UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2004

Project #3: Checkers

Due: 8 December 2004

1 Introduction

Checkers (or *draughts* outside the United States) is a pure strategy game between two players. For this last project, you are to implement a checkers-playing program, capable of handling games between a human and a machine player, two machine players, or a human or machine and a remote player (either human or machine). We're not going to be picky about how good your machine player is, as long as it plays by the rules. If you want it to make random legal moves, that's fine. However, you are invited to give it an improved strategy; we'll be providing supplementary notes on the subject.

2 Rules of Checkers

Checkers is played on what I used to call a checkerboard, but which now seems to be called a chess board. Various versions of the game use different sizes of board. We'll use 8×8. Pieces occupy only the dark squares and move along diagonals. The player who moves first gets twelve black pieces and the player who moves second gets twelve white pieces. Figure 1 shows the initial setup of pieces and the standard numbering scheme used to designate squares[1].

Simple moves. There are two kinds of pieces: initially, all are *single men,* as shown in the figure. When a single man advances to the last rank (squares 1–4 for white pieces, 29–32 for black), it becomes a *king* (we say it is *crowned*). Single pieces move one square diagonally, with single white pieces moving up and single black pieces moving down. Kings can move one square in any of the four diagonal directions. Black moves first.

[1] In fact, the pieces are officially red and white, respectively, for black and white. However, we'll use black instead of red, since the paper handout is black-and-white and since many textbooks use the term "black" for "red." Also, you will often see the board printed in books with the dark squares being light colored and the light squares dark colored, so as to show up the pieces more clearly. We won't do this.

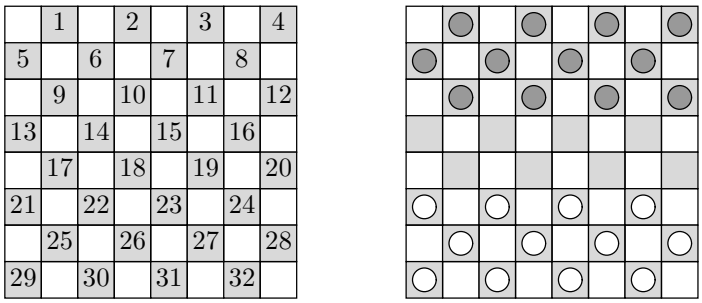

Figure 1: On the left: a checkerboard showing the standard numbering of squares. On the right: the initial configuration of pieces.

Captures. A piece may not move into an occupied square. However, if the square is occupied by a piece of the opposing color, and if there is an unoccupied square on the other side of the opposing piece, a piece may *jump* over the opposing piece, thus moving two diagonal squares. A jump also *captures* the opposing piece, removing it from the board. If, after making a jump, another jump is possible from the capturing piece's new square, then the piece may make that capture as well, and this process may continue as long as the piece has a capturing move. Pieces are crowned only at the end of a move, so a single piece that lands on the last rank cannot make any further capture on that move. Figure 2 illustrates legal moves and captures.

Forced moves. A player who has a legal capturing move *must* jump, and must continue to jump with that piece until no capture is possible. When more than one capture is possible, the player has a free choice of which to take (it is not necessary to take the capture that allows one to capture the most pieces).

End of the game. A player loses when he has no legal moves. This may either be because all his pieces are captured, or because his pieces are completely blocked, as in Figure 3. At any time before a loss by either side, the two players may agree to a draw (well, this isn't the real rule, but that is too complicated and vague for our purposes). Finally, a player can lose by resigning on his turn, or by making any kind of illegal move.

