In Lab 2, you will program a player for the Domineering game. A program that performs plays Domineering with a simple evaluation function and a depth 1 search is provided to you. It is up to you to improve this program with your own evaluation function, with Minimax search, with Alpha-Beta Pruning), and/or with intelligent time management. All of your programs will be played against each other. Prizes will be given for the top five programs (that are not late). In general, your grade on the lab will depend on the quality of the player that is implemented.

**Domineering**

Domineering is played on an $n$-by-$n$ board. We will use $n = 8$. The board is initially empty:

```
  1 2 3 4 5 6 7 8
+-----------------+
1 |                 |
2 |                 |
3 |                 |
4 |                 |
5 |                 |
6 |                 |
7 |                 |
8 |                 |
+-----------------+```

The two players, vertical and horizontal, alternate turns, with vertical playing first. On each turn, a player puts a domino (a $2 \times 1$ piece) on the board. Player vertical places dominos vertically, and player horizontal places dominos horizontally. No domino can be played on top of another domino, and no domino can be moved once it is played. For example, vertical (also called player 1) might choose row 7, column 2, which results in the following position:
Now horizontal (player 2) can put a domino in row 2, column 1, which results in this position:

```
1 2 3 4 5 6 7 8
+-------------------------------+
1 | | 1
2 | + + + + + + + + +
3 | + + + + + + + + +
4 | + + + + + + + + +
5 | + + + + + + + + +
6 | + +++ + + + + + +
7 | | 1 | | 7
8 | | 1 | | 8
+-------------------------------+
1 2 3 4 5 6 7 8
```

A player loses the game when it is the player’s turn, but cannot make a move. For example, in this position, it is player 1’s turn to move, but there is no move for a vertical domino.
Programming Framework

The lab5.zip download for this lab contains Interact2.java, Domineering.java, SimplePlayer.java and HumanPlayer.java. In a valiant attempt to avoid the problems of running my multithreaded programs on Windows, each `out.println` has been replaced with a `out.print` call followed by an `out.write(10)` call. This is to avoid writing the two-character newline sequence of Windows text files.

Interact2.java sets up the threads for one Domineering object and two SimplePlayer objects. One SimplePlayer object is agent 1; Interact2 prints a you are agent 1 message to this player. The other SimplePlayer object is player 2; Interact2 prints a you are agent 2 message to this player.

Any line from an agent is output to the Domineering object with a 1 or a 2 appended in front, indicating agent 1 or agent 2. Any line from Domineering has a 1 or a 2 in front indicating which agent the line should go to.

Each agent inputs and outputs moves according to whose turn it is. The Domineering object runs the game; it will output a win for agent X line when one agent wins the game, where X is 1 or 2.

The SimplePlayer code is already written to follow the protocol. You should copy this code to another class, naming it YourNameHerePlayer, replacing YourNameHere with your name. Besides changing the name, the only methods you need to change/replace are chooseMove and simpleEval. As already mentioned, the chooseMove method in SimplePlayer only looks one move ahead.

If you want to see the game as it progresses, set the debug variable in Domineering.java to true. If you want to play against SimplePlayer, change the initial assignment to agent1 or agent2 in Interact2.java to an instance of HumanPlayer.
Agent

Your task is to write a Domineering player that will improve on SimplePlayer.java. Here is the list you need to consider, arranged from most important to least important (or so your instructor thinks).

1. **First Moves.** The first moves are important for performing well. See the following article (which can be found on the internet): Julian West. Championship-level play of domineering. In *Games of No Chance*, edited by Richard Nowakowski, MSRI Publications, 1996. Programming these in will also help with time management.

2. **Evaluation Function.** You need an evaluation function that will return a reasonable judgment about the value of the board. What follows is a suggestion. If you can think of something better, use it. I would look for the amount of space on the board available to each player, or similarly, look for the number of dominos that each player can potentially play. It might be that certain squares are more valuable than others.

   Another feature to look for is if all the moves are forced. That is, at a certain point in the game, all the empty space is either vertical or horizontal, but not both. In this case, the winner of the game can be determined.

3. **Alpha-Beta Pruning.** Alpha-beta pruning will allow the depth of your search to be 2 or 3 moves deeper compared to minimax, so a program with alpha-beta pruning will have an advantage over minimax even with a unsophisticated evaluation function. The best results for alpha-beta pruning come if the moves are in considered in order of the evaluation function.

4. **Time Management.** Your program gets 30 seconds to make all of its moves. Early in the game, the branching factor is 56. Later in the game as the board fills up, the branching factor decreases rapidly, so you can search to a greater depth, perhaps all the way to the end of the game.

   One possibility is to do an iterative deepening version of alpha-beta pruning/minimax. This would perform searches to depth 1, then depth 2, then depth 3, and so on until some internal time limit is exceeded. Of course, I would choose the move chosen by the deepest search. You might choose to do something like 1 CPU second/move, or maybe something more sophisticated based on how much time you have left.

Prizes

I will be running your lab against the other students’ labs and programs of my own (versions of SimplePlayer.java and one competitive program). The prizes (my programs are not eligible) for the top five programs will be 50, 40, 30, 20, and 10 points, respectively. In the event of ties, the prizes will be distributed evenly.

Otherwise, your grade on the lab will depend on the performance of your program. I will supply some jar files later that will include versions of the SimplePlayer program that perform deeper searches.
1. A score of 80. Your program wins both games vs. SimplePlayer, depth 1, or performs better than this program in the all-students tournament.

2. A score of 90. Your program wins 3 out of 4 games vs. Simple Player, depth 1 and depth 2, or performs better than both of these programs in the all-students tournament.

3. A score of 100. Your program wins 5 out of 6 games vs. Simple Player, depth 1, depth 2 and depth 3, or performs better than all three programs in the all-students tournament.