JAVA

FUNDAMENTALS

ZEROTURNAROUND

THREAD SAFETY AND LOCKS

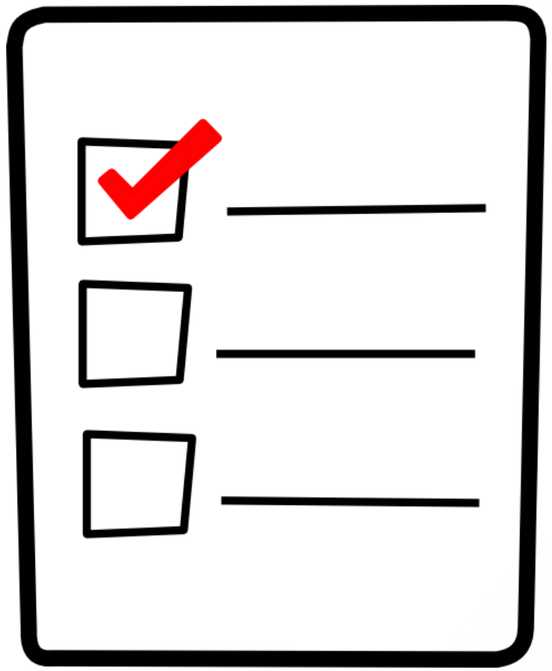
Mihhail Lapushkin

mihhail.lapushkin@zeroturnaround.com

March 13, 2017

AGENDA

- Executors
- Locks
- Concurrency idioms
- Sharing objects
- Homework



EXECUTORS

TASK

Tasks are independent activities

EXECUTING TASKS SEQUENTIALLY

```
class WebServer {
  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      handleRequest(connection);
    }
  }
}
```

EXPLICITLY CREATING THREADS

```
class WebServer {
  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      new Thread(() -> handleRequest(connection)).start();
    }
  }
}
```

