week 10 - reasoning about concurrency properties

Overview
In the past weeks, we have learned how to write multi-threaded applications in Java, and studied primitives (locks, the monitor pattern, conditional synchronization, etc.) to make such applications correct. What does correctness really mean for a multi-threaded application though? This week, we better define correctness, and describe correctness properties as well as their violation. We finally discuss how to reason about correctness of multi-threaded applications and larger systems.

References
Essential readings: GPBBHL 10

Chapter 10 - Avoiding Liveness Hazards

Liveness hazards are a serious problem in concurrent software, as there is no way to recover from them short of aborting the application. We use locking to ensure thread safety, but indiscriminate use of locking can cause **lock-ordering deadlocks**. There are also **resource deadlocks** which can stem from wrongly/misunderstanding the bounding resources consumption via thread pools and semaphores

Java applications don't recover from deadlock so you need to be aware of the potential liveness hazards that can affect your multi-threaded programs, when designing Java applications with concurrency.

10.1 Deadlock

Also known as a **deadly embrace**

Occurs when a thread holds a lock forever, and other threads attempting to acquire that same lock will block forever waiting, thus no progress can be made

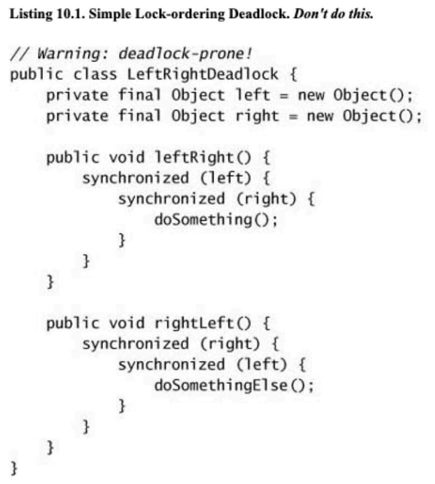
SImplest case of deadlock: when a thread A holds lock L and wants to acquire lock M but at the same time thread B holds lock M and wants to acquire lock L, both threads will have to wait forever due to a **cyclic locking dependency** (e.g. cyclic relationship in a directed graph can mean that there is a deadlock)

It is possible for two trasnactions to deadlock, where without intervention, it can wait forever. database systems can detect deadlocks and intervene, aborting the transacrtion

The only way to restore a Java application with deadlocks is to abort and restart it

Lock-ordering deadlocks
You can spot deadlocks by observing **nested lock acquisitions**

Example of a lock-ordering deadlock:



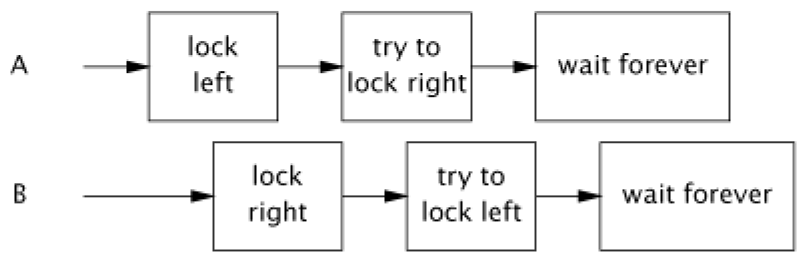
Listing 10.1. Simple Lock-ordering Deadlock. *Don't do this.*

```
// Warning: deadlock-prone!
public 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();
            }
        }
    }
}
```

This class has two locks: left and right locks where both methods acquire both locks but in different order. If one thread calls `leftRight` and another one calls `rightLeft` at the same time, and their actions are **interleaved**, the two threads will **deadlock**.

What happens if there is unlucky timing between two threads - they can be at risk of deadlock:

Figure 10.1. Unlucky Timing in `LeftRightDeadlock`.



The two threads attempted to acquire the same locks but in a different order, resulting in a cyclic locking dependency. If they ask for the locks in same order, there will be no dependency thus no deadlock.

