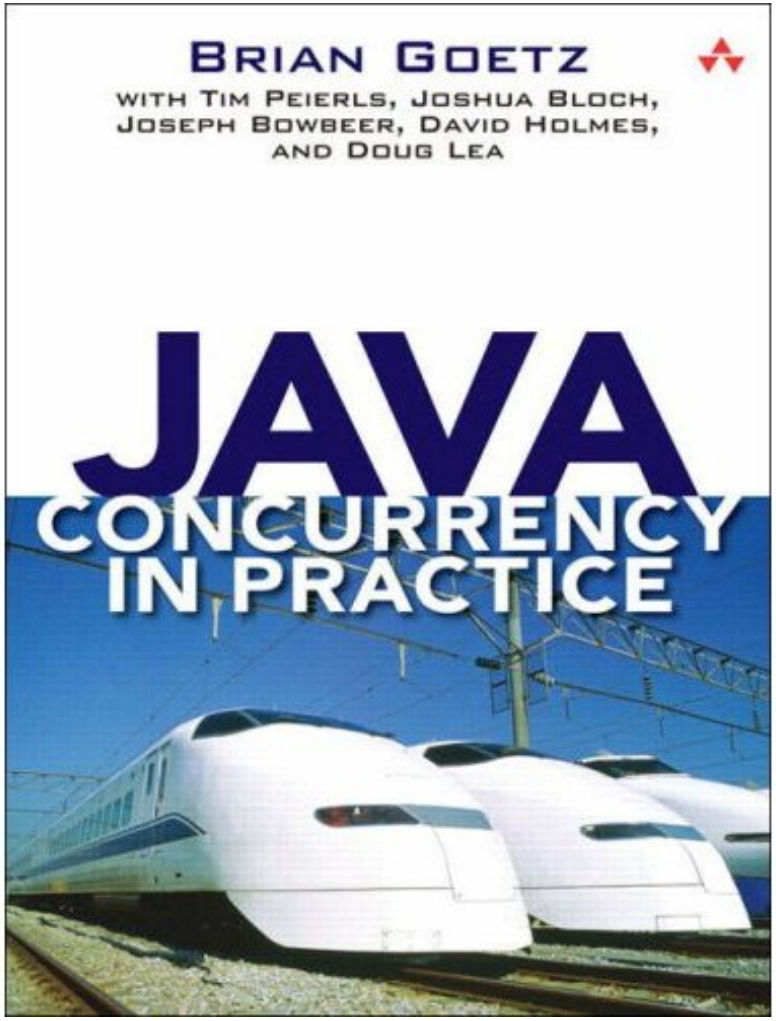
Concurrency

Reference


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE

Topics

- Problems
- Techniques
- Utilities

Problems

Deadlock

- Two threads - each require a resource held by the other
- Lock ordering,
  - Aggravated by obtaining multiple locks in inverted order
  - Dynamic ordering

Starvation & Livelock

- Thread is perpetually denied access to required resource, prohibiting it from making progress
- Livelock, thread is able to run but not make progress

Race Condition

- Completion order of tasks executed by separate threads determines correctness
- And, correct ordering is not assured

Techniques

Immutability

- Immutable objects have no variant state
- So… It is not possible for some number of threads to interact with the object in a manner that makes its state invalid or inconsistent

Confinement

- Eliminates the need for synchronization
- Simply limit access to a single thread
- Programmer responsibility, no language feature to enforce thread confinement

Reducing Synchronization

- Increasing concurrency improves performance, but blocking must be minimized
- Double check
  - Perform an initial check without synchronization, requires another check under synchronization to protect actual state change
- Lock splitting
  - Use discrete locks for independent critical sections

Correctness

- Correctness means a class honors its invariants and post-conditions
- Developing the proper locking strategy is critical in achieving correctness
- Lock hiding
  - Protects the designed locking strategy by prohibiting access to the locks being used

Collections

Queue

- **Collection interface designed to hold elements prior to processing**
- **Generally** `FIFO`**, but may support** `Comparator`/`Comparable` **ordering**
  - `boolean offer(E o)`
  - `E peek()`
  - `E element()`
    `throws NoSuchElementException`
  - `E poll()`
  - `E remove()`
    `throws NoSuchElementException`

BlockingQueue

- Queue **interface that supports blocking put and take operations**
- **Blocks for availability or capacity**
  - `void put(E o)`
  - `E take()`

SynchronousQueue

- `BlockingQueue` **implementation**
- **The put and take operations each block waiting for the other**
- **The queue has no capacity, serves as a rendezvous point for cooperating threads**
- **Supports an optional** `FIFO` **fairness policy**

