CS61B, Fall 2009 **Project #3: Reversi** **P. N. Hilfinger**

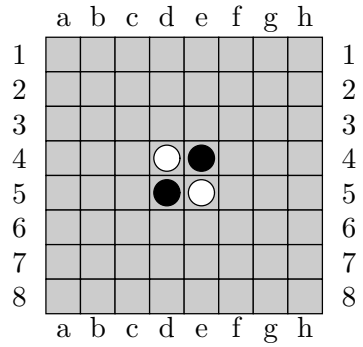
Due: Monday, 7 December 2009 at 2400

This is version 4, dated 18 November 2009.

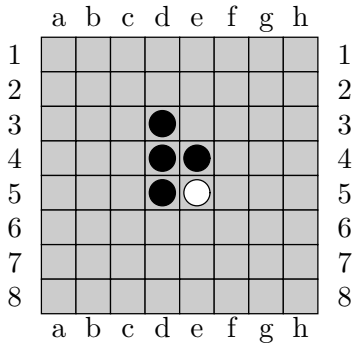
1 Background

Reversi[1] is a familiar two-person game played on an 8-by-8 square grid with white and black pieces. Beginning with the position shown on the left below, play alternates between players, with black going first. Each move consists of placing a piece of one's own color on an empty square so that there is a piece of the same color on the same row, column, or diagonal separated from the empty square by one or more squares that are all filled with enemy pieces (which we'll say are then *surrounded*). After making such a move to an empty square, all newly surrounded enemy pieces (there must be at least one) reverse their color (these reversals of color, however, do not cause any other pieces to become surrounded). If no such move is possible, the player passes. When both players must pass in succession, the game ends and the player with the most pieces on the board wins.

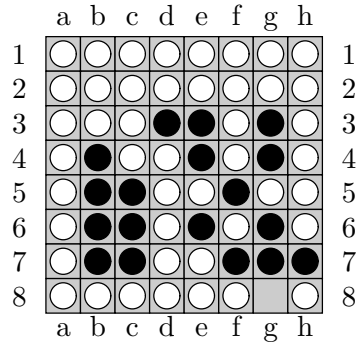
[1] If we are to believe Wikipedia, Lewis Waterman invented the game in 1883, and it was re-invented without proper attribution as the game Othello$^{(R)}$ in the 1970's. I will use "Reversi," that being the older name and I being a traditionalist at heart.



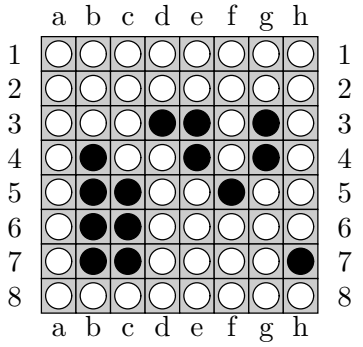
Starting position



Position after black moves to 3d



Position just before white's last move



Final position: white wins

1.1 Notation

We'll denote columns with letters a–h from the left and rows with numerals 1–8 from the top, as shown in the illustration of the initial position on the previous page. An ordinary move consists of a column letter followed by a row number, as in `d3`. A single hyphen (`-`) denotes a case where one player must skip a move.

2 Textual Input Language

From your program's point of view, we'll refer to the two players as the *Player* and the *Opponent.* The Player is either the person using the program (the *User*) or possibly an automated player (we'll call it an *AI* for short). The Opponent is either another *AI* or a remote opponent (i.e., some other Reversi program). In any given game, either of these may play black.

Commands are one to a line, with the operand (if any) separated from the command by whitespace. Additional whitespace is legal around the command. Comments begin with a '`#`' and proceed to the end of a line, and are ignored. Empty commands (containing only whitespace and comments) are likewise ignored.

Commands to begin and end a game.

clear Valid when no game is in progress (that is, before any game is played, or after someone has won). Clears the board to its initial configuration.

`start` Valid only when no game is in progress. Begins a game, using the parameters currently set for color, seed, and kind of player. Takes moves alternately from the Player and Opponent according to their color and the current move number, starting from the initial state of the board. If there have been moves made before play starts (to set up a partially played game) then play picks up at the point where these moves leave off (so, for example, if the Player is black and there was one move made before '`start`', then the Opponent will move first). In the (unusual) case where the set-up moves have already won the game, `start` causes simply causes the program to announce a winner and immediately end the game.