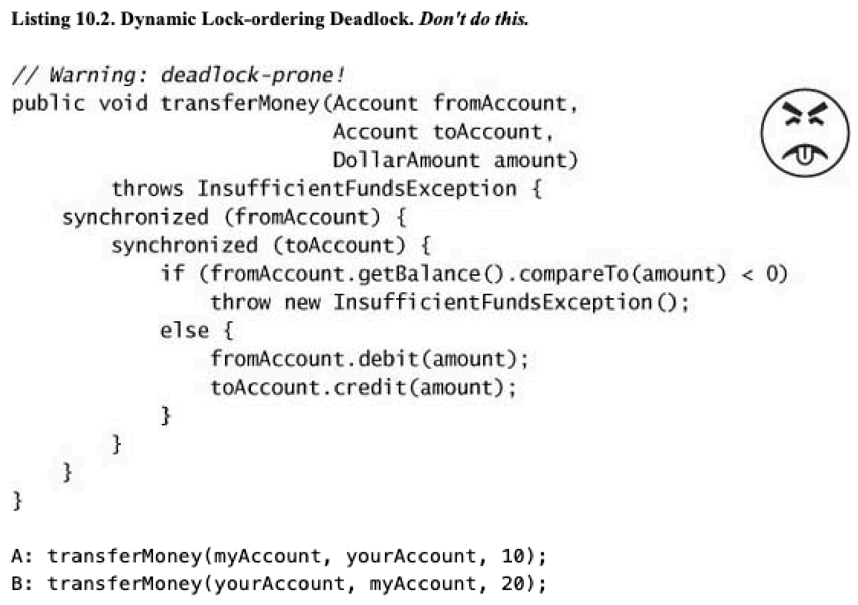
You need to guarantee that every thread that needs to lock two locks at the same time always acquire the same locks in the same order, to ensure NO deadlock

A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order

Left hand needs to know what the right hand is doing when it comes to locking - it is not sufficient to just inspect code paths that acquire locks individually.

Dynamic lock order deadlocks
