Solutions to Quiz 2 (November 28, 2016)

Problem 1 (Parsers and Grammars) (**10 points**).
Consider the following grammar:

```
file ::=  record*;
@skip whitespace {
record ::= 'Name:' name ';' 'Year:' year ';' ( 'Graduate' ';')? ;
}
name ::= first (' ' middle)? ' ' last;
first ::= [A-Z][a-z]*;
last ::= [A-Z][a-z]*;
middle ::= [A-Z] '.';
year ::= [1-9];
whitespace ::= [ \n\r\t]*;
```

(a) Select from the following files the ones that can be parsed by the grammar:

File 1:
Name: Mickey Mouse;
Year: 2;
Name: Donald Duck; Year: 1;

File 2:
Name: Mickey Mouse; Year: 2;
Name: Mr. T.; Year: 1;

File 3:
Name: Donald J. Duck; Year: 1;
Name: Dora T. Explorer; Year: 2; Graduate;

File 4:
Name: Armando Solar-Lezama; Year: 2; Graduate;
Name: Max Goldman ; Year: 3;

Solution. File 1, File 3 ■

(b) Which of the following fixes to the grammar would make it parse all the above files correctly?

1) Replace `record` with
`record ::= ('Name:'  name ';')?  ('Year:'  year ';')?  ('Graduate' ';')?  ;`
2) Replace `first` with
`first ::= [A-Z][a-z\.]*;` and `last` with `last ::= [A-Z][a-z\-]*;`
3) Replace `first` with
`first ::= [A-Z][a-z]* '.'?;` and `last` with `last ::= [A-Z][a-z]* ('-' [A-Z][a-z]*)?;`
4) Replace `first` with
`first ::= [A-Z][a-z]* '.'?;` and `last` with `last ::= [A-Z][a-z]* '-'?  [A-Z][a-z]*;`

Write your answers below.

Solution. None of the fixes above will allow the grammar to parse the files correctly. In particular, "Mr. T." will be a problem for all of them. Fix 3 comes closest; if the last name allowed a dot at the end like the first name does, it would work, but none work as written. ■

Problem 2 (Concurrency and Thread Safety) (**20 points**).
We start by defining an ADT to represent a collection of people waiting in a line.

```
/**
 * Represents a set of people waiting in line.
 * Every person is represented by an ID.  */
interface WaitingLine{
    public static WaitingLine empty(){
        return new Empty();
    }
    public static WaitingLine addToLine(int id, WaitingLine rest){
        return new Sequence(id, rest);
    }
    public static WaitingLine join(WaitingLine first, WaitingLine last){
        return new LineJoin(first, last);
}    }
/**
 * Represents an empty line with nobody waiting.  */
class Empty implements WaitingLine{
    @Override
    public String toString(){
        return "";
}    }
/**
 * Represents a sequence of people with one person first and
 * the rest of the people waiting after that.  */
class Sequence implements WaitingLine{
    private final int id;
    private final WaitingLine rest;
    public Sequence(int id, WaitingLine rest){
        this.id = id;
        this.rest = rest;
    }
    @Override
    public String toString(){
        return rest.toString() + id + ", ";
}   }
/**
 * Represents two lines that have been joined together; all the people from the
 * first line come first, and all the people from the second line come second. */
class LineJoin implements WaitingLine{
    private final WaitingLine first;
    private final WaitingLine last;
    public LineJoin(WaitingLine first, WaitingLine last){
        this.first = first;
        this.last = last;
    }
    @Override
    public String toString(){
        return first.toString()+last.toString();
 }  }
```

(a) Below is an attempt at having two threads collaborate to add different person IDs to the waiting line.

```
    class State{
        private WaitingLine line = WaitingLine.empty();
        public WaitingLine getLine(){
            return line;
        }
        public void addToLine(int id){
            line = WaitingLine.addToLine(id, line);
        }
        @Override
        public String toString(){
            return line.toString();
        }
    }
    public static void try1() throws InterruptedException{
        State state = new State();
        Thread thread1 = new Thread(() -> {
            state.addToLine(1);
            state.addToLine(2);
        });
        Thread thread2 = new Thread(() -> {
            state.addToLine(3);
            state.addToLine(4);
        });
        thread1.start();
        thread2.start();
        thread2.join();
        thread1.join();
        System.out.println(state);
    }
```

Explain why the strategy above is not thread safe by giving an example of a problematic interleaving. Note that we do not care about the relative order of people added through different threads, but the final line should include all the IDs added by the threads.

