1. Adapter Write an adapter from a BufferedReader to an `Iterator<String>`. The iterator should iterate over results returned by calling `readLine` on the BufferedReader, and the iteration should stop once all the lines have been read. If an I/O exception occurs, you can either just terminate the iteration with the last read String or rethrow the I/O exception (wrapped in a RuntimeException). Calling remove on the iteration cause an `UnsupportedOperationException` to be thrown.

2. Write a class with two methods, `t1()` and `t2()`, that if called by two different threads, might deadlock. Add whatever fields or other methods you need. This question isn’t asking you to write something useful, but rather to provide a very concrete description of what deadlock is. Also provide a short (English) guideline for how to avoid deadlock (it doesn’t matter if the guideline isn’t universally applicable; just describe what steps suffice in the majority of cases to avoid deadlock).
3. Figure 1 gives the skeleton for a class `LockSet` that allows you to create tokens and get locks on multiple tokens at once. Provide an implementation of the 4 incomplete methods in `LockSet` class. For example, assuming `a` and `b` are both tokens made from the `LockSet s`, the call `s.lock(a,b)` will wait until no locks are held on `a` or `b`, and obtain locks on both of them.

Your class should be immune from deadlock (assuming the threads do not make nested lock calls). Operations on two distinct `LockSet` shouldn't block each other, but it is OK if two operations on one `LockSet` momentarily block one another, even if they use disjoint tokens. Do not use spin waiting. You don't have to worry about reentrant behavior (obtaining a lock on a token you already have a lock on) or detecting errors (e.g., releasing a lock on a token that isn't locked, or passing a Token manufactured by one `LockSet` to a different `LockSet` instance.

The `LockSet` methods (lock, tryLock, ...) work as they do in the Lock interface. Note that `LockSet` uses Java varargs. For example, given the code below, both the lock and unlock methods will be passed an array of two `Token` objects as parameters.

```java
LockSet s = new LockSet();
LockSet.Token a = s.makeToken();
LockSet.Token b = s.makeToken();
s.lock(a,b);
// do something
s.unlock(a,b);
```

Feel free to use either 1.4 style synchronization (synchronized/wait/notify) or 1.5 style synchronization (e.g., using ReentrantLock).

```java
public final class LockSet {
    public final class Token {
        Token() { }
        // define any methods you need here
    }
    public Token makeToken() {
        return new Token();
    }

    public boolean tryLock(Token... locks) {
        // implement this
    }
    public void lock(Token... locks) {
        // implement this
    }
    public void lockInterruptibly(Token... locks) throws InterruptedException {
        // implement this
    }
    public void unlock(Token... locks) {
        // implement this
    }
}
```

Figure 1: `LockSet` class