DISADVANTAGES

- Thread lifecycle overhead
- Resource consumption
- Stability

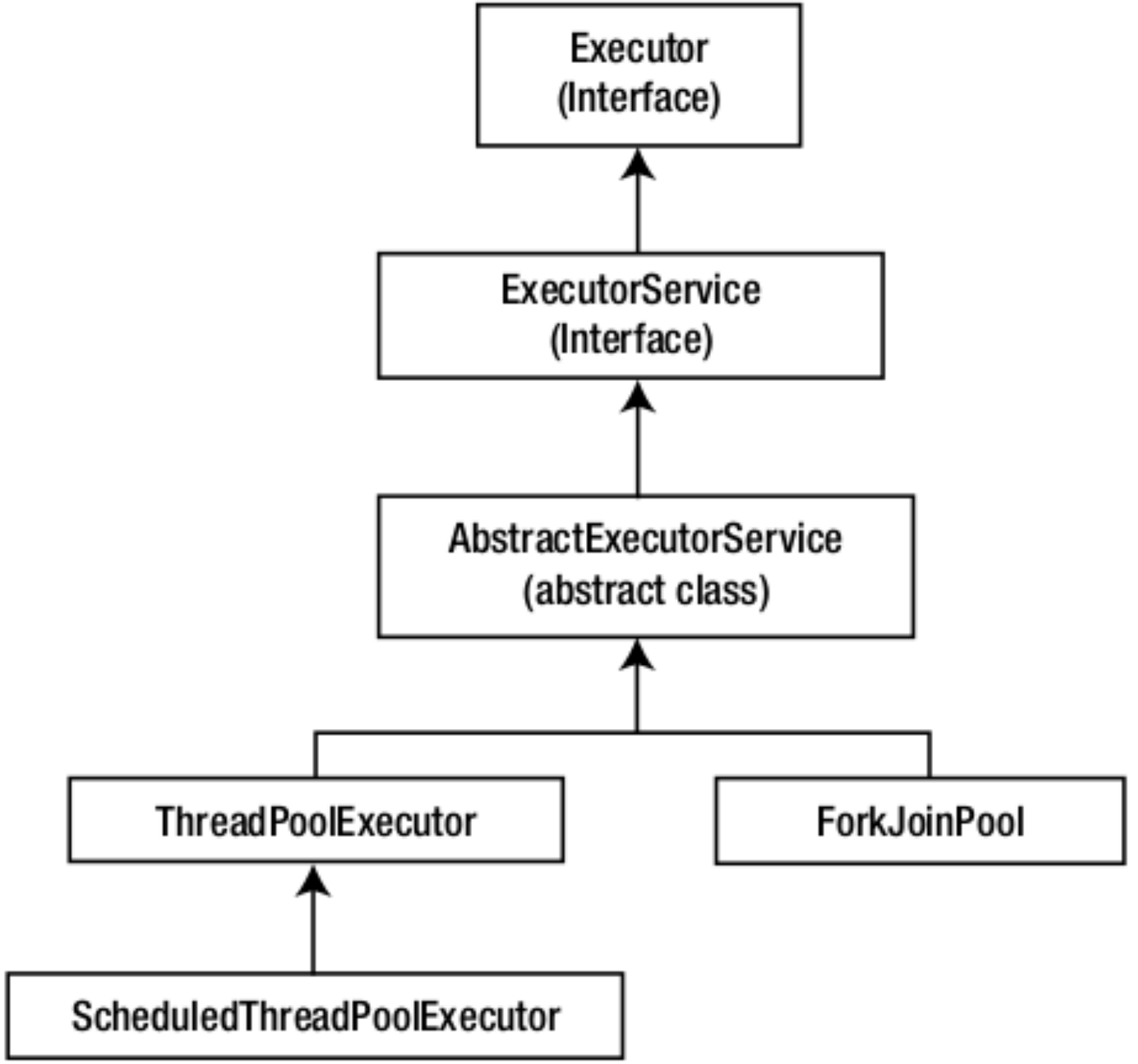
THREAD POOLS

Thread pool pattern consists of a number **m** of threads, created to perform a number **n** of tasks concurrently.

EXECUTOR

```
interface Executor {
  void execute(Runnable command);
}

interface Runnable {
  void run();
}
```


Executor
(Interface)
ExecutorService
(Interface)
AbstractExecutorService
(abstract class)
ThreadPoolExecutor
ForkJoinPool
ScheduledThreadPoolExecutor

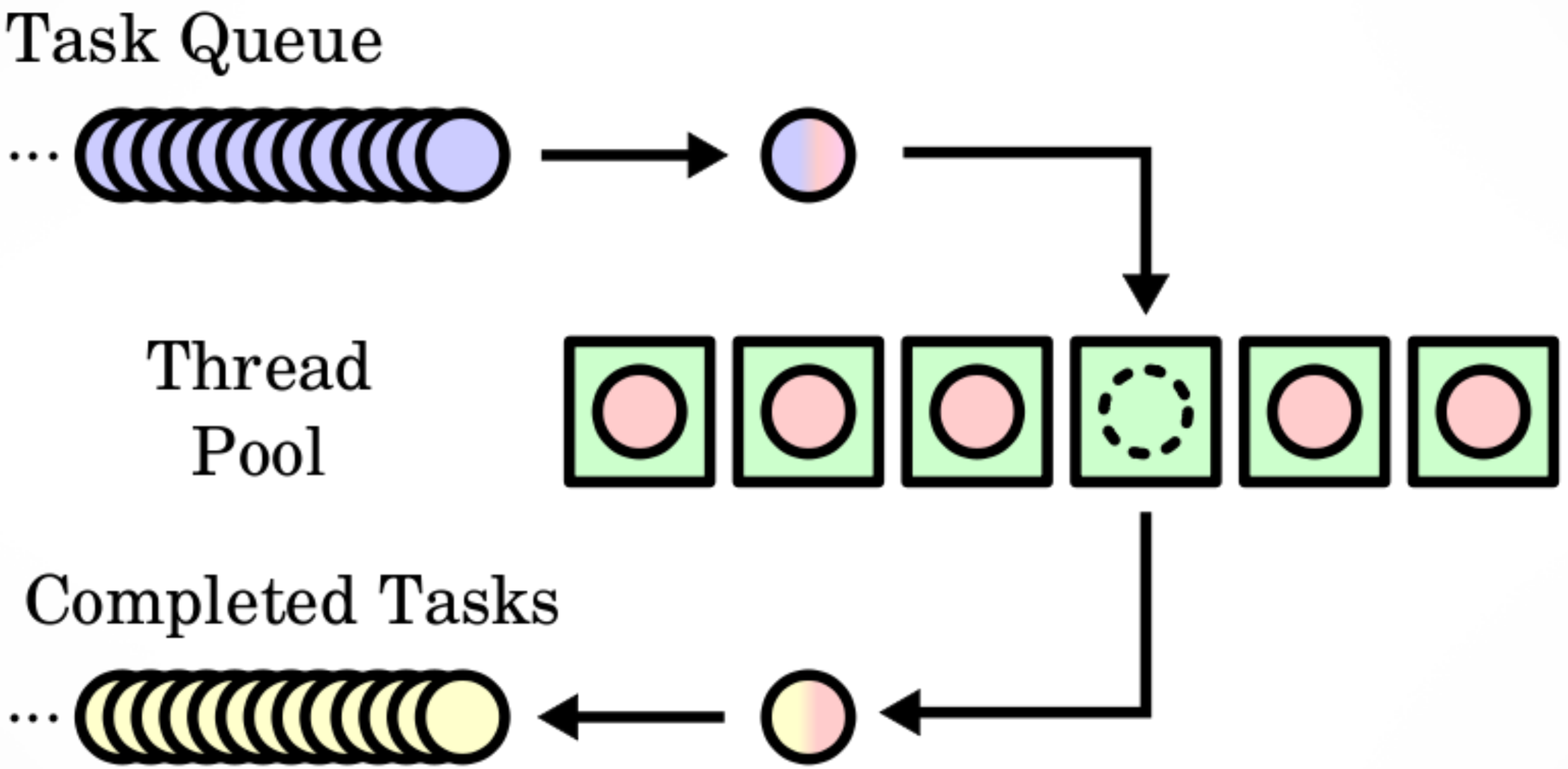
EXPLICITLY CREATING THREADS

```
class WebServer {
  public static void main(String[] args) {

    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      new Thread(() -> handleRequest(connection)).start();
    }
  }
}
```

