CS 61B Project 2
Network (The Game)
Due noon Wednesday, April 2, 2014
Interface design due in lab March 13-14

Warning: This project is substantially more time-consuming than Project 1.
Start early.

This is a team project. Form a team of 2 or 3 people. No teams of 1 or teams of 4 or more are allowed. Your project partners do NOT have to be in your lab.

Copy the Project 2 directory by doing the following, starting from your home directory.

```
cp -r ~cs61b/hw/pj2 .
cd pj2
```

Suggested Timeline (if you want to finish on time)
==========

Design the classes, modules, and interfaces (see "Teamwork").	March 13-14
Have working code for the easier modules.	March 14
Have working code for identifying a network; progress on search.	March 21
Finish project.	April 2

Network
=======
In this project you will implement a program that plays the game Network against a human player or another computer program. Network is played on an 8-by-8 board. There are two players, "Black" and "White." Each player has ten chips of its own color to place on the board. White moves first.

	10	20	30	40	50	60	
01	11	21	31	41	51	61	71
02	12	22	32	42	52	62	72
03	13	23	33	43	53	63	73
04	14	24	34	44	54	64	74
05	15	25	35	45	55	65	75
06	16	26	36	46	56	66	76
	17	27	37	47	57	67	

The board has four goal areas: the top row, the bottom row, the left column, and the right column. Black's goal areas are squares 10, 20, 30, 40, 50, 60 and 17, 27, 37, 47, 57, 67. Only Black may place chips in these areas. White's goal areas are 01, 02, 03, 04, 05, 06 and 71, 72, 73, 74, 75, 76; only White may play there. The corner squares--00, 70, 07, and 77--are dead; neither player may use them. Either player may place a chip in any square not on the board's border.

Object of Play
==========
Each player tries to complete a "network" joining its two goal areas.
A network is a sequence of six or more chips that starts in one of the player's goal areas and terminates in the other. Each consecutive pair of chips in the sequence are connected to each other along straight lines, either orthogonally (left, right, up, down) or diagonally.

The diagram below shows a winning configuration for Black. (There should be White chips on the board as well, but for clarity these are not shown.) Here are two winning black networks. Observe that the second one crosses itself.

```
60 - 65 - 55 - 33 - 35 - 57
20 - 25 - 35 - 13 - 33 - 55 - 57
```

		BB				BB		_0
								_1
					BB			_2
	BB		BB					_3
								_4
		BB	BB		BB	BB		_5
								_6
						BB		_7
0_	1_	2_	3_	4_	5_	6_	7_	

An enemy chip placed in the straight line between two chips breaks the connection. In the second network listed above, a white chip in square 56 would break the connection to Black's lower goal.

Although more than one chip may be placed in a goal area, a network can have only two chips in the goal areas: the first and last chips in the network. Neither of the following are networks, because they both make use of two chips in the upper goal.

```
60 - 20 - 42 - 33 - 35 - 57
20 - 42 - 60 - 65 - 55 - 57
```

A network cannot pass through the same chip twice, even if it is only counted once. For that reason the following is not a network.

```
20 - 25 - 35 - 33 - 55 - 35 - 57
```

A network cannot pass through a chip without turning a corner (i.e. changing direction). Because of the chip in square 42, the following is not a network.

```
60 - 42 - 33 - 35 - 55 - 57
```

Legal Moves
==========
To begin the game, choose who is Black and who is White in any manner (we use a random number generator). The players alternate taking turns, with White moving first.

The first three rules of legal play are fairly simple.
1) No chip may be placed in any of the four corners.
2) No chip may be placed in a goal of the opposite color.
3) No chip may be placed in a square that is already occupied.

The fourth rule is a bit trickier.
4) A player may not have more than two chips in a connected group, whether connected orthogonally or diagonally.

