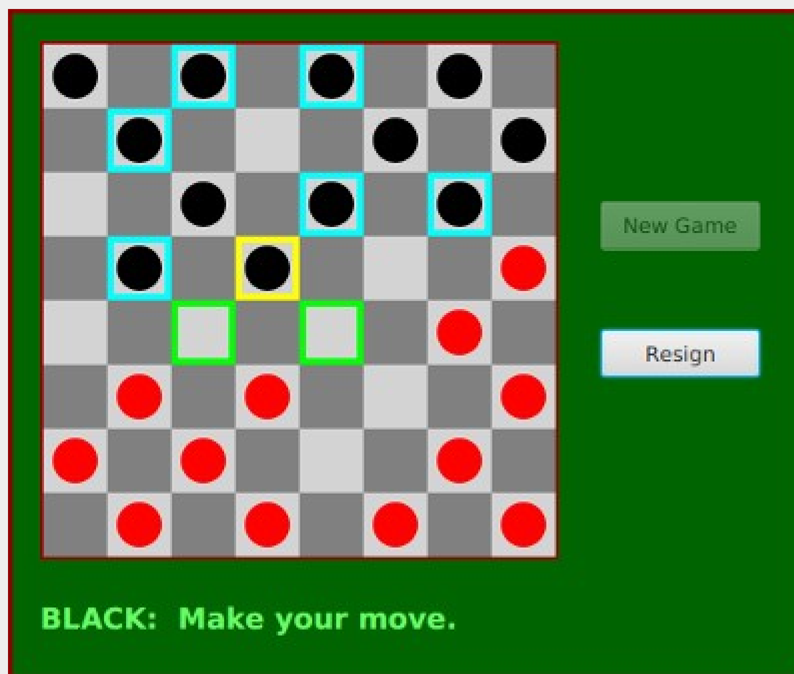
7.5.3 Checkers

As a final example for this chapter, we'll look at a more substantial example of using a 2D array. This is the longest program that we have encountered so far, with 745 lines of code. The program lets two users play checkers against each other. The checkers game is played on an eight-by-eight board, which is based on an example from Subsection 6.5.1. The players are called "red" and "black," after the color of their checkers. I'm not going to explain the rules of checkers here; possibly you can learn them by trying out the program.

In the program, a player moves by clicking on the piece that they want to move, and then clicking on the empty square to which it is to be moved. As an aid to the players, any square that the current player can legally click at a given time is highlighted with a brightly colored border. The square containing a piece that has been selected to be moved, if any, is surrounded by a yellow border. Other pieces that can legally be moved are surrounded by a cyan-colored border. If a piece has already been selected to be moved, each empty square that it can legally move to is highlighted with a green border. The game enforces the rule that if the current player can jump one of the opponent's pieces, then the player must jump. When a player's piece becomes a king, by reaching the opposite end of the board, a big white "K" is drawn on the piece. Here is a picture of the program early in a game. It is black's turn to move. Black has selected the piece in the yellow-outlined square to be moved. Black can click one of the squares outlined in green to complete the move, or can click one of the squares outlined in cyan to select a different piece to be moved.



I will only cover a part of the programming for this example. I encourage you to read the complete source code, *Checkers.java*. It's long and complex, but with some study, you should understand all the techniques that it uses. The program is a good example of state-based, event-driven, object-oriented programming.

The data about the pieces on the board are stored in a two-dimensional array. Because of the complexity of the program, I wanted to divide it into several classes. In addition to the main class, there are several nested classes. One of these classes is *CheckersData*, which handles the data for the board. It is not directly responsible for any part of the graphics or event-handling, but it provides methods that can be called by other classes that handle graphics and events. It is mainly this class that I want to talk about.