USING A THREAD POOL

```
class WebServer {
  public static void main(String[] args) {
    Executor exec = Executors.newFixedThreadPool(100);
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      Socket connection = socket.accept();
      exec.execute(() -> handleRequest(connection));
    }
  }
}
```


Task Queue
Thread Pool
Completed Tasks

ADVANTAGES OF THREAD POOLS

- Minimal thread lifecycle overhead
- Enable to have enough threads to keep the processors busy
- Limit the number of threads to avoid `OutOfMemoryError`
- Improved responsiveness

EXECUTION POLICIES

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

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?

EXECUTION POLICIES

- 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?

THREAD POOL EXECUTORS

```
Executors.newFixedThreadPool(int nThreads)

Executors.newCachedThreadPool()

Executors.newSingleThreadExecutor()

Executors.newScheduledThreadPool()
```

ScheduledThreadPoolExecutor

- Enables to schedule tasks
  - after a given delay
  - at the specified time
- Enables to repeat tasks
  - at fixed rate
  - at fixed delay

EXECUTOR LIFECYCLE

```
interface ExecutorService extends Executor {
  void shutdown()
  List<Runnable> shutdownNow()
  boolean isShutdown()
  boolean isTerminated()
  boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException
  …
}
```

RESULT-BEARING TASKS

```
interface ExecutorService extends Executor {
  <T> Future<T> submit(Callable<T> task)
  <T> Future<T> submit(Runnable task, T result);
  Future<?> submit(Runnable task);
  ...
}

interface Callable<V> {
  V call()
}
```

RESULT-BEARING TASKS

```
interface Future<V> {
  boolean cancel(boolean mayInterrupt)
  boolean isCancelled()
  boolean isDone()
  V get()
    throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException,
           TimeoutException;
}
```

EXECUTING A BATCH OF TASKS

```
interface ExecutorService extends Executor {
  <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
  <T> T invokeAny(Collection<? extends Callable<T>> tasks)
  ...
}
```

LOCKS

LOCK OBJECT VS MONITOR

- Locks are similar to `synchronized` methods and statements, but more flexible
- Customizable (custom conditions, non-sequential lock acquisition/release, deadlock detection, etc.)

```
Lock l = ...;
l.lock();
try {
  // access the resource protected by this lock
} finally {
  l.unlock();
}
```

