Questions

Can you write programs that allow multiple threads to update the same things simultaneously in order to act like a live update?

- yes
- mutual exclusion isn't to prevent updates of shared objects, just to prevent inconsistent results because of accessing or updating state while an update is in progress

Identifying Visibility Problems

Look for –

- instance variables accessed by multiple threads
  - private instance variables means that you only have to look for accesses within that class...and who calls the methods containing those accesses

when designing, also consider the potential for such situations

Identifying Race Conditions

Look for –

when designing, also consider the potential for such situations

- shared variables
  - private instance variables means that you only have to look for accesses within that class...and who calls the methods containing those accesses
- shared objects
  - can have multiple threads accessing a single reference
    - with GUI components, be aware of the JavaFX Application thread
  - can also have multiple threads accessing via multiple references
    - be aware of instance variables, return values, parameters passed
- multi-step operations
  - can be contained in a single line of code e.g. `x++;`
- situations where a state that is established at one point is expected to still hold later
  - can another thread change that state in the meantime?

Shared Variables

`shutdown_` is accessed by the thread running `run()`

`shutdown_` is set by the thread requesting the shutdown

(but `shutdown_` is a primitive type and all accesses are atomic, so visibility is the only problem and `volatile` is sufficient – do not need `synchronized`)

```
public class SlowCountdown2 implements Runnable {

    private int value_;
    private String indent_;

    private volatile boolean shutdown_;

    public SlowCountdown2 ( int value, String indent ) {
        value_ = value;
        indent_ = indent;
        shutdown_ = false;
    }

     * (non-Javadoc)
    @Override
    public void run () {
        for ( ; value_ >= 0 && !shutdown_ ; value_-- ) {
            System.out.println(indent_ + value_);
            try {
                Thread.sleep(1000);
            } catch ( InterruptedException e ) {
                System.out.println(indent_ + "interrupted!");
            }
        }
        System.out.println(indent_ + "done!");
    }

    public void shutdown () {
        shutdown_ = true;
    }

}
```

```
Thread thread1 = new Thread(countdown1);
Thread thread2 = new Thread(countdown2);

thread1.start();
thread2.start();
```

```
System.out.println("shutting down " + thread2.getName());
countdown2.shutdown();
thread2.interrupt();
```

Shared Objects

`account_` is not shared (`private`, only one thread calls `run()` on a given `SimpleATM` object)

```
public class SimpleATM implements Runnable {

    private BankAccount account_;

    public SimpleATM ( BankAccount account ) {
        super();
        account_ = account;
    }

     * (non-Javadoc)
    @Override
    public void run () {
        for ( int i = 0 ; i < 50 ; i++ ) {
            int balance = account_.getBalance();
            Thread.yield(); // give another thread a chance
            account_.setBalance(balance - 10);
        }
        System.out.println("total withdrawn: $500");
    }

}
```

but the same object is passed to `SimpleATM`, so different `account_` variables can refer to the same object and thus different threads access the same `BankAccount`

```
BankAccount account = new BankAccount(5000);

System.out.println("starting balance: " + account.getBalance());

Thread thread1 = new Thread(new SimpleATM(account));
thread1.start();

Thread thread2 = new Thread(new SimpleATM(account));
thread2.start();
```


Thread Safety, Not Overprotection

- *thread safe* refers to providing any necessary mutual exclusion when multiple threads access something
  - to say something isn't thread safe means that there could be problems if multiple threads access it
- but don't overprotect – mutual exclusion is expensive
  - the overhead of managing locks and reduced parallelism while threads wait
  - protected only data that is shared and changed
    - simultaneous reads are fine
  - prefer cheaper tools when they are enough
    - `volatile` for a single flag
    - keep data unshared if possible

Thread Safety

- the JavaFX scene graph is *not* thread safe
  - should be accessed only from the JavaFX Application thread (the event-handling thread)
- most Swing object methods are not thread safe
- most Java Collections classes are *not* thread safe
  - but a synchronized version is provided
- Java's streams/`Readers`/`Writers` are *not* thread safe
  - avoid sharing between threads or utilize mutual exclusion
  - avoid wrapping the same stream object with separate nozzles



```
BufferedReader reader1 =
  new BufferedReader(new InputStreamReader
                         (connection.getInputStream()));
BufferedReader reader2 =
  new BufferedReader(new InputStreamReader
                         (connection.getInputStream()));
```


Question

So, `synchronized` isn't the default because it's too computationally expensive and not always needed…?

- for collections classes, yes

"expensive" includes both the overhead of managing locks and the reduction in parallelism

Java provides synchronized versions of collections classes that wrap an existing object and provide thread safety.

static <T> Collection<T>	**synchronizedCollection**(Collection<T> c) Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T>	**synchronizedList**(List<T> list) Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V>	**synchronizedMap**(Map<K,V> m) Returns a synchronized (thread-safe) map backed by the specified map.
static <T> Set<T>	**synchronizedSet**(Set<T> s) Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V>	**synchronizedSortedMap**(SortedMap<K,V> m) Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static <T> SortedSet<T>	**synchronizedSortedSet**(SortedSet<T> s) Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.

- for objects (synchronized methods), yes
- for all applications where synchronized blocks are needed, the only possible default would be to not allow multiple threads

Deadlock

- *deadlock* occurs when two or more threads each wait for something another of them holds
  - no wrong result, just no result – program hangs

```
public void run () {
  synchronized ( from_ ) {
    pause(100);                    // something slow, e.g. checking for fraud
    synchronized ( to_ ) {
      if ( from_.withdraw(amount_) ) {
        to_.deposit(amount_);
      }
    }
  }
}
```

if two transfers A1→A2 and A2→A1 are initiated at the same time, each grabs the from account lock before trying to take the to account lock – transfer 1 holds A1 and wants A2, transfer 2 holds A2 and wants A1 … deadlock because neither releases the lock they have until after acquiring the lock the other holds

Avoiding Deadlock

- deadlock requires a cycle of threads each waiting for a lock another holds
- fix: break the cycle – always take locks in the same order
  - e.g. take the lock for the lower account number first, regardless of the direction of transfer

```
SafeBankAccount first = from_;
SafeBankAccount second = to_;
if ( to_.getID() < from_.getID() ) {
  first = to_;
  second = from_;
}
synchronized ( first ) {
  synchronized ( second ) { ... }
}
```

- as with races, deadlocks depend on timing and can't reliably be found only by testing – reason about lock order