The *CheckersData* class has an instance variable named `board` of type `int[][]`. The value of board is set to "`new int[8][8]`", an 8-by-8 grid of integers. The values stored in the grid are defined as constants representing the possible contents of a square on a checkerboard:

```
static final int
          EMPTY = 0,            // Value representing an empty square.
          RED = 1,              // A regular red piece.
          RED_KING = 2,         // A red king.
          BLACK = 3,            // A regular black piece.
          BLACK_KING = 4;       // A black king.
```

The constants `RED` and `BLACK` are also used in my program (or, perhaps, misused) to represent the two players in the game. When a game is started, the values in the array are set to represent the initial state of the board. The grid of values looks like

	0	1	2	3	4	5	6	7
0	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
1	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK
2	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
3	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
4	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
5	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED
6	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY
7	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED

A regular black piece can only move "down" the grid. That is, the row number of the square it moves to must be greater than the row number of the square it comes from. A regular red piece can only move up the grid. Kings of either color can move in both directions.

One function of the *CheckersData* class is to take care of changes to the data structures that need to be made when one of the users moves a checker. An instance method named `makeMove()` is provided to do this. When a player moves a piece from one square to another, the values of two elements in the array are changed. But that's not all. If the move is a jump, then the piece that was jumped is removed from the board. (The method checks whether the move is a jump by checking if the square to which the piece is moving is two rows away from the square where it starts.) Furthermore, a `RED` piece that moves to row 0 or a `BLACK` piece that moves to row 7 becomes a king. Putting all that into a subroutine is good programming: the rest of the program doesn't have to worry about any of these details. It just calls this `makeMove()` method:

```
/**
 * Make the move from (fromRow,fromCol) to (toRow,toCol).  It is
 * ASSUMED that this move is legal!  If the move is a jump, the
 * jumped piece is removed from the board.  If a piece moves
 * to the last row on the opponent's side of the board, the
 * piece becomes a king.
 */
void makeMove(int fromRow, int fromCol, int toRow, int toCol) {

   board[toRow][toCol] = board[fromRow][fromCol]; // Move the piece.
   board[fromRow][fromCol] = EMPTY; // The square it moved from is now
empty.

   if (fromRow - toRow == 2 || fromRow - toRow == -2) {
         // The move is a jump.  Remove the jumped piece from the
board.
      int jumpRow = (fromRow + toRow) / 2; // Row of the jumped piece.
      int jumpCol = (fromCol + toCol) / 2; // Column of the jumped
piece.
      board[jumpRow][jumpCol] = EMPTY;
   }

   if (toRow == 0 && board[toRow][toCol] == RED)
      board[toRow][toCol] = RED_KING;  // Red piece becomes a king.
   if (toRow == 7 && board[toRow][toCol] == BLACK)
      board[toRow][toCol] = BLACK_KING;  // Black piece becomes a
king.

}  // end makeMove()
```

An even more important function of the *CheckersData* class is to find legal moves on the board. In my program, a move in a Checkers game is represented by an object belonging to the following class:

```
/**
 * A CheckersMove object represents a move in the game of
 * Checkers.  It holds the row and column of the piece that is
 * to be moved and the row and column of the square to which
 * it is to be moved.  (This class makes no guarantee that
 * the move is legal.)
 */
private static class CheckersMove {

   int fromRow, fromCol;  // Position of piece to be moved.
   int toRow, toCol;      // Square it is to move to.

   CheckersMove(int r1, int c1, int r2, int c2) {
        // Constructor.  Set the values of the instance variables.
      fromRow = r1;
      fromCol = c1;
      toRow = r2;
      toCol = c2;
   }

   boolean isJump() {
        // Test whether this move is a jump.  It is assumed that
        // the move is legal.  In a jump, the piece moves two
        // rows.  (In a regular move, it only moves one row.)
      return (fromRow - toRow == 2 || fromRow - toRow == -2);
   }

}  // end class CheckersMove.
```