`quit` Abandons any current game and exits the program. Valid in any state.

Set-up. The following commands are valid only when a game is not in progress, and set various parameters for the next game.

`color` *C* Sets the Player's color to *C*, which may be '`white`' or '`black`'. By default the Player plays black.

`auto` Sets up the program so that the Player is an AI when play starts. Thus, when playing locally, '`auto`' causes both the Player and Opponent to be AIs, so that the `start` command causes the machine to play a game against itself. Initially, the Player is the User.

`manual` Sets up the program so that the Player is the User.

`seed` *N* *If* your program's AIs use pseudo-random numbers to choose moves, this command reseeds the pseudo-random number generator with *N* (a long integer). This command has no effect if there is no random component to your automated players (or if you don't use them in a particular game). It doesn't matter exactly how you use *N* as long as your automated player behaves identically in response to any given sequence of moves from its opponent each time it is seeded with *N*. In the absence of a `seed` command, do what you want to seed your generator.

Making moves. To make a move (that is, place a piece), one simply enters the move using the notation in §1.1 above. The effect of an ordinary move is to set a black or white piece, depending on whose move it is, and flip any pieces indicated by the rules. The effect of a pass ('`-`'), is to do nothing but change the player on move. Passes are legal only when no move is possible for the current player. Moves must be legal, or your program must reject them without affecting the board (since humans are expected to make errors, your program should ask for another move when this happens). Your AI should never even try an illegal move (at least visibly) and when playing a remote opponent (see "Remote play"), your program must

never send an illegal move. While a game is in progress (after a `"start"` command), a player may resign with the command `"resign"`, which ends the game. If you receive an erroneous move from a remote Opponent, report an error and treat it as if the opponent resigned. As soon as neither player can move, the game is over and neither player may move (not even to pass) until the board is cleared. Likewise, if a game ends because a player resigns, no player may move until the board is cleared.

Moves that the Player makes before a game starts serve to set up a starting position. That is, any moves before play starts place black and white pieces alternately (as if both players were playing). When play starts, the move goes to black if there have been an even number of moves made, and to white otherwise, just as in ordinary play.

Remote play. The following two commands are valid only when a game is not in progress, and the program is not currently connected to a remote opponent. They cause the Opponent to become a remote player, whose moves come from another execution of the program when play starts (see also §4 for technical details).

`host` *ID* Here, *ID* is any sequence of letters, underscores, and digits. This first executes a `clear` command. Next, the program waits for an opponent to join (see the '`join`' command). It then becomes the *host* of the next game.

`join` *ID*`@`*hostname* There may not be any whitespace in "*ID*@*hostname*." First, execute a `clear` command. There must be a program running on the machine *hostname* (a name like `nova.cs.berkeley`, or, for another program running on the same machine, `localhost`); its user must have entered the `host` *ID* command; and nobody else can have joined the same game. The program becomes the *visitor* of the next game.

After two players have entered these commands (one becoming the host and the other the visitor), the visitor program starts receiving commands from the host, which are executed as if by the User of the visiting program. When the host program sends a `start` command, the visiting program regains control, and all further commands from the host become the Opponent's moves.

The host program will first send a '`color`' command to synchronize the colors of the two programs. If the host's Player is currently set to play black, the host sends '`color white`', so that the visitor's Player will play white, and vice-versa. Any following '`color`' or '`clear`' commands by the host program's player get relayed to the visitor (with '`color`' commands reversed), as do any set-up moves. This ends when the host sends a '`start`' command or breaks the connection. When the host finally sends a `start` command, the client program will alternate reading commands from its Player and from the (now remote) Opponent, as usual (which one goes first depends on whose move it is, as usual).

Because various ISPs and University system administrators block various ports (even on the local network), it may or may not be possible to communicate to a remote machine. However, the host `localhost` (meaning "this computer") will always work, allowing you to talk to other programs running on the same machine. So if both programs are running on `torus`, say, remote communication should work.

Miscellaneous commands. The following commands are valid in any state.

`help` Print a brief summary of the commands.