This fourth rule means that you cannot have three or more chips of the same color in a cluster. A group of three chips form a cluster if one of them is adjacent to the other two. In the following diagram, Black is not permitted to place a chip in any of the squares marked with an X, because doing so would form a group of 3 or more chips. (Of course, the far left and right columns are also off-limits to Black.)

```
-----------------------------------------
|    |  X |  X | BB |  X |    |    |    |
-----------------------------------------
|    |  X | BB |  X |  X |  X |  X |    |
-----------------------------------------
|    |  X |  X |  X |  X | BB |  X |    |
-----------------------------------------
|    |    |    |    |  X | BB |  X |    |
-----------------------------------------
|    |    | BB |    |  X |  X |  X |    |
-----------------------------------------
|    |  X |  X |    |    |    | BB |    |
-----------------------------------------
|    | BB |    |    |    |  X |    |    |
-----------------------------------------
|    |    |    |    | BB |    |    |    |
-----------------------------------------
```

There are two kinds of moves: add moves and step moves. In an add move, a player places a chip on the board (following the rules above). Each player has ten chips, and only add moves are permitted until those chips are exhausted. If neither player has won when all twenty chips are on the board, the rest of the game comprises step moves. In a step move, a player moves a chip to a different square, subject to the same restrictions. A player is not permitted to decline to move a piece (nor to "move from square ij to square ij").

A step move may create a network for the opponent by unblocking a connection between two enemy chips. If the step move breaks the network at some other point, the enemy does not win, but if the network is still intact when the chip has been placed back on the board, the player taking the step move loses. If a player makes a move that causes both players to complete a network, the other player wins.

To make sure you understand the rules, try playing a few games against your project partners. See the instructions in "Running Network" below. Or, use ten pennies, ten silver coins, and a checkerboard.

Bibliographic note: Network is taken from Sid Sackson, "A Gamut of Games," Dover Publications (New York), 1992.

Your Task
=========
Your job is to implement a MachinePlayer class that plays Network well. One subtask is to write a method that identifies legal moves; another subtask is to write a method that finds a move that is likely to win the game.

The MachinePlayer class is in the player package and extends the abstract Player class, which defines the following methods.

```
  // Returns a new move by "this" player.  Internally records the move (updates
  // the internal game board) as a move by "this" player.
  public Move chooseMove();

  // If the Move m is legal, records the move as a move by the opponent
  // (updates the internal game board) and returns true.  If the move is
  // illegal, returns false without modifying the internal state of "this"
  // player.  This method allows your opponents to inform you of their moves.
  public boolean opponentMove(Move m);

  // If the Move m is legal, records the move as a move by "this" player
  // (updates the internal game board) and returns true.  If the move is
  // illegal, returns false without modifying the internal state of "this"
  // player.  This method is used to help set up "Network problems" for your
  // player to solve.
  public boolean forceMove(Move m);
```

In addition to the methods above, implement two constructors for MachinePlayer.

```
  // Creates a machine player with the given color.  Color is either 0 (black)
  // or 1 (white).  (White has the first move.)
  public MachinePlayer(int color)

  // Creates a machine player with the given color and search depth.  Color is
  // either 0 (black) or 1 (white).  (White has the first move.)
  public MachinePlayer(int color, int searchDepth)
```

As usual, do not change the signatures of any of these methods; they are your interface to other players. You may add helper methods.

Your MachinePlayer must record enough internal state, including the current board configuration, so that chooseMove() can choose a good (or at the very least, legal) move. In a typical game, two players and a referee each have their own internal representation of the board. If all the implementations are free of bugs, they all have the same idea of what the board looks like, although each of the three uses different data structures. The referee keeps its own copy to prevent malicious or buggy players from cheating or corrupting the board. If your MachinePlayer is buggy and attempts to make an illegal move, the referee will grant the win to your opponent.

Most of your work will be implementing chooseMove(). You will be implementing the minimax algorithm for searching game trees, described in Lecture 17. A game tree is a mapping of all possible moves you can make, and all possible responses by your opponent, and all possible responses by you, and so on to a specified "search depth." You will NOT need to implement a tree data structure; a "game tree" is the structure of a set of recursive method calls.

