# JAVA CONCURRENCY IN PRACTICE

## Chapter 1 - Introduction

Threads allow multiple streams of program control flow to coexist within a process. Thew share process-wide resources such as memory and file handles, but each thread has its own program counter, stack, and local variables. Multiple threads within the same program can be scheduled simultaneously on multiple CPUs.

Since threads share the memory address space of their owning process, all threads within a process have access to the same variables and allocate objects from the same heap, which allows finer-grained data sharing than inter-process mechanisms. But without explicit synchronization to coordinate access to shared data, a thread may modify variables that another thread is in the middle of using, with unpredictable results.

### 1.2 Benefits of threads

Threads can reduce development and maintenance costs and improve the performance of complex applications. Threads turn asynchronous workflows into most sequential ones.

#### 1.2.2 Simplicity of modeling

A program that processes one type of task sequentially is simpler to write, less error-prone, and easier to test than one managing multiple different types of tasks at once. A complicated, asynchronous workflow can be decomposed into a number of simpler, synchronous workflows each running in a separate thread, interacting only with each other at specific synchronization points.

### 1.3

#### 1.3.1

//Listing 1.1:

public class UnsafeSequence{

private int value;

public int getNext(){

return value++;

}

}

PROBLEMS:

With some unlucky timing, two threads could call getNext and receive the same value.

nextValue++ may appear to be a single operation, but is in fact three separate operations: read the value, add one to it, and write out the new value.

##### Solução:

public class Sequence {

private int nextValue;

public synchronized int getNext() {

return nextValue++;

}

}

### 1.3.2

#### 1.3.3

Context switches – when the scheduler suspends the active thread temporarily so another thread can run – are more frequent in applications with many threads, and have significant costs

### 1.4 Threads are everywhere

Every java application uses threads. When the JVM starts, it creates threads for JVM housekeeping (garbage collection, finalization) and a main thread for running the main method.

The Servlets framework is designed to handle all the infrastructure of deploying a web application and dispatching to handle all the infrastructure of deploying a web application and dispatching requests from remote HTTP clients. => servlets need to be thread-safe

## PART I Fundamentals

## Chapter 2 – Thread Safety

Writing thread-safe code is, at its core, about managing access to state, and in particular to shared, mutable state.

Informally, an object’s state is its data, stored in state variables such as instance or static fields.

By shared we mean that a variable could be accessed by multiple threads; by mutable, we mean that its value could change during its lifetime.

* Protect data from uncontrolled concurrent access

Whether an object need to be thread-safe depends on whether it will be accessed from multiple threads.

The primary mechanism for synchronization in Java is **synchronized** keyword, which provides exclusive locking, but the term “synchronization” also includes the use of **volatile variables, explicit locks, and atomic variables.**

If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken. There are three ways to fix it:

* Don’t share the state variable across threads;
* Make the state variable *immutable*;
* Use *synchronization* whenever accessing the state variable.

Encapsulation, immutability and clear specificarion

### 2.1 What is thread safety?

A class is thread-safe when it continues to behave 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.

public class StatelessFactorizer implements Servlet {

public void service(ServletRequest req, ServletResponse resp) {

BigInteger i = extractFromRequest(req);

BigInteger[] factors = factor(i);

encodeIntoResponse(resp, factors);

}

}

This servlet is stateless: it has no fields and references no fields from other classes.

***Stateless objects are always thread-safe.***

### 2.2 Atomicity

#### 2.2.1 Race conditions

The most common type of race condiition is ***check-then-act,*** where a potentially stale observation is used to make a decision on what to do next.

Check-then-act: you observe something to be true (file X doesn’t exist) and then take action based on that observation(create X); but in fact the observation could have become invalid between the time you observed it and the time you acted on it (someone else created file X in the meantime), causing a problem (unexpected exception, overwritten data, file corruption).