MEMORY VISIBILITY OF LOCKS

- All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in section 17.4 of The Java™ Language Specification:
  - A successful lock operation has the same memory synchronization effects as a successful Lock action.
  - A successful unlock operation has the same memory synchronization effects as a successful Unlock action.
- Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

LOCK INTERFACE

```
public interface Lock {
  void lock();                          // acquire lock, wait
  void lockInterruptibly()
    throws InterruptedException; // acquire lock, wait, interruptible
  boolean tryLock();                    // acquire lock, immediate
  boolean tryLock(long time, TimeUnit unit)
    throws InterruptedException; // acquire lock, immediate, …
                                        // … timed, interruptible
  void unlock();                        // release lock
  Condition newCondition();
}
```

KEEP A LOCK

```
class KeepLock extends Thread {
  private final Object lock;
  public KeepLock(Object lock) { this.lock = lock; }
  public void run() {
    try {
      synchronized (lock) {
        while (true) Thread.sleep(1000); // doing stuff
      }
    } catch (InterruptedException e) { }
  }
}
```

KEEP A LOCK

```
public class KeepLockRunner {
  public static void main(String[] args) {
    Object lock = new Object();
    Thread t = new KeepLock(lock);
    t.start();
    Thread.sleep(1000);
    t.interrupt();
  }
}
```

KEEP A LOCK

```
public class KeepLockRunner {
  public static void main(String[] args) {
    Object lock = new Object();
    Thread t1 = new KeepLock(lock);
    Thread t2 = new KeepLock(lock); // -added-
    t1.start();
    Thread.sleep(1000);
    t2.start();      // -added-
    t1.interrupt();
    Thread.sleep(1000); // -added-
    t2.interrupt(); // -added-
  }
}
```

KEEP A LOCK

```
class KeepLock extends Thread {
  private final Object lock;
  public KeepLock(Object lock) { this.lock = lock; }
  public void run() {
    try {
      synchronized (lock) {
        while (true) Thread.sleep(1000); // doing stuff
      }
    } catch (InterruptedException e) { }
  }
}
```

KEEP A LOCK

```
public class KeepLockRunner {
  public static void main(String[] args) {
    Object lock = new Object();
    Thread t1 = new KeepLock(lock);
    Thread t2 = new KeepLock(lock);
    t1.start();
    Thread.sleep(1000);
    t2.start();
    Thread.sleep(1000);
    t2.interrupt();
    t1.interrupt();
  }
}
```

OOPS!

Cannot interrupt the second thread while the first thread is not interrupted

ReentrantLock

- A re-entrant mutual exclusion Lock with the same basic behaviour and semantics as the implicit monitor lock accessed using `synchronized` methods and statements, but with extended capabilities.
  - Interruptible
  - Timed
  - Fairness
- Returns immediately if the lock is held by the current thread

KEEP A LOCK INTERRUPTIBLY

```
class KeepLock extends Thread {
  private final ReentrantLock lock;
  public KeepLock(ReentrantLock lock) { this.lock = lock; }
  public void run() {
    try {
      lock.lockInterruptibly();
      try {
        while (true) Thread.sleep(1000); // doing stuff
      } finally {
        lock.unlock();
      }
    } catch (InterruptedException e) {}
  }
}
```

KEEP A LOCK INTERRUPTIBLY

```
public class KeepLockRunner {
  public static void main(String[] args) {
    Object lock = new Object();
    Thread t1 = new KeepLock(lock);
    Thread t2 = new KeepLock(lock);
    t1.start();
    Thread.sleep(1000);
    t2.start();
    Thread.sleep(1000);
    t2.interrupt();
    t1.interrupt();
  }
}
```

KEEP A LOCK INTERRUPTIBLY

```
public class KeepLockRunner {
  public static void main(String[] args) {
    ReentrantLock lock = new ReentrantLock();
    Thread t1 = new KeepLock(lock);
    Thread t2 = new KeepLock(lock);
    t1.start();
    Thread.sleep(1000);
    t2.start();
    Thread.sleep(1000);
    t2.interrupt();
    t1.interrupt();
  }
}
```

KEEP A LOCK INTERRUPTIBLY

```
class KeepLock extends Thread {
  private final ReentrantLock lock;
  public KeepLock(ReentrantLock lock) { this.lock = lock; }
  public void run() {
    try {
      lock.lockInterruptibly();
      try {
        while (true) Thread.sleep(1000); // doing stuff
      } finally {
        lock.unlock();
      }
    } catch (InterruptedException e) {}
  }
}
```