The *CheckersData* class has an instance method which finds all the legal moves that are currently available for a specified player. This method is a function that returns an array of type `CheckersMove[]`. The array contains all the legal moves, represented as *CheckersMove* objects. The specification for this method reads

```
/**
 * Return an array containing all the legal CheckersMoves
 * for the specified player on the current board.  If the player
 * has no legal moves, null is returned.  The value of player
 * should be one of the constants RED or BLACK; if not, null
 * is returned.  If the returned value is non-null, it consists
 * entirely of jump moves or entirely of regular moves, since
 * if the player can jump, only jumps are legal moves.
 */
CheckersMove[] getLegalMoves(int player)
```

A brief pseudocode algorithm for the method is

```
Start with an empty list of moves
```

```
Find any legal jumps and add them to the list
if there are no jumps:
   Find any other legal moves and add them to the list
if the list is empty:
   return null
else:
   return the list
```

Now, what is this "list"? We have to return the legal moves in an array. But since an array has a fixed size, we can't create the array until we know how many moves there are, and we don't know that until near the end of the method, after we've already made the list! A neat solution is to use an *ArrayList* instead of an array to hold the moves as we find them. In fact, I use an object defined by the parameterized type `ArrayList<CheckersMove>` so that the list is restricted to holding objects of type *CheckersMove*. As we add moves to the list, it will grow just as large as necessary. At the end of the method, we can create the array that we really want and copy the data into it:

```
Let "moves" be an empty ArrayList<CheckersMove>
Find any legal jumps and add them to moves
if moves.size() is 0:  // There are no legal jumps!
   Find any other legal moves and add them to moves
if moves.size() is 0:  // There are no legal moves at all!
   return null
else:
   Let moveArray be an array of CheckersMoves of length moves.size()
   Copy the contents of moves into moveArray
   return moveArray
```

Now, how do we find the legal jumps or the legal moves? The information we need is in the `board` array, but it takes some work to extract it. We have to look through all the positions in the array and find the pieces that belong to the current player. For each piece, we have to check each square that it could conceivably move to, and check whether that would be a legal move. If we are looking for legal jumps, we want to look at squares that are two rows and two columns away from the piece. There are four squares to consider. Thus, the line in the algorithm that says "Find any legal jumps and add them to moves" expands to:

```
For each row of the board:
   For each column of the board:
      if one of the player's pieces is at this location:
         if it is legal to jump to row + 2, column + 2
             add this move to moves
         if it is legal to jump to row - 2, column + 2
             add this move to moves
         if it is legal to jump to row + 2, column - 2
             add this move to moves
         if it is legal to jump to row - 2, column - 2
             add this move to moves
```

The line that says "Find any other legal moves and add them to moves" expands to something similar, except that we have to look at the four squares that are one column and one row away from the piece. Testing whether a player can legally move from one given square to another given square is itself non-trivial. The square the player is moving to must actually be on the board, and it must be empty. Furthermore, regular red and black pieces can only move in one direction. I wrote the following utility method to check whether a player can make a given non-jump move:

```
/**
 * This is called by the getLegalMoves() method to determine
 * whether the player can legally move from (r1,c1) to (r2,c2).
 * It is ASSUMED that (r1,c1) contains one of the player's
 * pieces and that (r2,c2) is a neighboring square.
 */
private boolean canMove(int player, int r1, int c1, int r2, int c2) {

   if (r2 < 0 || r2 >= 8 || c2 < 0 || c2 >= 8)
      return false;  // (r2,c2) is off the board.

   if (board[r2][c2] != EMPTY)
      return false;  // (r2,c2) already contains a piece.

   if (player == RED) {
      if (board[r1][c1] == RED && r2 > r1)
          return false;  // Regular red piece can only move down.
       return true;  // The move is legal.
   }
   else {
      if (board[r1][c1] == BLACK && r2 < r1)
          return false;  // Regular black piece can only move up.
       return true;  // The move is legal.
   }

}  // end canMove()
```