3 Notation

A transcript of a game is written in free format using the following syntax.

```
<Game>          ::= <End-Moves>
                  | <Move-Pair> <Game>
<End-Moves>     ::=   <Move-Number> <Move> <Score>
                  | <Score>
<Move-Pair>     ::=   <Move-Number> <Move> <Move>
<Move-Number>   ::= <Number>.
<Move>          ::=   <Normal-Move>
                  | <Special-Move>
<Normal-Move>   ::= <Number>-<Number>
                  | <Normal-Move>-<Number>
<Special-Move>  ::= resign
```

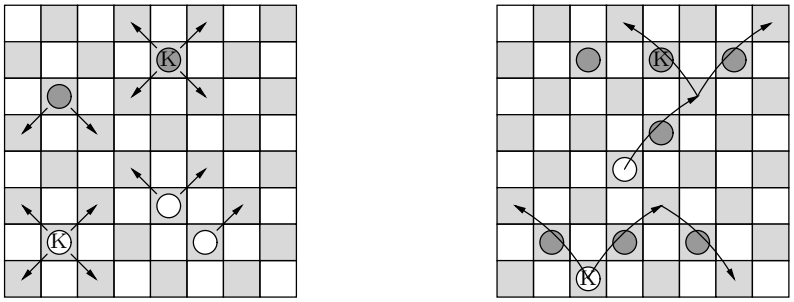

Figure 2: On the left: examples of simple moves. The white piece on 27 is blocked by another white piece from moving to 23. On the right: examples of capturing moves. The white piece on 18 may capture the black piece on 15 and either the black king on 7 (18-11-2) or the black single piece on 8 (18-11-4). It must capture two pieces. After either move, it will be on its last rank, and will therefore be crowned. However, it may not then capture the piece on 6 until its next move. The white king at 30, on the other hand, may reverse direction and capture both the pieces on 26 and 27 (30-23-32), or it may capture just the piece on 25 (30-21). Here, kings have 'K's on them; normally, you crown a piece by putting a second piece on top.

	| draw
	| accept
	| reject
<Score> ::=	0-1 | 1-0 | 1/2-1/2

Here, <Number> refers to a positive decimal integer. There may not be any space in a <Move-Number>, <Score>, or <Move>. The score marks the end of a game. It is either 1-0 (black wins), 0-1 (white wins), or 1/2-1/2 (draw). The <Move-Number>s are supposed to be consecutive, beginning at 1, except that a <Special-Move> does not increment the move count. Each <Move-Number> should start a new line in a transcript, as should the <Score>. Each move consists of a sequence of two or more square numbers (1–32), the first being the starting square for the piece that is moving, the last being its ultimate destination, with any middle ones being the intermediate squares in a multiple jump. The usual convention is that if there is only one legal way to reach the final square from the first in a multiple jump, then the intermediate steps need not be shown. However, we aren't going to use this abbreviation.

The <Special-Move> `resign` indicates that the player on move loses through resignation. It should always be followed by the final score in a transcript. A player who loses in the normal way (by not having a legal move) is not allowed to resign; the transcript at that point simply records the score.

The <Special-Move> `draw` offers a draw. The responding move must either be `accept,` which is immediately followed in a transcript by the score `1/2-1/2`, or `reject,` which is followed by a regular move (i.e., from the player who offered the draw). A `reject` may not be followed by another `draw`. The moves `reject` and `accept` may not appear in any other context.

4 The Problem

Your job is to write a program to play checkers. Your program must be called `checkers`. You'll start it with

`java checkers` *options*

where the available *options* are as follows:

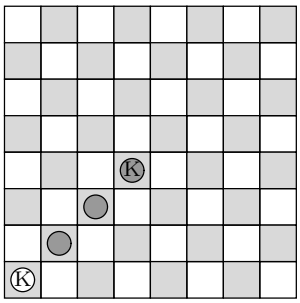

Figure 3: A lost position for white. White still has a piece, but it has no legal moves.

`--transcript=`*FILE* write a record of the game in the notation described in §3 to *FILE*.

`--display` Use a graphical interface for the board. (Default is text.)

`--player1=auto` The first player is a machine player. (Default is that the first player is the person sitting at the keyboard.) When the not using either `--display` or `--transcript`, this option also causes the transcript to be printed on the standard output.

`--player2=`*NAME* The second player is using a remote program. The first player's program (the one running locally) is hosting the game. This game can be referred to by other programs wishing to join it as *NAME* (see the next form of `--player2`). By default, the second player is the machine.

`--player2=`*NAME*`@`*HOST* The second player is using a remote program on the machine *HOST* and is hosting the game, which is designated *NAME*.