ReadWriteLock

A lock that offers better concurrent access. Useful for synchronisation when reads are frequent and writes infrequent

- Read lock can be held by multiple threads as long as write lock is not held.
- Write lock is exclusive
- Must guarantee that the memory synchronization effects of writeLock operations also hold with respect to the associated readLock. A thread successfully acquiring the read lock will see all updates made upon previous release of the write lock.

```
interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

DEADLOCK

- When a thread holds a lock forever, other threads attempting to acquire that lock will block forever waiting.
- When thread **A** holds lock **L** and tries to acquire lock **M**, but at the same time thread **B** holds **M** and tries to acquire **L**, both threads will wait forever
- No way to resolve a deadlock on a JVM, when a set of threads deadlock, thats it. The only way to restore the application to health is to abort and restart it - and hope the same thing doesn't happen again.
- Depending on what those threads do, the application may stall completely, or a particular subsystem may stall, or performance may suffer.

DEADLOCK

```
class LeftRightDeadlock {
  private final Object left = new Object();
  private final Object right = new Object();
  public void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }
  public void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomethingElse();
      }
    }
  }
}
```

DEADLOCK

- The deadlock in `LeftRightDeadlock` came about because the two threads attempted to acquire the same locks in a different order.
- If they asked for the locks in the same order, there would be no cyclic locking dependency and therefore no deadlock.
- If you can guarantee that every thread that needs locks **L** and **M** at the same time always acquires **L** and **M** in the same order, there will be no deadlock

AVOIDING DEADLOCKS

- If possible, never acquire more than one lock
- When acquiring multiple locks, ensure that lock ordering is consistent across your entire program
- `Lock.tryLock(long time, TimeUnit unit)`

AVOIDING DEADLOCKS

```
void transferMoney(Account fromAccount, Account toAccount, Amount amount) {
  synchronized (fromAccount) {
    synchronized (toAccount) {
        fromAccount.debit(amount);
        toAccount.credit(amount);
    }
  }
}
```

AVOIDING DEADLOCKS

```
void transferMoney(Account fromAccount, Account toAccount, Amount amount) {
  Account account1 = …
  Account account2 = …

  synchronized (account1) {
    synchronized (account2) {
        fromAccount.debit(amount);
        toAccount.credit(amount);
    }
  }
}
```

THREAD-DUMP ANALYSIS

- While preventing deadlocks is mostly your problem, the JVM can help identify them when they do happen using thread dumps.
- A thread dump includes a stack trace for each running thread, similar to the stack trace that accompanies an exception.
- Thread dumps also include locking information, such as which locks are held by each thread, in which stack frame they were acquired, and which lock a blocked thread is waiting to acquire

GENERATING THREAD-DUMPS

- Send the JVM process a **SIGQUIT** signal (kill -3) on Unix platforms
- Press the **Ctrl-** key on Unix platforms
- Press **Ctrl-Break** on Windows platforms
- Many IDEs can request a thread dump also

CONCURRENCY IDIOMS

CONCURRENT PROGRAMS

- *“No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state.”*
- Manages access to **shared mutable state**
- **Visibility** across different threads of execution
- **Atomicity** of compound operations

SHARED MUTABLE STATE

```
class TellMeTheNumber {
  static boolean ready;
  static int number;
  static class ReaderThread extends Thread {
    public void run() {
      while (!ready) Thread.yield();
      System.out.println(number);
    }
  }
  public static void main(String[] args) {
    new ReaderThread().start();
    number = 42; ready = true;
  }
}
```

VISIBILITY

- Visibility in a single thread is natural and intuitive
- In multi-threaded applications, things that can go wrong are subtle and counterintuitive
- Visibility across threads must be ensured by using proper synchronisation
- Happens-before relationship

LOCKING AND VISIBILITY

- Threads entering `synchronized` blocks guarded by the same lock see the each other writes.
  - Without synchronization, there is no such guarantee.
  - You could see stale values
- Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops.

LOCKING AND VISIBILITY

```
public class MutableInteger {
  private int value;

  public int get() {
    return value;
  }

  public void set(int value) {
    this.value = value;
  }
}
```

LOCKING AND VISIBILITY