#### 2.2.2 Example: race conditions in lazy initialization

A common idom that uses check-then-act is lazy initialization.

public class LazyInitRace {

private ExpensiveObject instance = null;

public ExpensiveObject getInstance() {

if(instance == null) {

instance = new ExpensiceObject();

}

return instance;

}

}

### 2.3 Locking

#### 2.3.1 Intrinsic locks

Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block. A synchronized block has two parts:

* A reference to an object that will serve as the lock
* A block of code to be guarded by that lock

Synchronized (lock) {

//Access or modify shared state guarded by lock

}

Every java object can implicity act as a lock for purposes of synchronization; these built-in locks are called *instrinsic locks* or *monitor locks.*

The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block, whether by the normal control path or by throwing an exception out of the block.

Intrinsic locks in Java act as *mutexes(*or *mutual exclusion locks),* which means that at most one thread may own the lock. When thread A attempts to acquire a lock held by thread B, A must wait, or block, until B releases it. If B never releases the lock, A waits forever.

#### 2.3.2 Reentrancy

Intrinsic locks are *reentrant,* if a thread tries to acquire a lock that it already holds, the request succeeds. Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.

### 2.4 Guarding state with locks

Compound actions on shared state, such as incrementing a hit counter(read-modify-write) or lazy initialization (check-then-act), mut be made atomic to avoid race conditions.

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 thar lock.

Acquiring the lock associated with an object does *not*prevent other threads from accessing that object – the only thing that acquiring a lock prevents any other thread from doing is acquiring that same lock.

Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.

A common locking convention is to encapsulate all mutable state within an object and to protect it from concurrent access by synchronizing any code path that accesses mutable state using the object intrinsic lock.

Not all data needs to be guarded by locks – only mutable data that will be accessed from multiple threads.

When a variable is guarded by a lock you’ve ensured that only one thread at a time can be access that variable.

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

@ThreadSafe

Public class CachedFactorizer implements Servler {

@GuarderBy(“this”) private BigInteger lastNumber;

@GuarderBy(“this”) private BigInteger[] lastFactors;

@GuarderBy(“this”) private long hits;

@GuarderBy(“this”) private long cacheHits;

Public synchronized long getHits() { return this; }

Public synchronizes double getCacheHitRatio() {

Return (double) cacheHits / (double) hits;

}

Public void service(ServletRequest req, ServletResponse resp) {

BigInteger i = extractFromRequest(req);

BigInteger[] factors = null;

Synchronized (this) {

(…)

Synchronized(this) {

(…)

}

}

}

}

Two separate synchronized blocks. One guards the check-then-act sequence that tests whether we can just return the cached result, and the other guards updating both the cached number and the cached factors.

## Chapter 3 – *Sharing Objects*

Synchronization also has another significant aspect: *memory visibility*

### 3.1 Visibility

#### 3.1.1 stale data

Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops.

#### 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 shares mutable variables, the reading and writing threads must synchronize on a common block.

#### 3.1.4 Volatile variables

Volatile variables – an alternative, weaker form of synchronization, to ensure that updates to a variable are propagated predictably to other threads.

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

The most common use for volatile variables is as a completion, interruption, or status flag.

Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.

You can use volatile variables only when all the following criteria are met:

* Writes to the variables 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;
* Locking is not required for any other reason while the variable is being accessed.

#### 3.2.1

Mistake: when an object creates a thread from its constructor, it almost always shares its this reference with the new reference with the new thread. There is nothing wrong with crearing a thread in a constructor, but it is best not to start the thread immediately.

### 3.3 Thread confinement

If data is only accessed from a single thread, no synchronization is needed.

#### 3.3.1 Ad-hoc thread confinement

Ad-hoc thread confinement describes when the responsibility for maintaining thread confinement falls entirely on the implementation.

A special case of thread confinement applies to volatile variables. It is safe to perform read-modify-write operations on shared volatile variables as long as u ensure that the volatile variable is only written from a single thread.