The forceMove() method forces your player to make a specified move. It is for testing and grading. We can set up particular board configurations by constructing a MachinePlayer and making an alternating series of forceMove() and opponentMove() calls to put the board in the desired configuration. Then we will call chooseMove() to ensure that your MachinePlayer makes a good choice.

The second MachinePlayer constructor, whose second parameter searchDepth is the

chosen search depth, is also used for debugging and testing your code.
A search depth of one implies that your MachinePlayer considers all the moves and chooses the one that yields the "best" board. A search depth of two implies that you consider your opponent's response as well, and choose the move that will yield the "best" board after your opponent makes the best move available to it. A search depth of three implies that you consider two MachinePlayer moves and one opponent move between them.

The first MachinePlayer constructor should create a MachinePlayer whose search depth you have chosen so that it always returns a move within five seconds. (This precise time limit will be important only for the Network tournament late in the semester.) The second MachinePlayer constructor MUST create a MachinePlayer that always searches to exactly the specified search depth.

You may want to design the MachinePlayer constructed by your first constructor so that it searches to a variable depth. In particular, you will almost certainly want to reduce your search depth for step moves, because there are many more possible step moves than add moves, and a search depth that is fast for add moves will be very slow for step moves.

The Move class in Move.java is a container for storing the fields needed to define one move in Network. It is not an ADT and it has no interesting invariants, so all its fields are public. It is part of the interface of your MachinePlayer, and it is how your MachinePlayer communicates with other programs, so you cannot change Move.java in any way. If you would like to have additional methods or fields, feel free to extend the Move class; your MachinePlayer may return subclasses of Move without any fear.

Strategy
========
Where should you start? First, design the structure of your program (see "Teamwork" below). Then begin by writing a relatively simple MachinePlayer class that simply chooses some correct move, no matter how bad. These actions will give you partial credit on the project. Based on that foundation, you can implement something more sophisticated that incorporates strategy.

Game trees rely on an "evaluation function" that assigns a score to each board that estimates how well your MachinePlayer is doing. An evaluation function is necessary because it is rarely possible to search all the way to the end of the game. You need to estimate your odds of winning if you make a particular move. Your evaluation function should assign a maximum positive score to a win by your MachinePlayer, and a minimum negative score to a win by the opponent.

Assign an intermediate score to a board where neither player has completed a network. One of the most important but difficult parts of implementing game search is inventing a board evaluation function that reliably evaluates these intermediate boards. For example, a rough evaluation function might count how many pairs of your chips can see each other, and subtract the opponent's pairs. A slightly better evaluation function would also try to establish at least one chip in each goal early in the game. I leave you to your own wits to improve upon these ideas.

You should assign a slightly higher score to a win in one move than to a win in three moves, which should get a higher score than a win in five moves, and so on. Otherwise, your MachinePlayer might always choose the win in three over the win in one, move after move, and never get around to actually winning.

You will need to invent an algorithm that determines whether a player has a winning network. A good place to look for clues is Section 13.3 of Goodrich and Tamassia, which describes depth-first search in graphs. It's not quite what you need for the job, but close enough that you'll be able to modify it. This is the hardest part of the whole project, so assign it to your smartest teammate and START EARLY!!!

To earn full credit, you must implement alpha-beta search, which is discussed in Lecture 17. Alpha-beta search is a technique for "pruning" a game tree, so you don't need to search the entire tree. Alpha-beta search can be significantly faster than naive tree search. You can earn partial credit by implementing game tree search without pruning. If you can't get that working, you can earn a little bit of partial credit by looking ahead one move.
This is the second-hardest part of the project, but the closer you stick to the Lecture 17 pseudocode, the more likely you are to succeed.

You will almost certainly want to create a separate class to represent game boards internally. One decision you will have to make is whether to create a new game board or change an existing one each time you consider a move. The latter choice is faster (i.e. more likely to win the tournament), but it's also more bug-prone; it could cause very hard-to-solve bugs if you're not extremely careful about how and when you manipulate game boards.