`dump` This command is especially for testing and debugging. It prints out the board and some other information in *exactly* the following format:

```
===
  - - - - - - - -
  - - - - - - - -
  - - - - - - - -
  - - - w b - - -
  - - - b w - - -
  - - - - - - - -
  - - - - - - - -
  - - - - - - - -
User: black
Moves: 0
===
```

Here, '`-`' indicates an empty square, '`b`' indicates a black piece, and '`w`' indicates a white piece. The "User:" line tells which color the User (whether human or AI) has (as last set at the beginning or by the '`color`' command), and the "Moves:" line tells how many moves (including passes) have been made since the beginning of the game or last '`clear`' command. These include both moves made after and before playing starts. Don't use the two '`===`' markers anywhere else in your output. This command gives the autograder a way to determine the state of your game board at any point. It does not change any of the state of the program.

`load` *file* Reads the given *file* and in effect substitutes its contents for the `load` command itself.

You may add any additional commands you choose.

3 Output

This time, you can prompt however you want, and print out whatever user-friendly output you wish (other than debugging output or Java exception tracebacks, that is). For example, you will probably want to print the game board out after each move is complete (this is distinct from printing the board in the special format required by '`dump`'). When users enter erroneous input, you should print an error message, and the input should have no effect (and in particular, the user should be able to continue entering commands after an error).

After the move (or '`resign`' command) that ends play, the program should print a line saying either `"Black wins."`, `"White wins."`, or `"Draw."` as appropriate. Use exactly those phrases, alone on their lines. Don't use these phrases in any other situation. After this point, the game is over, but the board remains in its final state.

4 Communicating with a Remote Program

Java supplies a Remote Method Invocation (RMI) package that allows two separate programs (possibly on different machines) to communicate with each other by calling each other's methods. In effect, one program can have pointers (called *remote pointers*) to objects in the other program. We have developed our own packages that allow you to make use of this facility.

You can communicate with a remote job by means of a "mailbox" abstraction that we supply in the form of classes in the package `ucb.util.mailbox`. Take a look at the interface `Mailbox` in that package (in the on-line documentation). The idea is that a mailbox is simply a kind of queue. Its methods allow you to *deposit* messages into it, and to wait for and *receive* messages that have been deposited into it, in the order they were deposited. You can do this even if the mailbox is on another machine. The class `QueuedMailbox` is probably the only implementation of `Mailbox` you'll need.

To talk to a remote program, you will employ two mailboxes: one to send it messages and one to receive messages from it. Both mailboxes will reside in the host program (the one that issues the `host` command). The messages they send each other will be a subset of the commands: the host or the client may send `"resign"` or moves; the host alone may additionally send `"start"`, `"clear"`, or `"color"` commands the client alone may send an empty command (empty string) as its first message (to tell the host that it wants to join the game), and either may send an empty command as a closing acknowledgement after receiving the winning move from their Opponent. The format is less free: a single space between the operands of the '`color`' command, and no leading or trailing whitespace.

The tricky part is getting pointers to the mailboxes from one program to the other. For this purpose, we provide another useful type: `ucb.util.SimpleObjectRegistry`. A registry is like a Map, in that it allows one to associate names with values (object references). Suppose that the host player has entered '`host Foo`' and is running on machine M. The host program creates a `SimpleObjectRegistry` and stores two `Mailboxes` in it named `"Foo.IN"` and `"Foo.OUT"` using the `rebind` method. When the joining (client) program executes `join FOO@`M, it retrieves these `Mailboxes` using (for example)

```
(Mailbox<String>) SimpleObjectRegistry.findObject("Foo.IN","M")
```

The client program sends messages to the host by depositing them into the mailbox `Foo.IN`, and reads messages from the host out of `Foo.OUT`. The roles of the two mailboxes are reversed in the host program, of course.

Once the client program has fetched remote pointers to `Foo.IN` and `Foo.OUT`, it informs the host program that it is ready to begin a game by sending a message consisting of the empty string to `Foo.IN`. It then executes commands sent from the host, reading them from `Foo.OUT`, and executes them as if they came from the User. When it receives a '`start`' command, it continues to use `Foo.OUT` to receive the Opponent's moves and `Foo.IN` to send copies of the Player's moves to the host.