Pagina 44

#### 3.3.2 Stack confinement

*Stack confinement* is a special case of thread confinement in which an object can only be reached through local variables. Just as encapsulation can make it easier to preserve invariants, local variables can make it easier to confine objects to a thread. Local variables are intrinsically confined to the executing thread; they exist on the executing thread’s stack, which is not accessible to other threads. Stack is simpler to maintain and less fragile than ad-hoc thread confinement.

public int loadTheArk(Collection<Animal> candidates) {

SortedSet<Animal> animals;

int numPairs = 0;

Animal candidate = null;

// animals confined to method, don’t let them escape!

animals = new TreeSet<Animal>(new SpeciesGenderComparator());

animals.addAll(candidates);

for (Animal a : animals) {

if (candidate == null || !candidate.isPotentialMate(a)) candidate = a;

else {

ark.load(new AnimalPair(candidate, a)); ++numPairs;

candidate = null;

}

}

return numPairs;

}

. There is no way to obtain a reference to a primitive variable, so the language semantics ensure that primitive local variables are always stack confined.

#### 3.3.3 ThreadLocal

ThreadLocal allows you to associate a per-thread value with a value-holding object. Thread- Local provides get and set accessor methods that maintain a separate copy of the value for each thread that uses it, so a get returns the most recent value passed to set from the currently executing thread.

private static ThreadLocal<Connection> connectionHolder

= new ThreadLocal<Connection>() {

public Connection initialValue() {

return DriverManager.getConnection(DB\_URL);

}

};

public static Connection getConnection() {

return connectionHolder.get();

}

#### 3.4 Immutability

An immutable object is one whose state cannot be changed after construction. Immutable objects are inherently **thread-safe**; their invariants are established by the constructor, and if their state cannot be changed, these invariants always hold.

Immutable objects are **simple**. They can only be in one state, which is carefully controlled by the constructor.

Immutable objects are also **safer**.

An object whose fields are all **final** may still be mutable, since final fields can hold references to mutable objects.

An object is immutable if:

* Its state cannot be modified after construction;
* All its fields are final and
* It is properly constructed (the this reference does not escape during construction).

@Immutable

public final class ThreeStooges {

private final Set<String> stooges = new HashSet<String>();

public ThreeStooges() {

stooges.add("Moe");

stooges.add("Larry");

stooges.add("Curly");

}

public boolean isStooge(String name) {

return stooges.contains(name);

}

}

* Immutable class built out of mutable underlying objects.
* While the Set that stores the names is mutable, the design of ThreeStooges makes it impossible to modify that Set after construction.

There is a difference between an object being immutable and the reference to it being immutable.

#### 3.4.1 Final fields

Final fields can’t be modified. It is the use of final fields that makes possible the guarantee of initialization safety that lets immutable objects be freely accessed and shared without synchronization. Declaring fields final also documents to maintainers that these fields are not expected to change.

## Chapter 4 – Composing Objects

### 4.1 Designing a thread-safe class

While it is possible to write a thread-safe program that stores all its state in public static fields, it is a lot harder to verify its thread safety or to modify it so that it remains thread-safe than one that uses encapsulation appropriately. Encapsulation makes it possible to determine that a class is thread-safe without having to examine the entire program.

An object’s state starts with its fields. If they are all of primitive type, the fields comprise the entire state.

@ThreadSafe

public final class Counter {

@GuardedBy("this") private long value = 0;

public synchronized long getValue() {

return value;

}

public synchronized long increment() {

if (value == Long.MAX\_VALUE)

throw new IllegalStateException("counter overflow");

return ++value;

}

}

* Simple thread-safe counter using the Java monitor pattern.

#### 4.1.1 Gathering synchronization requirements

Making a class thread-safe means ensuring that its invariants hold under concur- rent access; this requires reasoning about its state. Objects and variables have a state space: the range of possible states they can take on. The smaller this state space, the easier it is to reason about. By using final fields wherever practical, you make it simpler to analyze the possible states an object can be in.

### 4.2 Instance confinement

@ThreadSafe

public class PersonSet {

@GuardedBy("this")

private final Set<Person> mySet = new HashSet<Person>();

public synchronized void addPerson(Person p) {

mySet.add(p);

}

public synchronized boolean containsPerson(Person p) {

return mySet.contains(p);

}

}

* Using confinement to ensure thread safety.

### 4.2.1 The Java monitor pattern

Pagina 60