Late in the semester, we will hold a tournament pitting student MachinePlayers against each other. Participation in the tournament is optional and does not affect your grade. You will submit your contestant several weeks after the Project 2 due date, so you will have time to improve your MachinePlayer's evaluation function and strategy in April. During the tournament, we will strictly enforce a time limit of five seconds (which will be checked by our refereeing software) on the time to perform one chooseMove(). The winning team will receive gift certificates to Amoeba Music or Amazon.

This is a difficult project. Do not wait to start working on it. If you don't have the code that identifies networks implemented by March 21, I advise you to wallow in neurotic spasms of fear and worry. We will have autograder software set up to test your submitted code for legal moves.

Teamwork (10% of project grade) (show to your TA in Lab 8, March 13-14)
========
Before you start programming, read the Lecture 18 notes carefully, then break the project up into multiple modules (tasks). Decide what high-level methods and classes must be implemented, define the interfaces by which these methods and classes will communicate, and divide up the work among your team. Some possible modules (these seem reasonably modular) are

1) determining whether a move is valid,
2) generating a list of all valid moves,
3) finding the chips (of the same color) that form connections with a chip,
4) determining whether a game board contains any networks for a given player (very difficult; put your smartest teammate on this),
5) computing an evaluation function for a board (possibly difficult), and
6) performing minimax tree search (difficult).

The file GRADER provided in the pj2 directory includes a questionnaire, which you are required to submit. Once you've worked out your classes, modules, and interfaces, write them down at the bottom of GRADER. Your description should include:

- A list of the classes your program will need.
- A list of each of the "modules" used in or by MachinePlayer, which might be similar to, but more detailed, than the list above.
- For each module, list the class(es) the module will be implemented in. It may (or may not) make it easier for you to work as a team if each module is in a separate class.
- For each module, describe its interface--specifically, the prototype and behavior of each method that is available for external callers (outside the module) to call. Don't include methods that are only meant to be called from within the module. For each method, provide (1) a method prototype and (2) a complete, unambiguous description of the behavior of the method/module. (This description will also appear before the method in your code's comments.)
- Which partner is assigned the task of implementing each module?

If you have defined your classes, modules, and module interfaces well, you should be able to implement any one of the modules without having decided how

to implement any of the others. This will allow you to divide up the chores and work quickly as a team. See the Lecture 18 notes for details.

You should have a draft of your GRADER file ready to show your TA in Lab 8 (March 13-14). Your Lab 8 score depends on having a finished draft of your modules and interfaces. Your TA will comment on your design decisions.

You may change some of your design decisions based on your TA's feedback, and you will probably make other changes as you program. Be sure to update your GRADER to reflect these changes. The GRADER file you submit with this project should reflect the FINAL decisions you make about modules and interfaces. Before you submit, make sure your GRADER file tells us who _actually_ implemented each portion of your project.

Your design of classes and interfaces will be worth 10% of your project grade.

Running Network

You can run Network from your pj2 directory in several ways.

```
java Network human random
```
This pits you against a very naive machine player that makes random legal moves. Use this to learn to play the game. The human plays white and the random player play black. To reverse colors, swap "human" and "random".

```
java Network human human
```
Compete against your project partner.

```
java Network human machine
```
Compete against your MachinePlayer.

```
java Network machine random
```
Your MachinePlayer competes against the random player.

```
java Network machine machine
```
Your MachinePlayer competes against itself.

Every combination of "machine", "human", and "random" works. It's amusing to pit two random players against each other.

If you put a "-q" switch right after the word "Network", Network will quit immediately when the game ends. This can be useful for batch testing of evaluation functions.

Grading

Your project will be graded in part on correctness and the quality of moves chosen by chooseMove(). This grading will be done with automatic test cases. Be sure the following statements apply to your chooseMove().