ArrayBlockingQueue

- **A bounded** `BlockingQueue`**, backed by an array**
- **Capacity is fixed upon creation**
- `FIFO` **order**
- **Supports an optional** `FIFO` **fairness policy**

LinkedBlockingQueue

- Based on linked nodes
- Capacity may be fixed upon creation
- Link nodes are created upon insertion
- FIFO order
- Supports an optional FIFO fairness policy

PriorityBlockingQueue

- **Unbounded** PriorityQueue
- **Ordered by** Comparable/Comparator

DelayBlockingQueue

- **Unbounded** `PriorityQueue`
- **Elements have an associated delay which determines when they may be taken**
- **Ordered delay expiration**
  - `boolean offer(E o, long timeout, TimeUnit unit)`
  - `E poll(long timeout, TimeUnit unit)`

ConcurrentHashMap

- Supports full concurrency of access and update operations
- Retrieval operations do no block
- Level of concurrency for update operations is configurable upon construction

Collections Utility

- **Factory for providing synchronized wrappers for collections**
- java.util.Collections
  - **Synchronized wrappers for collections** (Collection, List, Map, ...)

```
static <T> List<T> synchronizedList(List<T> list)
static <K,V> Map<K,V> synchronizedList(Map<K,V> m)
```

  - **Other variations**

Synchronization Aids

Atomics

- `java.util.concurrent.atomic`
- **Classes that support lock-free thread-safe programming on single variables**

```
boolean compareAndSet(expectedValue,
                      updateValue);
```

- **Pre and post increment and decrement methods**

CountDownLatch

- **Enable a thread to wait for some number of tasks to be completed**
- **The latch is initialized with a count, calls to `await` blocks until count reaches zero**

```
void await()
boolean await(long timeout,
              TimeUnit unit)
void countDown()
```

CyclicBarrier

- **Provides a common synchronization point (barrier) for multiple threads**
- **Barrier is initialized with a count, n, all calls to `await` block until the n^{th} call**
- **May be reused once the barrier is broken**

```
int await()
boolean await(long timeout,
              TimeUnit unit)
boolean isBroken()
```

Locks

- `java.util.concurrent.locks`
- **Providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors**
- **Principle interfaces**; `Lock`, `Condition` `ReadWriteLock`

Lock

- **Interface defines operations for flexible structuring of locks and multiple conditions**

```
void lock()
boolean tryLock()
boolean tryLock(long, TimeUnit)
void unlock()
```

  - **Always call from within** `finally` **block**

```
Condition newCondition()
```

Condition

- **Factors our the wait/notify mechanism of the** `Object` **monitor**
- **Allows multiple wait sets per object**

```
void await()
boolean await(long, TimeUnit)
boolean awaitUntil(Date)
void signal()
void signalAll()
```

Lock/Condition Example

```
private Object lock = new Object();

private Lock lock;

/** The condition. */
private Condition condition;

/** The value to be incremented. */
private int value = 0;

public LockConditionDemo() {
    lock = new ReentrantLock();
    condition = lock.newCondition();
}
```

Lock/Condition Example

```
void method1() {
    lock.lock();
    try {
        ++value;
        condition.await();
    } catch (InterruptedException ex) {
        // means something called interrupt()
    } finally {
        --value;
        lock.unlock();
    }
}
```

Lock/Condition Example

```
void method2() {
    lock.lock();
    try {
        condition.signal();
        --value;
    } finally {
        lock.unlock();
    }
}
```

ReentrantLock

- Reentrant, mutual exclusive, lock
- Optionally supports FIFO fairness
- Methods for:
  - Obtaining the current owning thread
  - Determine if there are any queued threads
  - Getting queued threads

ReadWriteLock

- Maintains a pair of locks, one for read only operations one for writes
  - Read lock may be help by multiple readers while there are no writers
  - Write lock is exclusive
- Allows multiple wait sets per object

```
Lock readLock()
Lock writeLock()
```

- Implemented by `ReentrantReadWriteLock`

Semaphore

- A counting semaphore
- Manages a set of permits, initialized at creation
  - A binary semaphore is referred to a mutex

```
void acquire()
void release()
```

Executor

- **Replacement for direct use of** `Thread`
- `Executor` **should be used in place of** `Thread` **to execute** `Runnable` **objects**
  - `void execute(Runnable)`
- **Multiple** `Executor` **implementations**
  - `ThreadPoolExecutor`
  - `ScheduledThreadPoolExecutor`
- `Executors` **class is a factory for** `Executor` `objects`

Callable<V>

- **Similar to Runnable**
  - **May return a value**
  - **May throw checked exceptions**
- **Executed by** `Executor`

```
V call()throws Exception
```