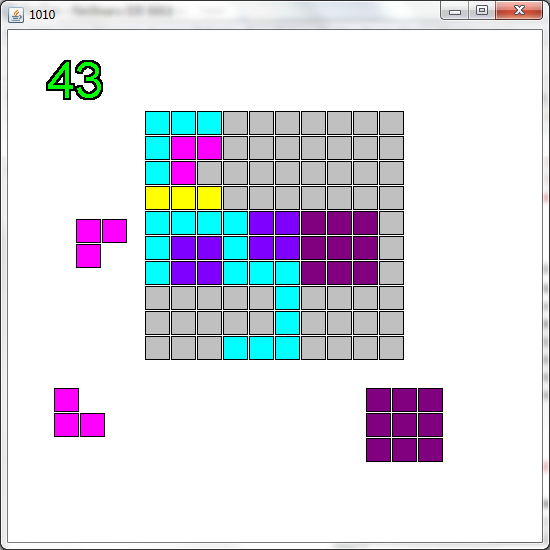
TenTen Lab

In this lab, you will complete an implementation of the game 1010!. The game 1010! is played on a 10x10 board (and, presumably, it’s pretty exciting, hence the name) which is initially empty. The player is given three pieces (polyonimoes like in Tetris) to place on the board. The player places each piece in turn (he may choose in which order to place the pieces) without rotating. The placed piece must be completely on the board and must not overlap with any occupied square. If placing a piece completes one or more rows or columns, the squares in those rows and columns are cleared. Once the player has placed all three pieces, three new pieces are selected and play continues. If the player is unable to place any of his pieces, then the game is over.



A game of 1010! in progress. The player is currently trying to play a pink L-shaped piece. Two other pieces – another pink L and a purple square – are available to play.

In our implementation, the board will be represented using a 2-dimensional array of Integers (10x10). We’re using the type Integer rather than int because we want the empty squares to be null (ints can’t be null). Integers are used just like ints (other than possibly taking on the value null). A non-null entry in the board will be a number between 0 and 8 inclusive indicating what color that square should be (0 = red, 1 = orange, 2 = yellow etc). Each piece is also represented using a 2-dimensional array of Integers with the same color-coding scheme. Each piece is a different size. These arrays can be found in the TenTenGame and TenTenPiece classes/objects:

public class TenTenGame {

public Integer [][] getBoard() { . . . }

}

public class TenTenPiece {

public Integer [][] getGrid() { . . . }

}

Exercise 1: Creating the Piece Grids

There are three different types of pieces in 1010!: line pieces, square pieces, and corner pieces. Line pieces are a single row or column of length 1 – 5. For example:









Note that a line piece can be oriented either vertically (like the blue 5x1 above) or horizontally (like the yellow 1x3 and orange 1x2 above). Also note that the red 1x1 piece is also a line piece.

Class TenTenPiece contains a function called createLinePiece which consumes an int, a boolean and another int. The first int indicates the (length) of the piece. The boolean indicates whether the piece is horizontal (true) or vertical (false). The third int (0 – 8) indicates what color to use to fill the grid. createLinePiece returns a 2D array of Integers. For example, to create the blue piece above, we would call createLinePiece( 5, false, 4 ) and expect to get back a 5x1 array of Integers with each entry set to 4. Your job is to write the code for createLinePiece so that it creates the array.

Square pieces are, well, square. They come in two sizes – 3x3 and 2x2 (1x1 pieces are, of course squares as well, but we handle them as line pieces rather than squares). Here are two examples:





The createSquarePiece consumes two ints. The first indicates the size (either 2 or 3) and the second indicates the color (0-8). Just like createLinePiece, the return value is a 2D array of Integers with each entry set to the fill value. Your second job is to write the code for this function.

Finally, corner pieces are either 2x2 or 3x3, but, unlike squares, only have one row and one column filled in. They can come in several different configurations based on which row (either the top or the bottom) and which column (either the left or the right column) is filled in. Here are some examples:



Function createCornerPiece consumes an int indicating the size (2x2 or 3x3), two booleans indicating whether to use the top (true) or bottom (false) rows and the left (true) or right (false) columns, and a second int indicating the fill. It returns a 2D array of the appropriate size with only the indicated row and column filled in. All other entries should be null. Your third job is to write the code for this function.