Deadlocks can happen even if two threads call the same method at the same time, where in the method, locks are acquired in the same order

for example:



Listing 10.2. Dynamic Lock-ordering Deadlock. ***Don't do this.***

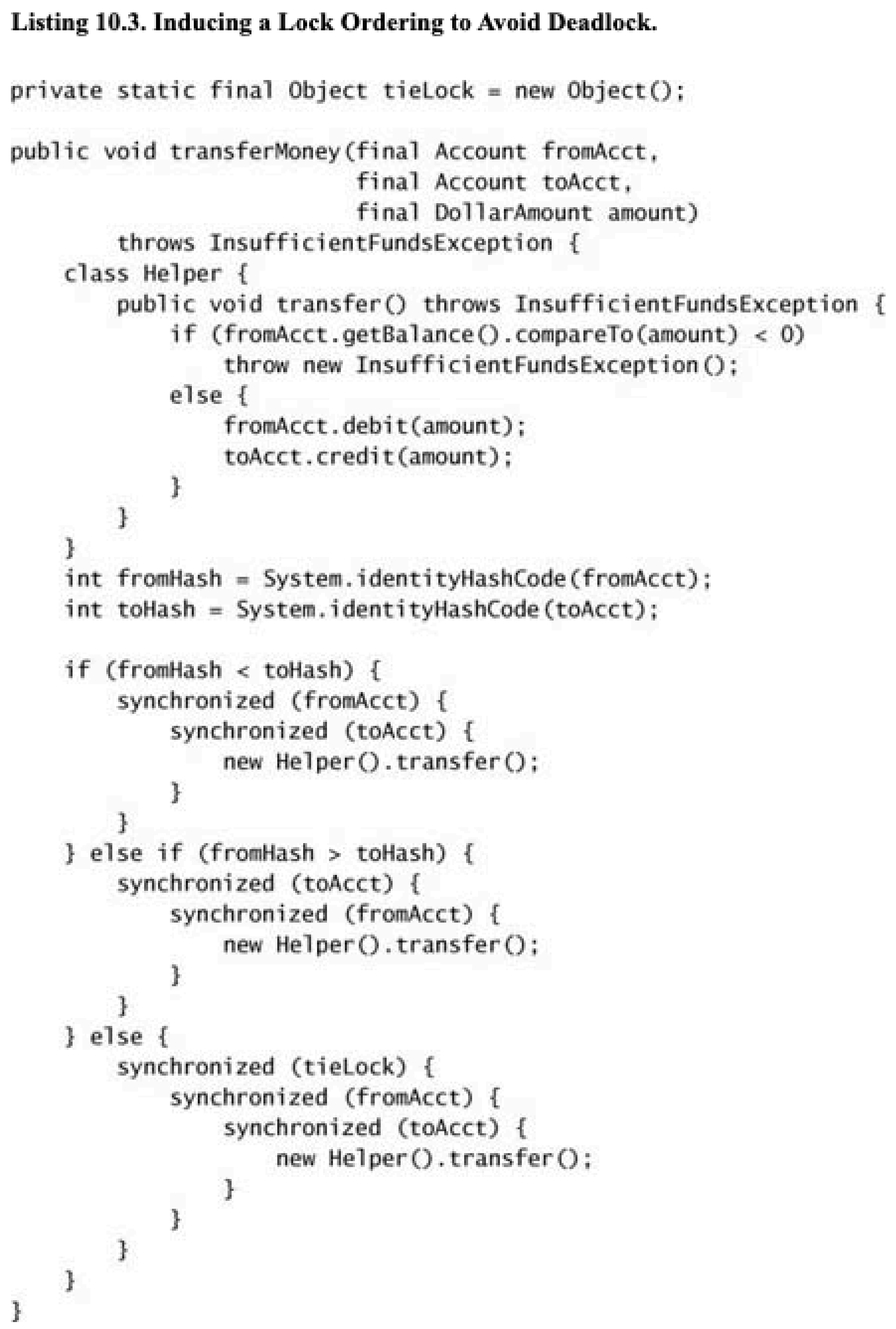
```
// Warning: deadlock-prone!
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amount)
        throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}

A: transferMoney(myAccount, yourAccount, 10);
B: transferMoney(yourAccount, myAccount, 20);
```

