1. Explain when you would use the following Java statement. Be sure to state explicitly any assumptions.

```
new Thread(this).start();
```

✓ Start a new thread running in parallel with the current thread. Assumes that this class implements Runnable (i.e., supports a public void run() methods). NB: this could also be an instance of a subclass of String.

What is the difference between that statement and the one below?

```
new Thread(this).run();
```

✓ The second statement just executes the run method without starting a new thread.

2. Is the Point class below safe in the presence of multiple threads? Why or why not? What about subclasses of Point?

```
class Point {        // is this class thread-safe?
  protected long x_, y_;

  public Point(long x, long y) { x_ = x; y_ = y; }

  public Point plus(Point other) {
    return new Point(x_ + other.x_, y_ + other.y_);
  }

  public Point minus(Point other) {
    return new Point(x_ - other.x_, y_ - other.y_);
  }
}
```

✓ Immutable, after creation, hence safe. Subclasses are safe if they are also immutable.

3. One solution to the Dining Philosophers problem is to make one Philosopher always grab his right fork first, and the other four grab their left fork first. How does this avoid deadlock?

✓ It is impossible to obtain a waits-for cycle. If the first Philosopher is waiting for the fork, then the Philosopher to his left cannot deadlock. If he has the fork, then he does not deadlock.

4. This Lock class is implemented using an instance of Slot (from the lecture notes). Does it suffer from any liveness problems? If so, what are they, and how would you fix them?

```
public class Lock {
  private Slot slot_ = new Slot();

  public synchronized void acquire() {
    slot_.put(this);
  }

  public synchronized void release() {
    slot_.get();
  }
}
```

Here, once again, is the code of the Slot class:

```
class Slot {
  private Object slotVal;   // initially null

  public synchronized void put(Object val) {
    while (slotVal != null) {
      try { wait(); }
      catch (InterruptedException e) { }
    }
    slotVal = val;
    notifyAll();
    return;
  }

  public synchronized Object get() {
    Object rval;
    while (slotVal == null) {
      try { wait(); }
      catch (InterruptedException e) { }
    }
    rval = slotVal;
    slotVal = null;
    notifyAll();
    return rval;
  }
}
```

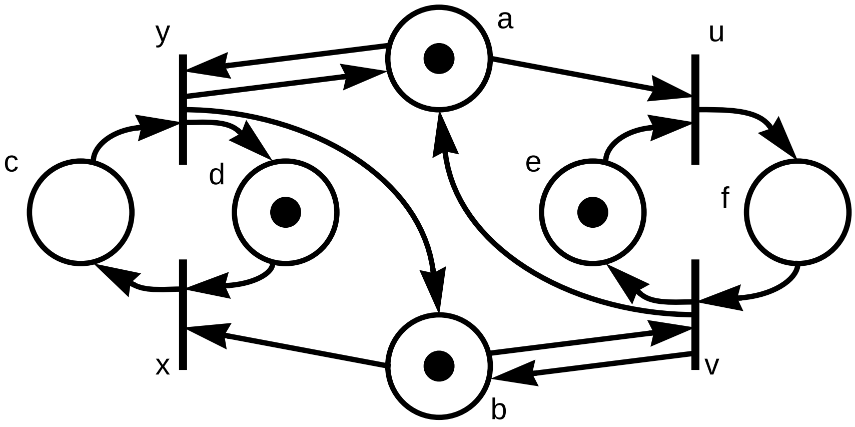
✓ There is a nested monitor problem. Two threads who try to acquire a shared lock will deadlock after the second performs an acquire(), since the Java lock on the Lock instance will still be held when the Slot instance does a wait(). Since the Lock is immutable (a relay), the best solution is to remove the synchronization of the Lock class.

5. Consider the following implementation of a Future. Would it work the same if you replace "while" by "if" in the value() method? What if you replace "notifyAll()" by "notify()" in the set() method? Justify your answers.

```
class Future {
  private Object val_ = null;
  public synchronized Object value() {
    while (val_ == null) {  // can you replace while by if?
      try { wait(); }
      catch(InterruptedException err) {}
    }
    return val_;
  }
  public synchronized void set(Object val) {
    if (val_ == null)
      val_ = val;
    notifyAll();            // can you use notify() instead?
  }
}
```

✓ Since you will only get notified when the future is set, and the condition will never change, so if should work. (Bonus point: you should still use the while loop because you might get interrupted for other reasons. Could also get woken incorrectly if someone sets the value to null!) You must use notifyAll() if there may be more than one thread waiting for the value of the Future.

6. Is the given Petri net bounded? Safe? Conservative? Are all the transitions live?



✓ It is 1-bounded (safe) since the only reachable markings are {abde}, {bfe}, {ace}, and {cf}. Clearly not conservative. {cf} is a deadlock state, so no transition is live.

What might the places and transitions represent in a real Java program?

✓ a and b represent shared resources. x and y represent one thread that tries to acquire b, then a, and u and v represent another thread that tries to grab a, then b. If each thread succeeds to grab one resource, they will deadlock.

7. What is the difference between starvation and deadlock? **(2 points)**

✓ With deadlock, no process makes progress. With starvation, one given process fails to progress.

8. How can optimistic methods livelock? How can you avoid livelock in optimistic methods?

✓ Optimistic methods attempt an update, and rollback if another thread has modified the state in the mean time. If the optimistic method simply tries again an unbounded number of times, it could repeatedly fail and rollback. Avoid livelock by bounding the number of retries, or by switching to a pessimistic strategy.

9. Is deadlock possible in a pipes and filters chain? Justify your answer.

   ✓ No, because you can’t construct a waits-for cycle.

