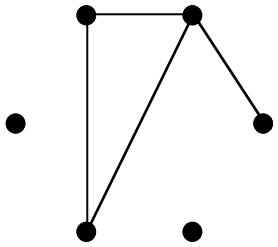
CS 61BL (Clancy) **Project 1**
Summer 2013

Submission information

A solution to this project is due at 10pm on Sunday, July 14. You will work in a partnership of two or three. Also due at that time is a set of ratings of your work and your teammates. (You get 24 "grace hours" to use toward submitting projects late.)

Background

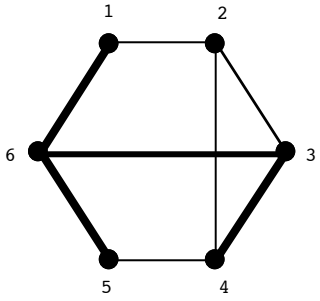
First, some vocabulary: This project involves a *board*, a diagram with *points*, some of which may be connected by *lines*. A sample board with six points, four of which have lines connecting them, appears on the right.



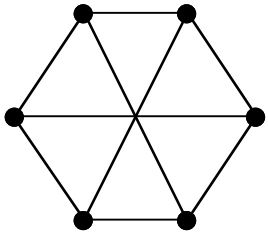
Consider the following game played on a board of six points. Two players play, one assigned the color red, the other the color blue. Each player takes turns connecting two unconnected points with a line—a "connector"—of the player's own color. If a player is forced to color a connector that makes a triangle of his or her own color—a set of three points, all connected by connectors of that color—that player loses. (There can be no ties in this game, since one may show that if all the connectors are colored, there must be a triangle of one of the colors.) [1]

Here's the start of an example game. Assume that blue moves are the bold lines.

red move: 12
blue move: 34
red move: 24
blue move: 56
red move: 23; note that 14 is a losing move for red
blue move: 36
red move: 45
blue move: 16



You may wish to play this game a few times with a friend. One strategy is to avoid coloring lines that are losing moves for your opponent. Another is to choose a move that is good for your opponent. A third makes use of the fact that the board on the right contains no triangles, but has as many lines as possible.



[1] For a more detailed explanation of the game, see *Knotted Doughnuts and Other Mathematical Entertainments* (chapter 9), by Martin Gardner, Freeman, 1986.

Problem

The directory ~cs61bl/code/proj1 contains a framework for a program to play this game and to display each move graphically. You are to complete the program, incorporating checks of the user's move and at least one nontrivial strategy for choosing among the moves available to the program. Assume that the user will make the first move, and will color his or her connector red (the program will play blue).

The framework consists of six files.

- Board.java
  implements the game board. Headers are given for methods called by the other classes. You supply all the code, including an isOK method that checks that the values of your instance variables are internally consistent, and a choice method that chooses a move for the computer player to make.
- BoardTest.java
  starts you off with some test cases.
- Connector.java
  implements a single line that connects two of the six points. Code is given for all methods except toConnector, a method that converts a text version of a line to a Connector object. You supply the toConnector code.
- Display.java
  handles the graphical display of the board. Don't modify this file.
- Driver.java
  provides a main method that plays the game. Don't modify this file.
- IllegalFormatException.java
  implements the exception thrown when the user inputs an illegal move. Don't modify this file.

The main method in Driver.java first sets up an uncolored board, then repeatedly does the following.

1. It reads a connector from the user, calling the toConnector method (analogous to "toString"). This method either throws IllegalFormatException for input that is syntactically illegal, or returns the corresponding Connector object.
2. It checks to make sure that the specified connector isn't already colored and it doesn't form a red triangle. If the move does produce a red triangle, it prints "You lose!" and exits. Otherwise, it colors the connector red. (Relevant Board methods are colorOf, formsTriangle, add, and isOK.)
3. It then requests the computer to select a connector, which will be colored blue, using the choice method in the Board class. (The choice method should avoid losing if possible.) If there is a blue triangle, the user wins; "You win!" is printed and the program exits. Otherwise, it colors the connector blue.
4. Finally, it updates the display (repaint calls paintComponent to do this).

In summary, your program should check each move typed by the user. If the move is legal, your program should respond with a nonlosing move of its own if possible, preferably a good one. When more than one move is available to the program, it should apply a tactic such as one of those mentioned above.

Detailed specifications

A legal input from the user consists of the following:

- optional white space;
- a digit between 1 and 6, inclusive;
- a different digit between 1 and 6, inclusive;
- optional white space.

The toConnector method should check for *anything* that doesn't match the specification above (including a null string), and throw IllegalFormatException accordingly. You may not make *any* assumptions about what the user types. The toConnector method should only worry about the syntax of its argument; it shouldn't care about the rest of the board (e.g. whether or not an already colored connector is input).