This method is called by my `getLegalMoves()` method to check whether one of the possible moves that it has found is actually legal. I have a similar method that is called to check whether a jump is legal. In this case, I pass to the method the square containing the player's piece, the square that the player might move to, and the square between those two, which the player would be jumping over. The square that is being jumped must contain one of the opponent's pieces. This method has the specification:

```
/**
 * This is called by other methods to check whether
 * the player can legally jump from (r1,c1) to (r3,c3).
 * It is assumed that the player has a piece at (r1,c1), that
 * (r3,c3) is a position that is 2 rows and 2 columns distant
 * from (r1,c1) and that (r2,c2) is the square between (r1,c1)
 * and (r3,c3).
 */
private boolean canJump(int player, int r1, int c1,
```

```
                                    int r2, int c2, int r3, int c3) { .
. .
```

Given all this, you should be in a position to understand the complete `getLegalMoves()` method. It's a nice way to finish off this chapter, since it combines several topics that we've looked at: one-dimensional arrays, *ArrayLists*, and two-dimensional arrays:

```
CheckersMove[] getLegalMoves(int player) {

   if (player != RED && player != BLACK)
      return null;  // (This will not happen in a correct program.)

   int playerKing;  // The constant for a King belonging to the
player.
   if (player == RED)
      playerKing = RED_KING;
   else
      playerKing = BLACK_KING;

   ArrayList<CheckersMove> moves = new ArrayList<CheckersMove>();
               // Moves will be stored in this list.

   /*  First, check for any possible jumps.  Look at each square on
       the board.  If that square contains one of the player's pieces,
       look at a possible jump in each of the four directions from
that
       square.  If there is a legal jump in that direction, put it in
       the moves ArrayList.
   */

   for (int row = 0; row < 8; row++) {
      for (int col = 0; col < 8; col++) {
        if (board[row][col] == player || board[row][col] ==
playerKing) {
            if (canJump(player, row, col, row+1, col+1, row+2, col+2))
               moves.add(new CheckersMove(row, col, row+2, col+2));
            if (canJump(player, row, col, row-1, col+1, row-2, col+2))
               moves.add(new CheckersMove(row, col, row-2, col+2));
            if (canJump(player, row, col, row+1, col-1, row+2, col-2))
               moves.add(new CheckersMove(row, col, row+2, col-2));
            if (canJump(player, row, col, row-1, col-1, row-2, col-2))
               moves.add(new CheckersMove(row, col, row-2, col-2));
        }
      }
   }

   /*  If any jump moves were found, then the user must jump, so we
       don't add any regular moves.  However, if no jumps were found,
       check for any legal regular moves.  Look at each square on
       the board.  If that square contains one of the player's pieces,
       look at a possible move in each of the four directions from
       that square.  If there is a legal move in that direction,
       put it in the moves ArrayList.
   */
```

```
   if (moves.size() == 0) {
      for (int row = 0; row < 8; row++) {
         for (int col = 0; col < 8; col++) {
           if (board[row][col] == player || board[row][col] ==
playerKing) {
              if (canMove(player,row,col,row+1,col+1))
                 moves.add(new CheckersMove(row,col,row+1,col+1));
              if (canMove(player,row,col,row-1,col+1))
                 moves.add(new CheckersMove(row,col,row-1,col+1));
              if (canMove(player,row,col,row+1,col-1))
                 moves.add(new CheckersMove(row,col,row+1,col-1));
              if (canMove(player,row,col,row-1,col-1))
                 moves.add(new CheckersMove(row,col,row-1,col-1));
           }
         }
      }
   }

   /* If no legal moves have been found, return null.  Otherwise,
create
      an array just big enough to hold all the legal moves, copy the
      legal moves from the ArrayList into the array, and return the
array.
   */

   if (moves.size() == 0)
      return null;
   else {
      CheckersMove[] moveArray = new CheckersMove[moves.size()];
      for (int i = 0; i < moves.size(); i++)
         moveArray[i] = moves.get(i);
      return moveArray;
   }

}  // end getLegalMoves
```