On receiving the initial empty string (on `Foo.IN`), the host replies (on `Foo.OUT`) with a '`color`' command as described above, and then continues to send copies of '`clear`' commands, set-up moves, and '`color`' commands (but reversing the color) from the User until the User

types a '`start`' command. It relays this '`start`' command and then continues to send copies of the Player's moves (legal ones only) and to receive (and execute) moves from the visitor for the Opponent. The host and visitor continue to send moves to each other until either the end of the game arrives or one of the two sends a '`resign`' message.

When the game ends, whoever sent the last move (or resigned) should then wait for the other side to send an empty message, which acknowledges receipt of the last move. In the unusual case where the set-up moves have already won the game, the "winning move" is the '`start`' command from the host.

Any other sequence of messages is illegal. Your program can recover when it detects an illegal message sequence by treating it as if the opponent resigned (and presumably printing an error message). This is an abnormal occurrence, so we really aren't fussy about the details of how you choose to recover.

Upon sending or receiving its last message for a game to one of these mailboxes, the host program (not the visitor) should apply its `.close` method to make sure that the message has been received and to shut down the mailbox. Also, the host should close its repository when it receives the first message from the visitor. Finally, the `quit` command should close any mailboxes, regardless of whether it is a host or a client, in order to signal to the other program that it is no longer present (otherwise, when a client quits, the host might just sit waiting forever to receive something on the client's outbox, which the host owns).

Annoying technical glitch. When the program that creates a remote object (a mailbox in this case) terminates, the object is destroyed, and attempts to call its methods (including `.close` get `RemoteExceptions`. If you get such an exception, it probably means that the other program has terminated. Be prepared to receive such exceptions (that is, don't simply surround everything with

```
try {
   ...
} catch (RemoteException e) { }
```

and effectively ignore the exception.) If you're in the middle of a game, interpret such an exception as a `"resign"` message. If you receive one of these at the end of the game (when waiting for an empty message after sending a winning move, for example), then ignore it.

5 Running Your Program

Your job is to write a program to play Reversi. Appropriately enough, we'll call the program `reversi`. Your AIs can play arbitrarily stupidly, as long as they make only legal moves. For extra credit, you can make them smart enough to reliably find a forced win five moves away (i.e., a situation in which you have won after making three more moves). There will be a tournament at the end. Also for extra credit, you can provide a GUI (after all, what computer board game is complete without a GUI?).

To run your program, the User types

```
java reversi [ --display ]
```

The optional parameter `--display` indicates that the User will communicate through a graphical interface (GUI). Your program must work without this option (that's how we test it). If you choose not to implement a GUI, make sure that your program terminates with an error exit code other than 0 when given the `--display` parameter. Otherwise, your program should exit with exit code 0. For this project, even if someone entered invalid commands during a session, your program should exit normally (and, of course, print error messages in response to the errors).

6 Your Task

The staff directory will contain skeleton files for this project in `proj3`.

Please read *General Guidelines for Programming Projects.*

Be sure to include tests of your program (that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them. To help with testing and debugging, we will provide our own version of the program, so that you can test your program against ours (we'll be on the lookout for illegal moves). More details will follow.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes an illegal move, tell him and just give him another chance. We don't care about the message format you use to do this.

Be sure to include documentation. This consists of a user's manual explaining how to use your program, and a brief internals document describing overall program structure, and any important data structures and algorithms you use (especially, how does your machine player choose its moves?).

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that

```
gmake check
```

runs your tests.

7 Advice

At first glance, it might seem that with all the options available, the program would have to be a mass of **if** statements covering all possible cases. But with proper use of abstraction, this does not have to be.

We've included documentation for the `ucb.util` and `ucb.util.mailbox` packages in the on-line materials. You will also want to read Chapter 11 of *A Java Reference* and Chapter 10 of *Data Structures (Into Java)* (on concurrency). If you need random numbers, take a look at `java.util.Random` and Chapter 11 of *Data Structures (Into Java).*

I suggest working first on the classes `Board`, which is supposed to represent the game board, and `Command`, which houses some command-parsing code.

Next, implement the human player. If you now get the remote game part working (it's not really as hard as it sounds) you can run two versions of your program and play against yourself, checking that the board works properly. Then you can tackle writing a machine player. Start with something really simple (perhaps choosing a legal move at random) and introduce strategy only when you get everything working properly.