You may choose any representation you want for the Board. (We encourage you to use the builtin classes in the java.util library rather than reinventing the wheel.). However, you may not use any public methods other than those provided. None of the Board methods should throw an exception. All the Board instance variables should be private.

A full credit solution will avoid making a losing move if possible; it will avoid choosing a losing move for the opponent if possible; and it will implement some more clever tactic for a choice (and defend it in the "readme" file described below).

Don't change anything in the Driver, Display, or IllegalFormatException classes. Other than toConnector, don't change anything in the Connector class.

Include two JUnit files ConnectorTest.java and BoardTest.java with your submission. The ConnectorTest code should test the toConnector method. The BoardTest code should assume without verification that the connectors in the board all satisfy the invariant

```
1 <= myPoint1 < myPoint2 <= 6
```

but should test that the *collections* of connectors stored in the board are consistent with one another and represent boards that would be created as a result of playing an actual game. A couple of checks are provided in BoardTest.java to get you started.

Your additions to BoardTest.java should include tests of isOK, including cases where it should fail. The add method provides a useful way to set up inconsistent boards for this purpose.

If your program produces any output other than what's already produced in the driver program, surround the output statements by checks of the Board.iAmDebugging variable. For example,

```
if (iAmDebugging) {
   System.out.println("Connector " + cnctr + " forms " + k + " triangles");
}
```

Do not remove the debugging output statements from your program before submission.

Your code should display good style. Follow reasonable conventions for formatting and use of white space. Choose descriptive variable and method names. Organize your code—especially your tests—so that related methods and tests are grouped together. If you're writing code to handle each of each of the 6 points or each of the 15 connectors or each of the 20 triangles, use loops and auxiliary methods to do the processing, not 6 or 15 or 20 copies of copy-and-pasted code. Any methods longer than a screenful of code should probably be split into smaller pieces.

Your partnership submission will consist of five files: Connector.java, Board.java, ConnectorTest.java, BoardTest.java, and either readme.txt or readme.pdf (see below). Individually, you will submit ratings about your and your teammates' work on the project; more information about how to do this will be available around the submission due date.

The readme file

Along with your Java files, you are to provide a text file named readme.txt or a PDF file named readme.pdf that includes the following aspects of your design and development. (Note that the file names are all lower-case letters.) Convert files in other formats (e.g. .doc or .rtf) to PDF format prior to submission.

- It is likely that your board representation will either consist of one or more linear collections of connectors, or a *two-dimensional* structure that is first indexed by one endpoint of a connector and then indexed by the other. Indicate which organization more closely matches your solution, and describe why you chose it rather than an alternative.
- Describe the "clever" tactic that you implemented for choosing a move, and explain how it improves the play of the program.
- Describe your development sequence—i.e. the steps you took to construct a working program. What did you code first, and what did you test first? What did you do next? How did you choose test cases at each stage? And so on.
- Finally, explain what each of your team members contributed to the solution.

Miscellaneous requirements

In lab activities on July 5, partnerships will meet with course staff to evaluate your plan for completing the project. Lab activities on July 11 will include an interview with course staff about how much progress you've made toward a solution. A small number of homework points will be awarded for this activity, based on how far along you are. To get all the points, you need to at least be able to make a legal move or to detect an illegal move, and have tests that demonstrate these capabilities. (There is an exam on July 10; we suggest that you start early on the project.)

Please confine your detailed discussion about the project to your partnership as described in the "General Information" document.

Grading

Points for components of your solution will be allotted as follows.

Board.java	30%	Connector.java	16%	readme file	22%
BoardTest.java	16%	ConnectorTest.java	16%		

Appendix: Information on Iterator objects

If you're using an ArrayList<Connector> to store your connectors, you may call the method named iterator. Here's an example:

```
ArrayList<Connector> values = new ArrayList<Connector>;
   ...
Iterator<Connector> valuesIter = values.iterator ( );
```

If instead you are using an array and therefore need to build your own next and hasNext, you should use a *nested class*. Here's an example of an iterator for IntSequence.

```
// Inside the IntSequence class
private class SeqIterator implements Iterator<Integer> {

   private int iterIndex;

   public SeqIterator ( ) {
      iterIndex = 0;
   }

   public boolean hasNext ( ) {
      return iterIndex < myCount;
   }

   public Integer next ( ) {
      Integer rtn = new Integer (myValues[iterIndex]);
      iterIndex++;
      return rtn;
   }

   // Required in Iterator<Integer> interface but not needed by us.
   public void remove ( ) {
   }
}

public SeqIterator valueIter ( ) {
   return new SeqIterator ( );
}
```

You can then construct and use a SeqIterator object as follows.

```
   // Inside the main method
   IntSequence s = new IntSequence (10);
      ...
   SeqIterator iter;
   iter = s.valueIter ( );
   while (iter.hasNext ( )) {
      System.out.println (iter.next ( ));
   }
```