`--white` First player (at the terminal) takes the white pieces. This option is not allowed with the second form of the `--player2` option. In that case, the hosting player decides the color, and any `--white` option is ignored.

`--init=`*FILE* Read and make moves from *FILE* and then let the players continue the game from that point. The moves in *FILE* should obey the format used for transcripts (see §3).

`--level=`*NUM* If you provide any intelligence in your program, this parameter should limit the number of moves it looks ahead. Your program need only support `--level=0`, and can ignore other values (except that *NUM* must be a non-negative integer).

`--seed=`*NUM* If you use random numbers in your program, it is a good idea to provide for reproducible results. Do this by having all random-number generators in your program start with seeds that are derived from *NUM*. For example, if you have two machine players, you could have the first use a random number generator seeded with *NUM* and the second could start with *NUM*+42. See the `.setSeed`(N) method in `java.util.Random` for how to arrange this. Of course, they could both start with the same seed, but the game might not be as interesting. I do *not* recommend having two machine players in the same program share a random-number generator, by the way; depending on your implementation, this could lead to non-deterministic results.

What all this means is that

```
java checkers
```

lets you play against the machine using the terminal, and

```
java checkers --player1=auto
```

plays the machine against itself, printing the game on the terminal. To make this game reproducible, you might use

```
java checkers --player1=auto --seed=4242421
```

You can host a game against another program with something like

```
java checkers --player2=ourgame                  # You play, or
java checkers --player1=auto --player2=ourgame   # Your program plays
```

If you did this on, say, `nova.cs`, then the other player could join this game later by typing (on the other machine)

```
java checkers --player2=ourgame@nova.cs                # He plays, or
java checkers --player1=auto --player2=ourgame@nova.cs # His machine plays
```

You need not accept the GUI (`--display`) argument; it is an optional (extra credit) part of this project. We will actually do automatic testing only on commands like

```
   java checkers --player1=auto --init=... --seed=...
   java checkers --player1=auto --player2=X@Y --seed=...
or
   java checkers --player1=auto --player2=X --seed=...
```

(that is, we'll check that it plays legal games against our program and itself). The readers will check your GUI, if provided. If you do not provide a GUI, your program should print an error message when started with the `--display` option.

5 Format of Text Moves

When playing with a textual interface, players type only their own <Move>s, not the opponent's moves, move numbers, or scores. The terminal session on the left, below, shows the case of a human player playing black, and the one on the right shows the case of a human player playing white. The underlined portions in both cases are what the human types.

```
1. 11-15
1. ... 24-20
2. 8-11
2. ... 28-24
3. draw
3. ... reject
3. 9-13
3. ... 22-18
4. b
+-+-+-+-+-+-+-+-+
| |b| |b| |b| |b|
+-+-+-+-+-+-+-+-+
|b| |b| |b| | | |
+-+-+-+-+-+-+-+-+
| | | |b| |b| |b|
+-+-+-+-+-+-+-+-+
|b| | | |b| | | |
+-+-+-+-+-+-+-+-+
| | | |w| | | |w|
+-+-+-+-+-+-+-+-+
|w| | | |w| |w| |
+-+-+-+-+-+-+-+-+
| |w| |w| |w| | |
+-+-+-+-+-+-+-+-+
|w| |w| |w| |w| |
+-+-+-+-+-+-+-+-+
4. 15-22
4. ... 25-18
5. resign
0-1
```

```
1. 11-15 24-20
2. 8-11 28-24
3. draw reject
3. 9-13 22-18
4. 15-22 21-17
Error: illegal move.  Try again.
4. 15-22 b
+-+-+-+-+-+-+-+-+
| |b| |b| |b| |b|
+-+-+-+-+-+-+-+-+
|b| |b| |b| | | |
+-+-+-+-+-+-+-+-+
| | | |b| |b| |b|
+-+-+-+-+-+-+-+-+
|b| | | | | | | |
+-+-+-+-+-+-+-+-+
| | | | | | | |w|
+-+-+-+-+-+-+-+-+
|w| |b| |w| |w| |
+-+-+-+-+-+-+-+-+
| |w| |w| |w| | |
+-+-+-+-+-+-+-+-+
|w| |w| |w| |w| |
+-+-+-+-+-+-+-+-+
4. 15-22 25-18
5. resign
0-1
```

