for using and sharing objects in a concurrent program are:** 
  + **Thread confined.** A thread confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.
  + **Shared read only.** A shared read only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read only objects include immutable and effectively immutable objects.
  + **Shared thread safe.** A thread safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.
  + **Guarded.** A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread safe objects and published objects that are known to be guarded by a specific lock.

## Chapter 4: Composing objects

## 4.1 Designing a thread-safe class

* The design process for a thread safe class should include these three basic elements:
  + Identify the variables that form the object's state;
  + Identify the invariants that constrain the state variables;
  + Establish a policy for managing concurrent access to the object's state.

4.1.1 Gathering synchronization requirements

* Thread safety cannot be ensured without understanding an object's invariants and post conditions. Constraints on the valid values or state transitions for state variables can create atomicity and encapsulation requirements.

4.1.2 State-dependent operations

* Operations with state based preconditions are called state dependent
* Concurrent programs add the possibility of waiting until the precondition becomes true, and then proceeding with the operation.

4.1.3 State ownership

* When defining which variables form an object's state, we want to consider only the data that object owns. Ownership is not embodied explicitly in the language, but is instead an element of class design.

## 4.2 Instance confinement

* Encapsulation simplifies making classes thread safe by promoting instance confinement, often just called confinement
* Encapsulating data within an object confines access to the data to the object's methods, making it easier to ensure that the data is always accessed with the appropriate lock held.
* Confinement makes it easier to build thread safe classes because a class that confines its state can be analyzed for thread safety without having to examine the whole program.

 4.2.1 The Java monitor pattern

* Synchronized method

## 4.3 Delegating thread safety

4.3.2 Independent state variables

* If a class is composed of multiple independent thread safe state variables and has no operations that have any invalid state transitions, then it can delegate thread safety to the underlying state variables.

4.3.3 When delegation fails

* If a class has compound actions, delegation alone is again not a suitable approach for thread safety. In these cases, the class must provide its own locking to ensure that compound actions are atomic, unless the entire compound action can also be delegated to the underlying state variables.
* Sample: Range: setLower, setUpper,. lower must be < upper

4.3.4 Publishing underlying state variables

* If a state variable is thread safe, does not participate in any invariants that constrain its value, and has no prohibited state transitions for any of its operations, then it can safely be published.

## 4.4 Adding Functionality to Existing Thread safe Classes

* The safest way to add a new atomic operation is to modify the original class
* Another approach is to extend the class

Extension is fragile than adding code directly to a class, because the implementation of the synchronization policy is now distributed over multiple, separately maintained source files. If the underlying class policy is changed, the subclass will break.

4.4.1 Client-side locking

* client side locking is even more fragile because it entails putting locking code for class C into classes that are totally unrelated to C

4.4.2 Composition

## 4.5 Document synchronization policies

* Document a class's thread safety guarantees for its clients; document its synchronization policy for its maintainers.

## Chapter 5: Building blocks

## 5.1 Synchronized blocks

5.1.1 Problem with synchronized collections

* Sometimes need to use additional client side locking to guard compound actions, including iteration, navigation and conditional operations such as put-if-absent

5.1.2 Iterators and ConcurrentModificationException

* fast-fails are implemented by associating a modification count with the collection: if the modification count changes during iteration, hasNext or next throws ConcurrentModificationException. However, this check is done without synchronization, so there is a risk of seeing a stale value of the modification count and therefore that the iterator does not realize a modification has been made.
* This was a deliberate design tradeoff to reduce the performance impact of the concurrent modification detection code

5.1.3 Hidden iterators

* Collections toString iterates all the objects
* Just as encapsulating an object's state makes it easier to preserve its invariants, encapsulating its synchronization makes it easier to enforce its synchronization policy.

## 5.2 Concurrent collections

* Replacing synchronized collections with concurrent collections can offer dramatic scalability improvements with little risk.

5.2.1 ConcurrentHashMap

* weakly consistent iterators: Collections which rely on CAS (*compare-and-swap*) have weakly consistent iterators, which reflect some but not necessarily all of the changes that have been made to their backing collection since they were created. For example, if elements in the collection have been modified or removed before the iterator reaches them, it definitely will reflect these changes, but no such guarantee is made for insertions.
* Cannot be locked for exclusive access

5.2.2 Additional atomic map operations

5.2.3 CopyOnWriteArrayList

* Fail safe iterator iterator mechanism makes a copy of the internal Collection data structure and uses it to iterate over the elements. This prevents any concurrent modification exceptions from being thrown if the underlying data structure changes. Of course, the overhead of copying the entire array is introduced.