Solution. A correct solution should point out that multiple threads will concurrently modify `State.line`. If two threads concurrently execute `addToLine` there is a risk that they both read the same initial line and so only the update of whichever thread writes the line last will be preserved. ■

(b) Below is a skeleton of a different solution that relies on confinement and immutability to add the same elements to the waiting line in two threads. Your goal is to complete that solution. Your code should not use locks or other data structures.

```
 public static void try2() throws InterruptedException{
        State state1 = new State();
        State state2 = new State();
        Thread thread1 = new Thread(() -> {
            //IDs 1 and 2 are added by this thread.
```

Solution.

```
state1.addToLine(1);
state1.addToLine(2);
```

■

```
        });
        Thread thread2 = new Thread(() -> {
            //IDs 3 and 4 are added by this thread.
```

Solution.

```
state2.addToLine(3);
state2.addToLine(4);
```

■

```
        });
        thread1.start();
        thread2.start();
        thread2.join();
        thread1.join();
        WaitingLine line =  /*something needs to be done here.*/
```

Solution.

```
WaitingLine.join(line1.getLine(), line2.getLine())
```

■

```
       ;
        System.out.println(line); // line should contain all the added IDs.
    }
```

Problem 3 (Map / Filter / Reduce) (**20 points**).
We are going to continue to use the WaitingLine datatype from above, but now we are going to be using Map/Filter/Reduce. In particular, for this question you should use some of the following methods:

```
List<E>:        Stream<E> stream()
Stream<T>:      <R> Stream<R>  map(Function<? super T,? extends R> mapper)
                Optional<T>  reduce(BinaryOperator<T> accumulator)
                T reduce(T identity, BinaryOperator<T> accumulator)
                Stream<T> filter(Predicate<? super T> predicate)
Optional<T>:    T get()
```

(a) For the first step, your goal is to implement the function below using only `map/filter/reduce`; you are not allowed to use iteration of any kind, only the functions listed above.

```
    /**
     * Produce a WaitingLine representing the list of IDs passed as input.
     * @param idList a list of IDs.
     * @return WaitingLine with all the IDs passed in the input list.
     */
    public static WaitingLine listToWaitingLine(List<Integer> idList){
        /* YOUR CODE HERE */
```

Solution.

```
a)
return idList.stream()
        .map((i)->(WaitingLine.addToLine(i, WaitingLine.empty())) )
        .reduce(WaitingLine::join).get();
b)
return idList.stream()
        .map((i)->(WaitingLine.addToLine(i, WaitingLine.empty())) )
        .reduce(WaitingLine.empty(), WaitingLine::join);
c) // discouraged since it uses a version of reduce not listed above, but allowed.
return idList.stream()
        .reduce(WaitingLine.empty(),
                     (l, id)-> WaitingLine.addToLine(id, l),  WaitingLine::join);
```

■

```
    }
```

(b) Now, use the function above to implement the following function, again without using explicit iteration; only `map/filter/reduce`.

```
    /**
     *
     * @param groupsOfPeople
     * @return a WaitingLine containing all the IDs of all the people in groupsOfPeople.
     */
    public static WaitingLine listlistToWaitingLine(List<List<Integer>> groupsOfPeople){
```

Solution.

```
a)
return groupsOfPeople.stream()
        .map(ThreadSafetyQuestion::listToWaitingLine)
        .reduce(WaitingLine::join).get();
b)
return groupsOfPeople.stream()
        .map(ThreadSafetyQuestion::listToWaitingLine)
        .reduce(WaitingLine.empty(), WaitingLine::join);
```

■

```
    }
```

Problem 4 (Sockets) (**10 points**).
Consider the following wire protocol:

```
request ::=  add | sub | total ;
add ::= 'ADD ' number '\n';
sub ::= 'SUBTRACT ' number '\n';
total ::= 'TOTAL\n'
number ::= [0-9]+
```

The server must keep a running tally for each session. The tally starts at zero, and `ADD n` adds `n` to that tally, while `SUBTRACT n` subtracts `n` from it. The server responds to add and sub with `'OK'`; the response for `'TOTAL'` is the value of the running total, and the response to anything else is 'ERR'.

Consider the following implementation of the tally server:

```
class TallyHandler{
    private int tally = 0;
    private final Socket socket;
    public TallyHandler(Socket socket){
        this.socket = socket;
    }
   /** handle will read from the socket, update the tally and write responses back to
    *   the socket. No side effects besides reading and writing to the socket and
    *   updating the tally. No threads are spawned by this method.
    */
    public void handle() throws IOException{/* Some code.  */ }
}
```

The server instances are launched in the following loop:

```
public static void main(String[] args) throws IOException{
        ServerSocket serverSocket = new ServerSocket(SQUARE_PORT);
        while (true) {
            // block until a client connects
            Socket socket = serverSocket.accept();
            TallyHandler handler = new TallyHandler(socket);
            Thread t = new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        handler.handle();
                    } catch (IOException ioe) {
                        ioe.printStackTrace(); // but don't terminate
                    } finally {
                        try {
                            socket.close();
                        } catch (IOException ioe){
                            ioe.printStackTrace();
                }   }   }
            });
            t.start();
        }
    }
```

(a) What is the purpose of creating and starting a new thread in the code? Why does the server not just run handler.handle() directly?