All of the threads acquire their locks in the same order, but lock order depends on order of arguments passed to the method `transferMoney`, and these in turn might depend on external inputs. Deadlock can occur if two threads call that method at the same time, one transferring from X to Y and other doing opposite

Unlucky timing happens when A acquire locks on `myAccount` and waits for lock on `yourAccount` while B is holding the lock on `yourAccount` and is waiting for lock on `myAccount`. This can cause a cyclic locking dependency. You can look out for nested lock acquisitions to be able to spot potential deadlocks

SInce order of arguments are out of programmers' control, we must **induce an ordering on the locks** and acquire them according to the induced ordering consistency throughout the application.

You can use hash codes (might not be unique) to induce a lock ordering to eliminate possibility of deadlock, but a deadlock could happen if two objects have the same hash code. we can use a tie-breaking lock to prevent inconsitent lock ordering, so only 1 thread at a time perofrms acquring two locks in an arbitrary order, to eliminate possibility of deadlock.
e.g.


Listing 10.3. Inducing a Lock Ordering to Avoid Deadlock.

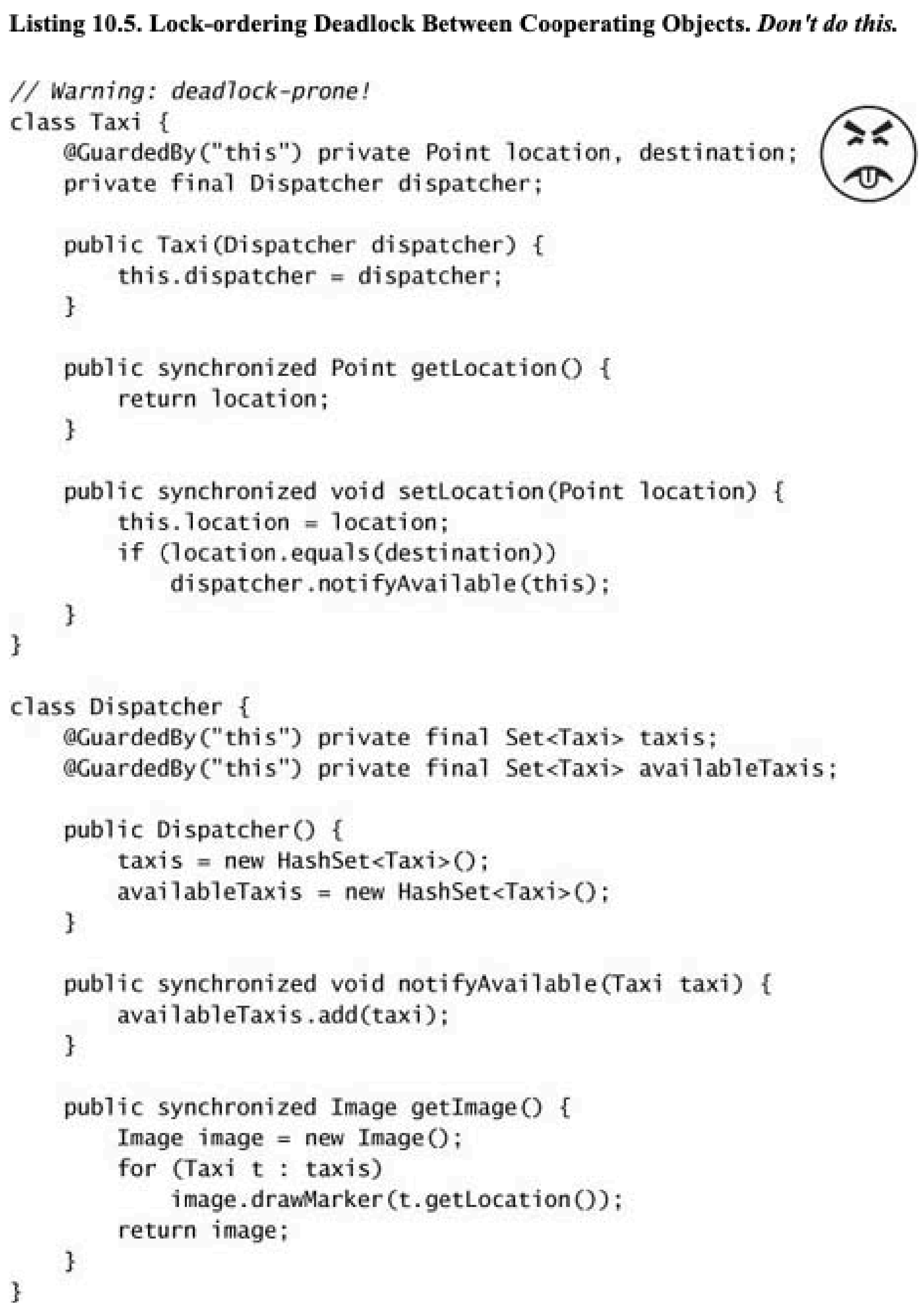
```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct,
                          final DollarAmount amount)
        throws InsufficientFundsException {
    class Helper {
        public void transfer() throws InsufficientFundsException {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
        }
    }
}
```

It is easier to use a unique, immutable key e.g. an id / accoutn number to induce a lock ordering (as you can order objects by their key - eliminating the need for a tie-breaking lock)

Lock ordering must be timed right otherwise there is a chance for applications in real-wrodl to deadlock, as they perform billions of lock acquire-release cycles per day

Deadlocks between cooperating objects
Multiple lock acquisition can occur even when two locks are not acquired by the same method, but among cooperating classes/objects as well. Deadlocks can happen if two locks are acquried by two threads but in differnet orders even among cooperating classes acquiring these locks (not same method by two threads)

Example:



Listing 10.5. Lock-ordering Deadlock Between Cooperating Objects. ***Don't do this.***

```java
// Warning: deadlock-prone!
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}

class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Invoking an alien method with a lock held is asking for liveness trouble as alien method may acquire other locks (risking deadlock) or block for an unexpectedly long time, stalling other threads that need the lock you hold.

Open calls
Calling an alien method with a lock held is difficult to analyse and therefore is risky

Open call: when you call a method with no locks held

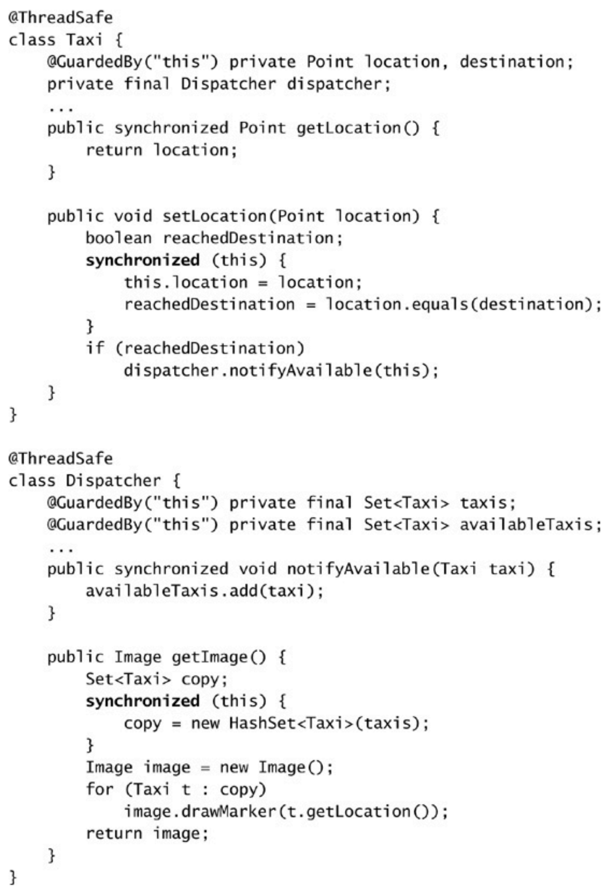
It is easier to perform liveness analysis on programs that use only open calls, compared to those that don't, as open calls make it easier to identify code paths that acquire multiple locks and therefore ensure locks are acquired in consistent orders.

Strive to use open calls throughout your program. Programs that rely on open calls are far easier to analyse for deadlock-freedom than those that allow calls to alien methods with locks held

Sometimes restructuring a `synchronized` block can cause loss of atomicity thus other methods have to be used to achieve atomicity, where you structure a concurrent object so that only 1 thread can execute the code path following the open call. This meant that instead of using locking to keep other threads out of critical sections of code, you construct protocols so that other threads don't try to get in

example:

Listing 10.6. Using open calls to avoiding deadlock between cooperating objects.



```
@ThreadSafe
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;
    ...
    public synchronized Point getLocation() {
        return location;
    }

    public void setLocation(Point location) {
        boolean reachedDestination;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
        if (reachedDestination)
            dispatcher.notifyAvailable(this);
    }
}

@ThreadSafe
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;
    ...
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Resource deadlocks
Threads can deadlock when waiting for resources

Assuming there are two pooled resources e.g. connection pools for 2 different databases, if a task requires connections to both databases and two resources are not always requested int he same order, thread A could hold a connection to database D_1 while waiting for connection to database D_2, and thread B could hold a connection to database D_2 while waitinfg for connection to D_1.

(Larger the pools are, the less likely this is to occur as if each pool has N connections, deadlock requires N sets of cyclically waiting threads and a lot of unlucky timing)

This can result in cyclic waiting threads due to unlucky timing

Thread-starvation deadlock

- another form of resource-based deadlock
- primary source is having tasks thjat wait for results of other tasks

10.2 Avoiding and diagnosing deadlocks

you won't have lock-ordering deadlocks when you have a program that never acquires more than one lock at a time.

Lock orderin gis essential when your program must acquire multiple locks but try to miniomise number of potential locking interactions and to follow/document a lock-ordering protocol for locks that may be acquired together

for programs that uses fine-grained locking, you can audit your code where it:

- identifies where multiple locks could be acquired
- then perform a global analysis of all such instances to ensure that lock ordering is consistent across the entire program
  try to use open calls wherever possible as it simplifies the analysis

Timed lock attempts
TImed `tryLock` feature of `Lock` classes instad of intrinsic locking to detect/recover from deadlocks

They allow to specify a timeout after which `tryLock` returns failure. YOu can regain control when something unexpected happens if you use a timeout that is much longer than the expected time it takes to acquire locks

It is a good method of logging any useful information about what you were trying to do and restart the computation, which is better than just killing the entire process whenever a timed lock attempt failed

Timed lock acquisitioon to acquire multiple locks can be effective against deadlocks even when timed locking is not used consistently throughout the prorgram, where if a lock acquisiton itmes out, you can release the locks and wait for a while and try again, possibly clearing the deadlock condition and allowing the program to recover. (Technique only works when two locks are acquired together; not if multiple locks are acquired due to nesting of method calls)