## 5.3 Blocking queues and the producer-consumer pattern

* Blocking queues provide blocking put and take methods as well as the timed equivalents offer and poll.
  + If the queue is full, put blocks until space becomes available;
  + if the queue is empty, take blocks until an element is available.
  + Queues can be bounded or unbounded; unbounded queues are never full, so a put on an unbounded queue never blocks.
* Blocking queues support the producer consumer design pattern.
  + If the producers don't generate work fast enough to keep the consumers busy, the consumers just wait until more work is available.
  + If the producers consistently generate work faster than the consumers can process it, eventually the application will run out of memory because work items will queue up without bound
* The producer consumer pattern also enables several performance benefits. Producers and consumers can execute concurrently; if one is I/O bound and the other is CPU bound, executing them concurrently yields better overall throughput than executing them sequentially.
* Bounded queues are a powerful resource management tool for building reliable applications: they make your program more robust to overload by throttling activities that threaten to produce more work than can be handled.
* BlockingQueue: LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, SynchronousQueue

5.3.2 Serial thread confinement

* For mutable objects, producer consumer designs and blocking queues facilitate serial thread confinement for handing off ownership of objects from producers to consumers. A thread confined object is owned exclusively by a single thread, but that ownership can be "transferred" by publishing it safely where only one other thread will gain access to it and ensuring that the publishing thread does not access it after the handoff. The safe publication ensures that the object's state is visible to the new owner, and since the original owner will not touch it again, it is now confined to the new thread. The new owner may modify it freely since it has exclusive access.

5.3.3 Deques and work stealing

* D.eque and BlockingDeque: extend Queue and BlockingQueue
  + A Deque is a double ended queue that allows efficient insertion and removal from both the head and the tail. ArrayDeque, LinkedBlockingDeque
* Work stealing pattern: A producer consumer design has one shared work queue for all consumers; in a work stealing design, every consumer has its own deque. If a consumer exhausts the work in its own deque, it can steal work from the tail of someone else's deque. Work stealing can be more scalable than a traditional producer consumer design because workers don't contend for a shared work queue; most of the time they access only their own deque, reducing contention. When a worker has to access another's queue, it does so from the tail rather than the head, further reducing contention.

## 5.4 Blocking and interruptible methods

* When your code calls a method that throws InterruptedException, then your method is a blocking method too, and must have a plan for responding to interruption. For library code, there are basically two choices:
  + Propagate the InterruptedException to your caller
  + Restore the interrupt

try {

processTask(queue.take());

} catch (InterruptedException e) {

// restore interrupted status

Thread.currentThread().interrupt();

}

## 5.5 Synchronizers

5.5.1 Latch

* A latch is a synchronizer that can delay the progress of threads until it reaches its terminal state
* Latches can be used to ensure that certain activities do not proceed until other one time activities complete
  + Ensuring that a computation does not proceed until resources it needs have been initialized.
  + Ensuring that a service does not start until other services on which it depends have started.
  + Waiting until all the parties involved in an activity, for instance the players in a multi player game, are ready to proceed
* Two common usages: Start gate and end gate

final CountDownLatch startGate = new **CountDownLatch(1);**

final CountDownLatch endGate = new **CountDownLatch(nThreads);**

for (int i = 0; i < nThreads; i++) {

Thread t = new Thread() {

public void run() {

try {

**startGate.await();**

try {

task.run();

} finally {

**endGate.countDown();**

}

} catch (InterruptedException ignored) { }

}

};

t.start();

}

long start = System.nanoTime();

**startGate.countDown();**

**endGate.await();**

long end = System.nanoTime();

return end-start;

5.5.2 FutureTask

* FutureTask implements Future, which describes an abstract result bearing computation

5.5.3 Semaphores

* Counting semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time. Counting semaphores can be used to implement resource pools or to impose a bound on a collection
* A Semaphore manages a set of virtual permits;
  + the initial number of permits is passed to the Semaphore constructor.
  + Activities can acquire permits (as long as some remain) and release permits when they are done with them.
  + If no permit is available, acquire blocks until one is (or until interrupted or the operation times out).
  + The release method returns a permit to the semaphore.
* A degenerate case of a counting semaphore is a binary semaphore, a Semaphore with an initial count of one.
* A binary semaphore can be used as a mutex with non reentrant locking semantics; whoever holds the sole permit holds the mutex.