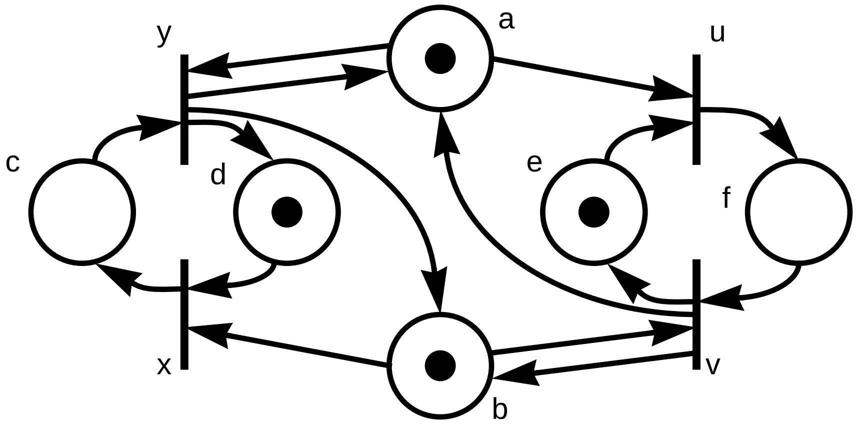
```
new Thread(this).start();
```

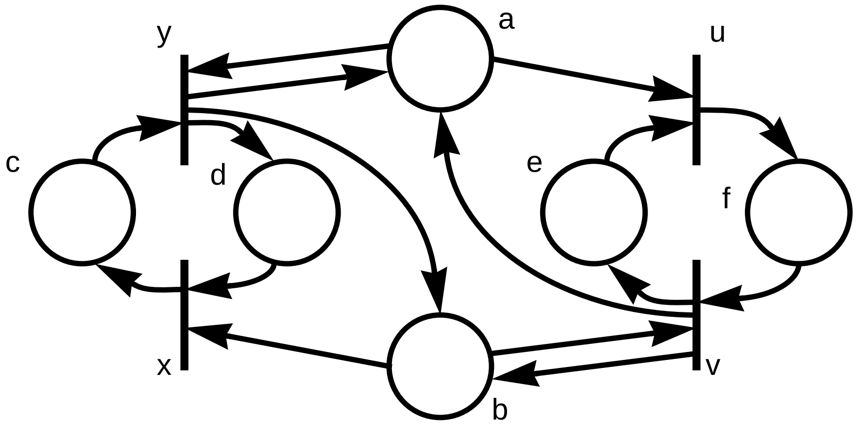
```
new Thread(this).run();
```

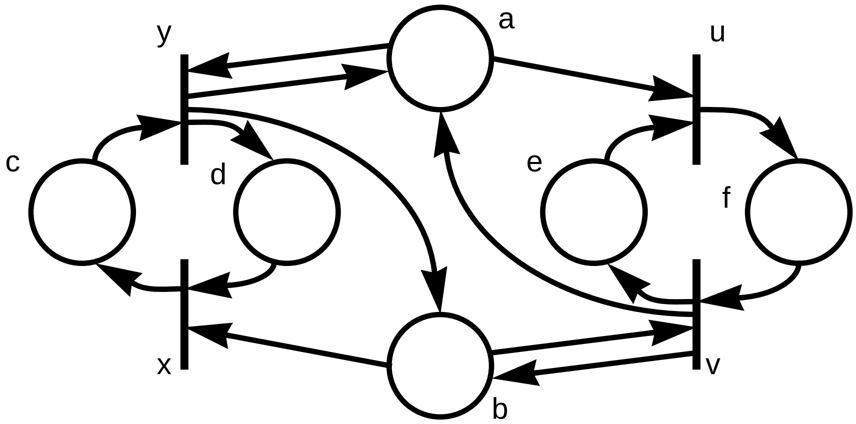
```
class Point {
  protected long x_, y_;

  public Point(long x, long y) { x_ = x; y_ = y; }

  public Point plus(Point other) {
    return new Point(x_ + other.x_, y_ + other.y_);
  }

  public Point minus(Point other) {
    return new Point(x_ - other.x_, y_ - other.y_);
  }
}
```

```
public class Lock {
  private Slot slot_ = new Slot();

  public synchronized void acquire() {
    slot_.put(this);
  }

  public synchronized void release() {
    slot_.get();
  }
}

class Slot {
  private Object slotVal;   // initially null

  public synchronized void put(Object val) {
    while (slotVal != null) {
      try { wait(); }
      catch (InterruptedException e) { }
    }
    slotVal = val;
    notifyAll();
    return;
  }

  public synchronized Object get() {
    Object rval;
    while (slotVal == null) {
      try { wait(); }
      catch (InterruptedException e) { }
    }
    rval = slotVal;
    slotVal = null;
    notifyAll();
    return rval;
  }
}
```

```
class Future {
  private Object val_ = null;
  public synchronized Object value() {
    while (val_ == null) {  // can you replace while by if?
      try { wait(); }
      catch(InterruptedException err) {}
    }
    return val_;
  }
  public synchronized void set(Object val) {
    if (val_ == null)
      val_ = val;
    notifyAll();              // can you use notify() instead?
  }
}
```


y
a
u
c
d
e
f
x
v
b


y
a
u
c
d
e
f
x
v
b


y
a
u
c
d
e
f
x
v
b

```
class A{
  B b;
  public void f(){
    synchronized(b){
      b.g();
    }
  }
}

class B{

  public synchronized void g(){
  ...
  }
}
```