1) forceMove and opponentMove return true if the given move is legal.
2) forceMove and opponentMove return false if the given move is illegal.
3) chooseMove returns only legal moves.
4) If a winning move exists, chooseMove selects one. (This will happen automatically if you are searching one level of the game tree.)
5) If you cannot win in this step, but can prevent your opponent from winning during its next move, chooseMove selects a move that does this. (This will happen automatically if you are searching two levels of the game tree.)
6) Your player can beat the random player almost every time. Any reasonable search strategy and evaluation function should accomplish this.

You will also be graded on style, documentation, efficiency, and the use of encapsulation.

1) Every method must be preceded by a comment describing its behavior unambiguously. These comments must include descriptions of what each parameter is for, and what the method returns (if anything). They must also include a description of what the method does (though not necessarily how it does it) detailed enough that somebody else could implement a method that does the same thing from scratch, using only the comments and this readme file.

   Some methods serve as entry points to the modules you designed when you began the project. The prototypes and behavioral descriptions of these methods are interfaces, and should be included in GRADER.
2) All classes, fields, and methods must have the proper public/private/protected/package qualifier. We will deduct points if you make things public that could conceivably allow a user to corrupt the data structure.
3) There are no asymptotic limits on running time. However, part of your job is to avoid using inefficient algorithms and data structures. If your MachinePlayer takes longer than 5 seconds to search to a depth of two on a Soda lab machine, it's too slow.
4) You should have divided up the tasks into well-defined modules in your GRADER file and in your software.
5) We will deduct points for code that does not match the following style guidelines.

- Classes that contain extraneous debugging code, print statements, or meaningless comments that make the code hard to read will be penalized. (It's okay to have methods whose sole purpose is to contain lots of debugging code, so long as your comments inform the reader who grades your project that he can skip those methods. These methods should not contain anything necessary to the functioning of your project.)
- Your file should be indented in the manner enforced by Emacs (e.g., a two-space or four-space indentation inside braces) and used in the lecture notes throughout the semester. The indentation should clearly show the structure of nested statements like loops and if statements. Sloppy indention will be penalized.
- All if, else, while, do, and for statements should use braces. (Ask me if you want to know why.)
- All classes start with a capital letter, all methods and (non-final) data fields start with a lower case letter, and in both cases, each new word within the name starts with a capital letter. Constants (final fields) are all-capital-letters only.
- Numerical constants with special meaning should always be represented by all-caps "final static" constants.
- All class, method, field, and variable names should be meaningful to a human reader. (Exception: short loop index variables like "i" are okay if their meaning is obvious from context.)
- Methods should not exceed about 120 lines. Any method that long can probably be broken up into logical pieces. The same is probably true for any method that needs more than 8 levels of indentation.
- Avoid unnecessary duplicated code; if you use the same (or very similar) fifteen lines of code in two different places, those lines should probably be a separate method call.
- Programs should be easy to read.

Finally, we will be looking at your code to see whether you have implemented minimax game tree search, and whether you use alpha-beta pruning.

```
Submitting your Solution
========================
Be sure that you have updated all your answers in GRADER before submitting to
reflect your final code and tell us who wrote which module.  Don't forget that
it's worth 10% of your grade.

Designate one member of your team to submit the project.  If you resubmit, the
project should always be submitted by the same student.  If for some reason a
different partner must submit (because the designated member is out of town,
for instance), you must send cs61b@cory.eecs a listing of your team members,
explaining which of them have submitted the project and why.  Let us know which
submission you want graded.  If you've submitted your project once, or even
written a substantial amount of code together, you may not change partners
without the permission of the instructor.

The designated teammate only:  change (cd) to your pj2 directory, which should
contain the player directory (i.e. the player package), which should contain
your MachinePlayer.java and other Java files.  You may also submit other
packages in your pj2 directory (e.g. a list package).  Type "submit pj2".
The submit program will not submit Move.java and Player.java, because you're
not allowed to change them.
```