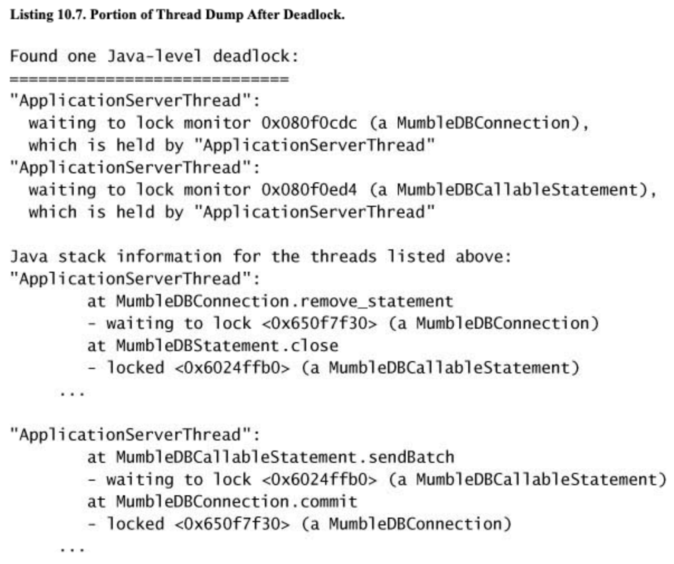
Deadlock analysis with thread dumps
JVm can identify deadlocks via thread dumps (stack trace for each running thread)

They include locking information e.g. which locks are held by each threaad, in which stack frame they were acquired, and which lock a bolocked thread is waiting to acquire.

JVM also checks for cycles in the `is-waiting-for` graph as they are a common source of deadlocks, where it will identify the involved locks/threads and where in the program the offending lock acquisitons are

JVM handles thread dumps for intrinsic locks better than explicit `Locks`

example of a thread dump in JVM (there is a cyclic locking dependency as shown in the analysis):



Listing 10.7. Portion of Thread Dump After Deadlock.

```
Found one Java-level deadlock:
=============================
"ApplicationServerThread":
  waiting to lock monitor 0x080f0cdc (a MumbleDBConnection),
  which is held by "ApplicationServerThread"
"ApplicationServerThread":
  waiting to lock monitor 0x080f0ed4 (a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread"

Java stack information for the threads listed above:
"ApplicationServerThread":
        at MumbleDBConnection.remove_statement
        - waiting to lock <0x650f7f30> (a MumbleDBConnection)
        at MumbleDBStatement.close
        - locked <0x6024ffb0> (a MumbleDBCallableStatement)
     ...

"ApplicationServerThread":
        at MumbleDBCallableStatement.sendBatch
        - waiting to lock <0x6024ffb0> (a MumbleDBCallableStatement)
        at MumbleDBConnection.commit
        - locked <0x650f7f30> (a MumbleDBConnection)
     ...
```

10.3 Other liveness hazards

Other liveness hazards encountered in concurrent programs:

- starvation
- missed signals (coverted in last week in section 14.2.3)
- livelock

Starvation
Occurs when a thread is perpetually denied access to resources it needs to make progress e.g. CPU cycles
Can be caused by inappropriate use of thread priorities, executing nonterminating constructs (infinite loops or resource waits that do not terminate) with a lock held, since other threads that need the lock will never be able to acquire it.

Avoid the temptation to use **thread priorities** as they incrase platform dependence and can cause liveness problems. Most concurrent applications can use defualt priority for all threads

Tweaking thread priorities can introduce risk of starvation as behaviour of application can become platform-specific.

Poor responsiveness
Not uncommon in GUI applications using background threads

CPU intensive background tasks can affect responsiveness as they compete for CPU cycles with event thread - altering thread priorities can make sense here where threads running purely background tasks can have lower priority to make foreground tasks more responsive

Poor responsiveness can be caused by poor lock management where if a thread holds a lock for along time,other threads that need to access that collection may have to wait for a very long time.

Livelock
A form of liveness failure in which a thread, while not block, still cannot make progress because it keeps retrying an opeartion that will always fails.

Often occurs in transactional messaging applications where the messaging infrastructure rolls back a transaction if a message cannot be processed successfully and puts it back at the head of the queue

Livelock can also occur when multiple cooperating threads change their state in response to others in such a way that no thread can ever make progress. Solution of that type of livelock is to introduce randomness into retry mechanism, where retrying with random waits and backoffs can be equally effective for avoiding livelokc in concurrent applications

Summary
There is no way to recover from liveness failures apart from aborting the application. They are also serious problems. Most common form of liveness failure is **lock-ordering deadlock**. You design protocols to avoid lock ordering deadlock where you ensure a consistent order when threads acquire multiple locks. Best way to do that is via **open calls** throughout your program as it reduces number of places where multiple locks are held at once, and makes it more obvious where those places are.