The special command 'b' (for "board") prints the current state of the board in the format shown. To denote kings, use capital letters ('B' and 'W'). As you can see from this example, your human-player text interface should not allow the player to make an illegal move (this is only friendly; if you were to transmit an illegal move to another program, it is supposed to treat the move as a resignation). The transcript for the game above would be

```
1. 11-15 24-20
2. 8-11 28-24
3. draw reject
3. 9-13 22-18
4. 15-22 25-18
5. resign
0-1
```

6 Communicating with a Remote Program

Java supplies a Remote Method Invocation (RMI) package that allows two separate program executions (possibly on different machines) to communicate with each other by calling each other's methods. 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 types in the package `ucb.util.mailbox`. Take a look at the interface `Mailbox` in that package. 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 is started with the option `--player2=`*NAME* with no `@...` part). Each message (except the first; see below) will be a string in the format <Move> or <Score>, as described in §3, with the <Score> coming last. That is, on receiving (or sending) the last move in a game, both programs send each other the score.

The tricky part is getting pointers to the mailboxes from one program to the other. For this purpose, we provide another useful type: `java.util.SimpleObjectRegistry`. A registry is simply a kind of dictionary in which one can associate names with values (object references). The program that is started

```
java checkers ... --player2=Foo
```

on machine M (the host) creates a `SimpleObjectRegistry` and stores two `Mailboxes` in it named `"Foo.IN"` and `"Foo.OUT"` using the `rebind` method. The program that is started

```
java checkers ... --player2=Foo@M
```

retrieves these `Mailboxes` using (for example)

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

This second program uses the mailbox `Foo.IN` to send messages to M and `Foo.OUT` to get responses (the roles of the two mailboxes are reversed in the program that created the repository, of course).

The `Foo.OUT` mailbox should start with the sequence of moves from the host's `--init` file, if any (just the moves; no line numbers). Next should follow either the message string `"white"` (meaning "play the white pieces and go second") or `"black"`. From then on, it's just a matter of reading moves from the .OUT mailbox and sending responses to the .IN mailbox.

Upon sending your last message to one of these mailboxes (which should normally be the score), apply its `.close` method to make sure that the message has been received and to shut down the mailbox.

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 get `RemoteExceptions`. If you get such an exception, it probably means that the other program has terminated (and you should too, probably). 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 receive one of these at the end of the game, then you can ignore it; at other times, treat it as if the opponent has resigned.

7 Advice

If you "don't know how to begin," start with reading and checking the command line (see `ucb.util.CommandArgs`). Then try writing methods that prompt for and read moves from the terminal, that check moves for legality, that generate possible legal moves starting from a given position, that select legal moves to make, that print moves in transcript form, and that make moves on a model of the checkerboard (you definitely want a class that represents the current state of a game—where all the pieces are and whose move it is). Don't even think about a GUI or about clever strategy until you have the basic stuff (required by the assignment) working.

Your final work must be your own, but especially on this project, feel free to get together with other students to discuss ideas and plan strategies. Of course, you should always feel free to consult your TA or me.

8 What to Turn In

As usual, be sure to provide a makefile, user manual, and INTERNALS document. Also, `gmake test` should do some kind of testing of your program, at least playing against itself. We don't expect tests of the GUI, if you provide one.

9 Optional: The GUI

We encourage you to try the extra-credit GUI (Graphical User Interface). To make life easier for you, we will provide a little widget: `ucb.games.checkers.CheckerboardUI`, plus a simple test GUI that uses it (`CheckerTest`). The idea is that the `CheckerboardUI` "listens" to a model of the board and reflects all changes it finds there. It also allows the rest of your program to "listen to" it and to respond to input events (moves) that are made on the board it displays. You will find that there isn't really a lot more you need to do, except to figure out what menus or buttons to include. Remember always: RTFM.