```
public class MutableInteger {
  private int value;

  public synchronized int get() {
    return value;
  }

  public synchronized void set(int value) {
    this.value = value;
  }
}
```

VOLATILE

- The visibility effects of `volatile` variables extend beyond the value of the `volatile` variable itself. When thread **A** writes to a `volatile` variable and subsequently thread **B** reads that same variable, the values of all variables that were visible to **A** prior to writing to the `volatile` variable become visible to **B** after reading the `volatile` variable.
- From a memory visibility perspective, writing a `volatile` variable is like exiting a `synchronized` block and reading a volatile variable is like entering a `synchronized` block
- Compound operations still require locks!

VOLATILE

```
public class MutableInteger {
  private int value;

  public int get() {
    return value;
  }

  public void set(int value) {
    this.value = value;
  }
}
```

VOLATILE

```
public class MutableInteger {
  private volatile int value;

  public int get() {
    return value;
  }

  public void set(int value) {
    this.value = value;
  }
}
```

ATOMICITY

- Compound operations that from a perspective of another thread should be atomic - either all operations are done or none of them
  - check-then-act (lazy initialization)
  - read-modify-write (increment operation)
- Use classes in the `java.util.concurrent.atomic` package or proper synchronization to ensure atomicity

CHECK-THEN-ACT

```
public class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }

    return instance;
  }
}
```

CHECK-THEN-ACT

```
public class Singleton {
  private static Singleton instance;

  public static synchronized Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }

    return instance;
  }
}
```

READ-MODIFY-WRITE

```
public class Counter {
  private int value = 0;

  public int get() {
    return value;
  }

  public void increment() {
    this.value++;
  }
}
```

READ-MODIFY-WRITE

```
public class Counter {
  private int value = 0;

  public synchronized int get() {
    return value;
  }

  public synchronized void increment() {
    this.value++;
  }
}
```

SHARING OBJECTS

PUBLISHING AN OBJECT

- Publishing an object means making it available to code outside of its current scope, such as by storing a reference to it where other code can find it, returning it from a non-private method, or passing it to a method in another class
- Object internals should generally not be published
- Publishing an object for general use should be done in a thread-safe manner
- Publishing objects before they are fully constructed can compromise thread-safety
- An object that is published when it should not have been is said to have escaped

PUBLISHING AN OBJECT

- Object internals should generally not be published

```
class Secrets {
  public Set<Secret> secrets;

  public Secrets() {
    secrets = new HashSet<>();
  }
}
```

PUBLISHING AN OBJECT

```
class Secrets {
  private Set<Secret> secrets;

  public Secrets() {
    secrets = new HashSet<>();
  }
}
```

PUBLISHING AN OBJECT

- Publishing an object for general use should be done in a thread-safe manner

```
class States {
  private String[] states = new String[]{"AK","AL"};

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

PUBLISHING AN OBJECT

```
class States {
  private String[] states = new String[]{"AK","AL"};

  public String[] getStatesSnapshot() {
    return Arrays.copyOf(states, states.length);
  }
}
```

PUBLISHING AN OBJECT

```
class States {
  private String[] states = new String[]{"AK","AL"};

  public synchronized String getState(int index) {
    return states[index];
  }

  public synchronized void setState(String state, int index) {
    states[index] = state;
  }
}
```

SAFE CONSTRUCTION

- An object is in a predictable, consistent state only after its constructor returns, so publishing an object from within its constructor can publish an incompletely constructed object
  - Do not let the `this` reference to escape
  - Creating an instance of an anonymous inner class includes a reference to `this`
  - Creating an instance of a `Runnable` and starting a `Thread` in a constructor

SAFE CONSTRUCTION

```
public class EventConsumer {
  public EventConsumer(EventSource source) {
    source.registerListener(new EventListener() {
      public void onEvent(Event e) {
        doSomething(e);
      }
    });
    …
  }
}
```

SAFE CONSTRUCTION

```
public class EventConsumer {
  private EventConsumer() {}
  public static EventConsumer create(EventSource source) {
    final EventConsumer consumer = new EventConsumer();
    source.registerListener(new EventListener() {
      public void onEvent(Event e) {
        consumer.doSomething(e);
      }
    });
    return consumer;
    …
  }
}
```

