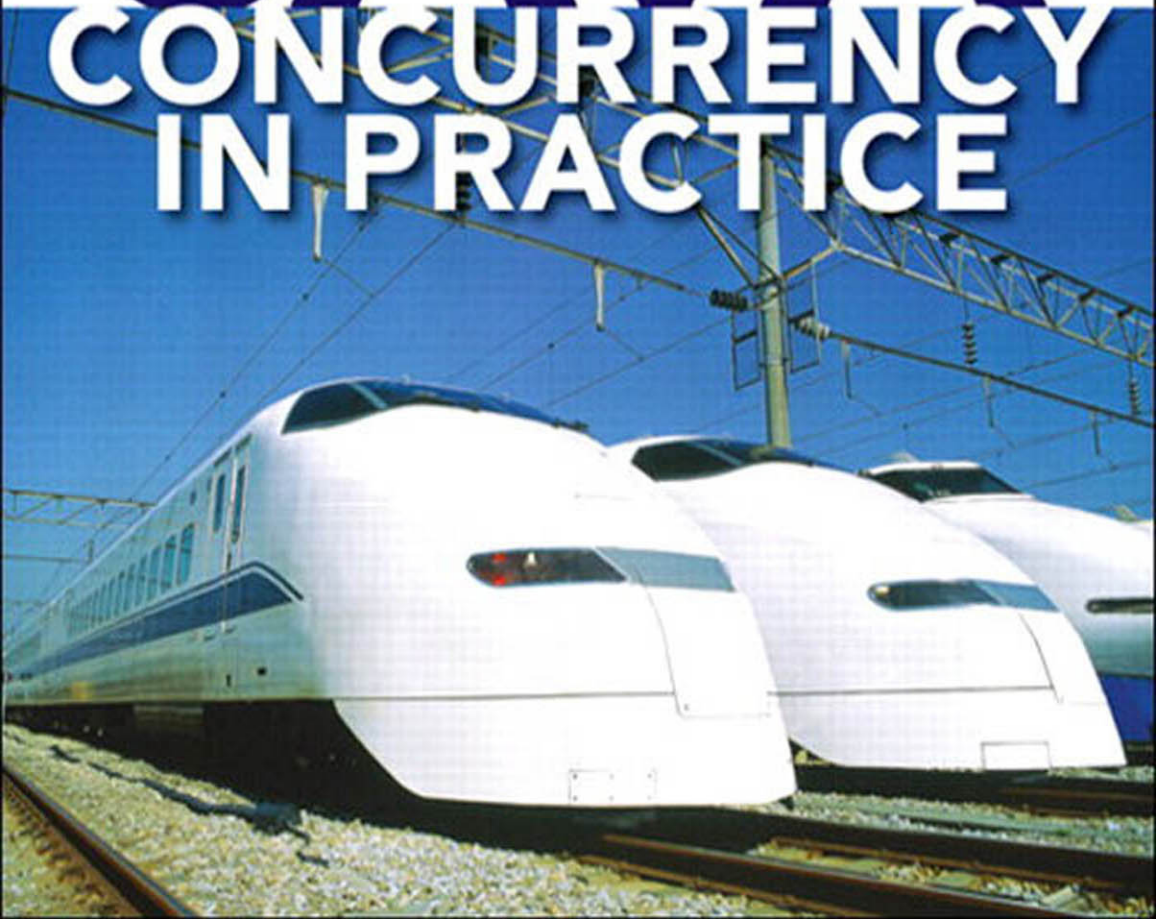
Brian Goetz
with Tim Peierls, Joshua Bloch,
Joseph Bowbeer, David Holmes,
and Doug Lea

Java Concurrency in Practice



Advance praise for *Java Concurrency in Practice*

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

—Martin Buchholz
JDK Concurrency Czar, Sun Microsystems

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

—Doron Rajwan
Research Scientist, Intel Corp

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

—Ted Neward
Author of Effective Enterprise Java

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

—Kirk Pepperdine
CTO, JavaPerformanceTuning.com

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

—Dr. Cliff Click
Senior Software Engineer, Azul Systems

```
public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentMap<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) { f = ft; ft.run(); }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f);
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
```

Listing 5.19. Final implementation of `Memoizer`.

```
@ThreadSafe
public class Factorizer implements Servlet {
    private final Computable<BigInteger, BigInteger[]> c =
        new Computable<BigInteger, BigInteger[]>() {
            public BigInteger[] compute(BigInteger arg) {
                return factor(arg);
            }
        };
    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req,
                        ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest(req);
            encodeIntoResponse(resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "factorization interrupted");
        }
    }
}
```

Listing 5.20. Factorizing servlet that caches results using `Memoizer`.

Summary of Part I

We've covered a lot of material so far! The following "concurrency cheat sheet" summarizes the main concepts and rules presented in Part I.

- *It's the mutable state, stupid.*[1]

  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.*

  You could write a thread-safe program with all data stored in global variables, but why would you want to? Encapsulating data within objects makes it easier to preserve their invariants; encapsulating synchronization within objects makes it easier to comply with their synchronization policy.

- *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 process—or explicitly document that your class is not thread-safe.*

- *Document your synchronization policy.*

1. During the 1992 U.S. presidential election, electoral strategist James Carville hung a sign in Bill Clinton's campaign headquarters reading "The economy, stupid", to keep the campaign on message.

Part II

Structuring Concurrent Applications

This page intentionally left blank

Chapter 6

Task Execution

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

6.1 Executing tasks in threads

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

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

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