Solution. The purpose is to allow the server to support multiple clients simultaneously. Every new connection is handled by its own thread which blocks awaiting for requests on that connection. ■

Problem 5 (Queues) (**20 points**).
Now, we are going to implement another version of the TallyHandler; this time instead of sockets, this version will be designed to run on a separate thread and will communicate using blocking queues. First, we are going to define some classes to represent the requests:

```
interface Request{
    public void process(TallyHandler server);
}

class Add implements Request{
    final int value;
    public Add(int value){
        this.value = value;
    }
    @Override
    public void process(TallyHandler server) {
        server.add(value);

    }
}

class Sub implements Request{
    final int value;
    public Sub(int value){
        this.value = value;
    }
    @Override
    public void process(TallyHandler server) {
        server.sub(value);
    }
}

class Total implements Request{
    @Override
    public void process(TallyHandler server) {
        server.total();
    }
}
```

We also need a class to represent the response from the server:

```
interface Response{  }

class OK implements Response{   }

class WithValue implements Response{
    private int total;
    public WithValue(int total){
        this.total = total;
}   }
```

(a) Below is an incomplete implementation of the server. Please complete the missing code to implement the functionality described in the previous section. Your code should get the Request from the input queue `in` and write its response to the output queue `out`. You do not have to worry about erroneous requests (the ERR case).

```
class TallyHandler{
    private int tally = 0;
    private final BlockingQueue<Request> in;
    private final BlockingQueue<Response> out;

    public TallyHandler(BlockingQueue<Request> in, BlockingQueue<Response> out){
        this.in = in;
        this.out = out;
    }
    public void handle() throws IOException, InterruptedException {
        while(true){
                Request r = /* Your code to get request from queue goes here */
```

Solution.

```
in.take()
```

■

```
                ;
                r.process(this);

        }
    }
    /**
     * Add x to the tally and send a reply through the output queue.       */
    public void add(int x){
        /* Your code goes here */
```

Solution.

```
tally += x;
out.put(new OK());
```

■

```
    }
    /**
     * Subtract x from the tally and send a reply through the output queue.  */
    public void sub(int x){
        /* Your code goes here */
```

Solution.

```
tally -= x;
out.put(new OK());
```

■

```
    }
    /**
     * Return total tally through the output queue.      */
    public void total(){
        /* Your code goes here */
```

Solution.

```
out.put(new WithValue(tally));
```

■

```
    }
}
```

Problem 6 (Locks) (**20 points**).
Consider the class below:

```
class Student{
    Student partner;
    int id;
    int grade;

    int getGrade(){
        return grade;
    }
    /**
     * Change the grade for this student to newgrade.
     * If the grade of the partner is not the same as newgrade,
     * the grade of the partner should also be updated to newgrade.
     * @param newgrade new grade for student
     */
    synchronized void updateGrade(int newgrade){
        //partner always has the same grade
        grade = newgrade;
        if(partner.getGrade() != newgrade){
            partner.updateGrade(newgrade);
        }
    }
    /**
     * This method sets the current grade to be amount better than the
     * grade of the partner.
     * @param amount how much better is the grade of this student
     * relative to the grade of the partner.
     */
    synchronized void betterThanPartner(int amount){
        grade = partner.getGrade() + amount;
    }
}
```

Each student in the class is identified by a unique id, and each student has a grade. Each student also has a partner. When the grade for a student is set, the code checks if the grade for the partner is different from this new grade, and if so, it updates the grade for the partner as well. If the `betterThanPartner` method is called, then the grade for this student is set to be some amount higher than the grade of the partner.

(a) Describe a scenario in which the methods above can deadlock. Assume only two threads calling one method each.

Solution.

Two threads call updateGrade concurrently on two students that are partners of each other. For both grades, the new grade is different from the grade of the partner, so both need to call updateGrade of the partner. Both calls block waiting for the other to finish. ■

(b) Is there a different scenario (with a different combination of methods from the previous one) where deadlock is possible? If so, describe it. If no, explain why not.

Solution. No, updateGrade is the only method that can potentially try to acquire two locks. ■

(c) Rewrite exactly one of the methods above to ensure the absence of deadlock while maintaining thread safety when two threads call the different methods in `Student` concurrently.

Solution. Solution should attempt to acquire locks inside updateGrade in a consistent order.

```
Student first = this;
Student second = partner;
if(first.id > second.id){ first = second; second = this; }
 if(partner.getGrade() != newgrade){
    synchronized(first){
      synchronized(second){
            grade = newgrade;
            partner.updateGrade(newgrade);
      }
    }
 }else{
     synchronized(this){
          grade = newgrade;
     }
}
```

■