THREAD CONFINEMENT

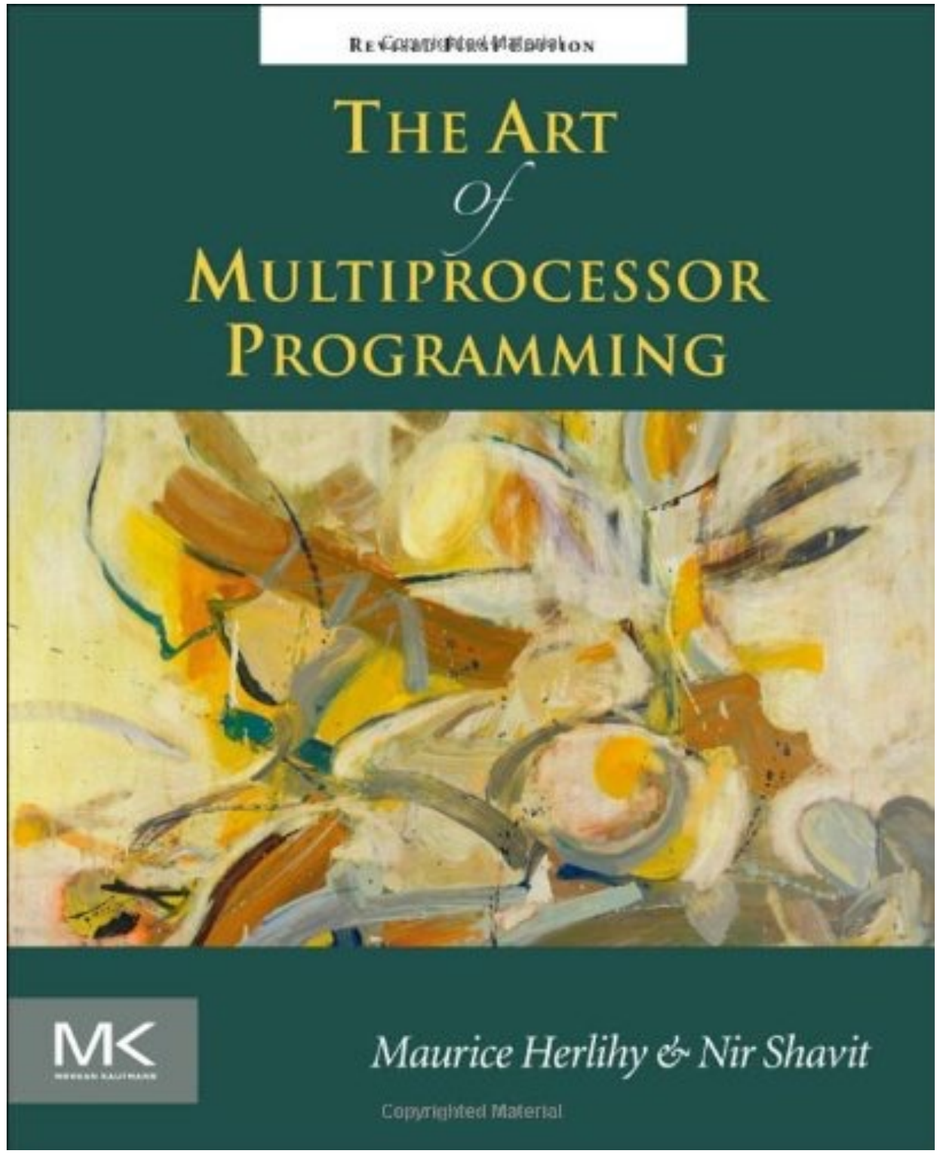
- One way to avoid accessing shared data is to not share
- If data is only accessed from a single thread, no synchronization is needed.
- Thread confinement is an element of your program's design that must be enforced by its implementation. The language has no mechanism for confining an object to a thread.

ThreadLocal

- `ThreadLocal` provides get and set accessor methods that maintain a separate copy of the value for each thread that uses it
- A get returns the most recent value passed to set from the currently executing thread

The Art of Multiprocessor Programming

by Maurice Herlihy, Nir Shavit



http://www.amazon.com/The-Multiprocessor-Programming-Revised-Reprint/dp/0123973376

HOMEWORK 7

https://github.com/JavaFundamentalsZT/jf-hw-money-transfers