5.5.4 Barriers

* Barriers are similar to latches in that they block a group of threads until some event has occurred.
  + The key difference is that with a barrier, all the threads must come together at a barrier point at the same time in order to proceed.
  + Latches are for waiting for events; barriers are for waiting for other threads.
* CyclicBarrier allows a fixed number of parties to rendezvous repeatedly at a barrier point and is
  + useful in parallel iterative algorithms that break down a problem into a fixed number of independent sub-problems.
  + Threads call await when they reach the barrier point, and await blocks until all the threads have reached the barrier point.
  + If all threads meet at the barrier point, the barrier has been successfully passed, in which case all threads are released and the barrier is reset so it can be used again.
  + If a call to await times out or a thread blocked in await is interrupted, then the barrier is considered broken and all outstanding calls to await terminate with BrokenBarrierException.
  + If the barrier is successfully passed, await returns a unique arrival index for each thread, which can be used to "elect" a leader that takes some special action in the next iteration.
  + CyclicBarrier also lets you pass a barrier action to the constructor; this is a Runnable that is executed (in one of the subtask threads) when the barrier is successfully passed but before the blocked threads are released.
* Another form of barrier is Exchanger, a two party barrier in which the parties exchange data at the barrier point.
  + Useful when the parties perform asymmetric activities, for example when one thread fills a buffer with data and the other thread consumes the data from the buffer; these threads could use an Exchanger to meet and exchange a full buffer for an empty one. When two threads exchange objects via an Exchanger, the exchange constitutes a safe publication of both objects to the other party.

## 5.6. Building an Efficient, Scalable Result Cache

## Summary:

* All concurrency issues boil down to coordinating access to mutable state. The less mutable state, the easier it is to  ensure thread safety.
* Make fields final unless they need to be mutable.
* Immutable objects are automatically thread safe. Immutable objects simplify concurrent programming tremendously. They are simpler and safer, and can be shared freely without locking or defensive copying.
* Encapsulation makes it practical to manage the complexity.
* Guard each mutable variable with a lock.
* Guard all variables in an invariant with the same lock.
* Hold locks for the duration of compound actions.
* A program that accesses a mutable variable from multiple threads without synchronization is a broken program.
* Don't rely on clever reasoning about why you don't need to synchronize.
* Include thread safety in the design processor explicitly document that your class is not thread safe.
* Document your synchronization policy.

## Chapter 6: Task execution

## 6.1 Executing tasks in threads

6.1.1 Executing Tasks Sequentially

6.1.2 Explicitly Creating Threads for Tasks

6.1.3 Disadvantages of Unbounded Thread Creation

## 6.2. The Executor Framework

6.2.2 Execution Policies

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

6.2.3 Thread pools

6.2.4 Executor Lifecycle

6.2.5 Delayed and Periodic Tasks

* Timer facility manages the execution of deferred and periodic tasks. However, Timer has some drawbacks, and ScheduledThreadPoolExecutor should be thought of as its replacement.
* A Timer creates only a single thread for executing timer tasks. If a timer task takes too long to run, the timing accuracy of other TimerTasks can suffer.

## 6.3. Finding Exploitable Parallelism

6.3.2 Result bearing Tasks: Callable and Future

* Runnable is a fairly limiting abstraction; run cannot return a value or throw checked exceptions, although it can have side effects such as writing to a log file or placing a result in a shared data structure.
* Future represents the lifecycle of a task and provides methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task.

6.3.4 Limitations of Parallelizing Heterogeneous Tasks

* A further problem with dividing heterogeneous tasks among multiple workers is that the tasks may have disparate sizes.

6.3.5 CompletionService: Executor Meets BlockingQueue

6.3.7 Placing Time Limits on Tasks

## Chapter 7: Cancellation and shutdown

## 7.1 Task cancellation

7.1.1 Interruption

* There is nothing in API or language specification that ties interruption to any specific cancellation semantics, but in practice, using interruption for anything but cancellation is fragile and difficult to sustain in larger applications
* Calling interrupt does not necessarily stop the target thread from doing what it is doing; it merely delivers the message that interruption has been requested
* Interruption is usually the most sensible way to implement cancellation.
* If calling interrupted returns true, unless planning to swallow the interruption, should throw InterruptedException or restore the interrupted status by calling interrupt again.

Thread.currentThread().interrupt();

7.1.2 Interruption policies

* Because each thread has its own interruption policy, you should not interrupt a thread unless you know what interruption means to that thread.
* A thread should be interrupted only by its owner.

7.1.3 Responding to interruption

* Only code that implements a thread’s interruption policy may swallow an interruption request. General-purpose task and library code should never swallow interruption requests.

7.1.5 Cancellation via Future

7.1.6 Dealing with Non-interruptible blocking

* Depends on the function implementation

7.1.7 Encapsulating nonstandard cancellation with Newtaskfor

## 7.2 Stopping a thread-based service

* Provide lifecycle methods whenever a thread-owning service has a lifetime longer than that of the method that created it.
* shutdown/shutdownNow/awaitTermination method
* Poison pills method: Place a recognizable object on the queue to mean “when get this, stop”
* Limitations of ShutdownNow: No general way to find out which task started but not finished

## 7.3 Handling abnormal thread termination

7.3.1 UncaughtExceptionHandler

## 7.4 JVM shutdown

7.4.1 Shutdown hooks

Runtime.addShutdownHook

7.4.2 Daemon threads

* Threads are divided into two types: Normal threads and daemon threads
* When JVM starts up, all the threads it creates (such as GC and other housekeeping threads) are damon threads, except the main thread. When a new thread is created, it inherits the daemon status of the thread that created it, so by default any thread created by the main thread are also normal threads.
* Daemon threads are not a good substitute for properly managing the lifecycle of services within an application

7.4.3 Finalizers

Avoid finalizers

## Chapter 8: Applying thread pools

## 8.1 Implicit couplings between tasks and execution policies

* Types of tasks that require specific execution policies:
  + Dependent tasks
  + Tasks that exploit thread confinement
  + Response-time-sensitive tasks
  + Tasks that use ThreadLocal
* Thread pools work best when tasks are homogeneous and independent.
* Tasks that depend on other tasks require that the thread pool be large enough that tasks are never queued or rejected.
* Tasks that exploit thread confinement require sequential execution.

8.1.1 Thread starvation deadlock

* All threads are executing tasks that are blocked waiting for other tasks still on the work queue
* Whenever submitting to an executor tasks that are not independent, be aware of the possibility of thread starvation deadlock, and document any pool sizing or configuration constraints in the code or configuration file where the executor is configured
* There may also be implicit limits because of constraints on other resources, eg. JDBC connection pool with 10 connections and each tasks require a connection.

8.1.2 Long-running tasks

* One technique that can mitigate the ill effects of long-running tasks is for tasks to use timed resource waits instead of unbounded waits, eg wait with timeout.

## 8.2 Sizing thread pools

* Thread pool sizes should be provided by a configuration mechanism or computed dynamically by consulting Runtime.availableProcessors
* For compute-intensive tasks, an N processor system usually achieves optimum utilization with a thread pool of N + 1 threads.
* For tasks that also include I/O or other blocking operation, a larger pool may be wanted, since not all of the threads will be schedulable at all times.
* Other resources that can contribute to sizing constraints are memory, file handles, socket handles, and database connections, etc.

## 8.3 Configuring ThreadPoolExecutor

8.3.1 Thread creation and teardown

* The core pool size, maximum pool size, and keep-alive time govern thread creation and teardown

8.3.2 Managing queued tasks

* Unbounded thread creation could lead to instability.
* newCachedThreadPool factory is a good default choice for an executor, providing better queuing performance than a fixed thread pool. A fixed thread pool is a good choice when need to limit the number of concurrent tasks for resource-management purposes, as in a server application that accepts requests from network clients and would otherwise be vulnerable to overload.

8.3.3 Saturation policies

* AbortPolicy, CallerRunsPolicy, DiscardPolicy and DiscardOldestPolicy

8.3.4 Thread factories

8.3.5 Customizing ThreadPoolExecutor after construction

## 8.4 Extending ThreadPoolExecutor

## 8.5 Parallelizing recursive algorithms

## Chapter 9: GUI Applications

## 9.1 Why are GUIs single threaded?

9.1.1 Sequential event processing

9.1.2 Thread confinement in Swing

* Swing single-thread rule: Swing components and models should be created, modified and queried only from the event-dispatching thread

## 9.2 Short-running GUI tasks

## 9.3 Long-running GUI tasks

9.3.1 Cancellation

9.3.2 Progress and completion indication

## 9.4 Shared data models

9.4.1 Thread-safe data models

9.4.2 Split data models

## 9.5 Other forms of single-threaded subsystem