Once you have completed these functions, you can test them using the testPieceCreation function. You should see printed out, in order, a 5x1 piece with 4’s, a 1x4 piece with 3’s, a 1x1 piece with 0’s, a 2x2 square with 5’s, a 3x3 square with 6’s, 4 2x2 corner pieces of 7’s, the first with its corner in the top-left, the second with its corner in the top-right, then bottom-left and finally bottom-right, and then finally 4 3x3 corner pieces of 8’s, again with corners in the top-left, top-right etc. For the corner pieces, you should see .’s where the empty (null) entries are. So for example, the first 2x2 corner piece should display:

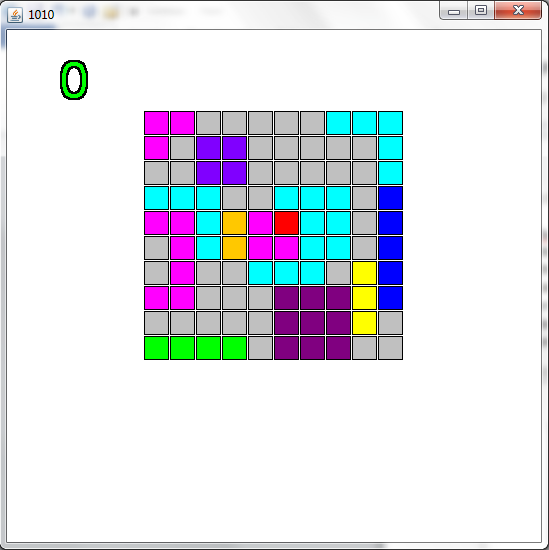
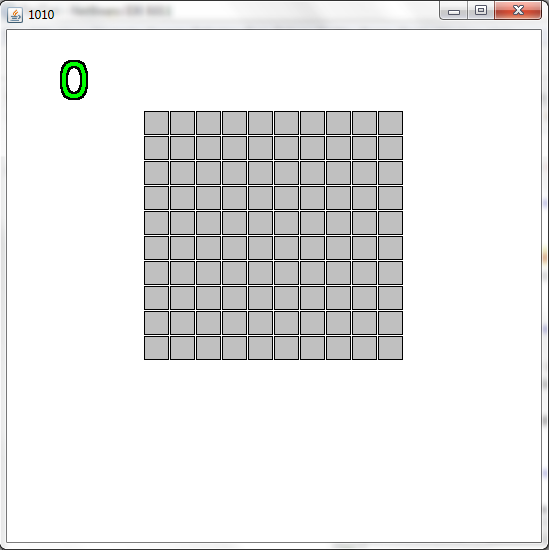
77

7.

Exercise 3: Creating and Displaying the Board

Class TenTenGame contains a method called TenTenGame. Note that the name of the function is the same as the name of the class… this is a “constructor”. It’s job is to initialize the game. There are several steps (enumerated in the comments). Right now, do steps 1-4.

Class TenTenDisplay contains a method called paintBoard. This function consumes a 2 dimensional array of Integers. Some entries may be null. Others are filled with a number between 0 and 8. In addition, the function consumes a Graphics class, and 6 ints. The first two ints indicate the coordinates of the top left corner of the board. The next two indicate the width and the height of the individual squares on the board. The last two ints indicate the horizontal and vertical spacing between squares. The function should paint squares on the screen. More detailed instructions are in the function. When you are done with this function, to test, run the launchGameFunction. You should see a blank board:

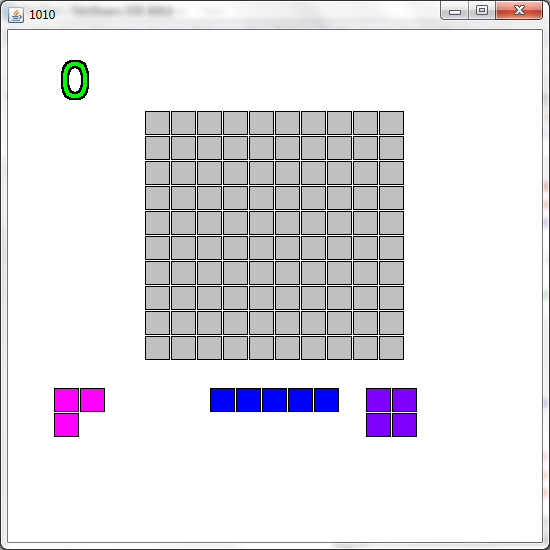


Press ‘l’ to load a board which has some actual squares filled in. You should see the 2nd image above.

Exercise 4: Creating and Displaying the Pieces

Go back to TenTenGame.TenTenGame and complete steps 5 and 6. Then, complete function getThreeNewPieces. This function should set each of the entries in the piece array to a new TenTenPiece.

Class TenTenDisplay contains a function called paintPiece. This function is very similar to paintBoard, but instead of displaying empty (null) squares as gray, it doesn’t display them at all. Once you have completed this function test by running launchGame() again. Now you should see a blank board with three pieces below the board. (Your pieces may be different than in the picture below since the pieces are randomly selected).

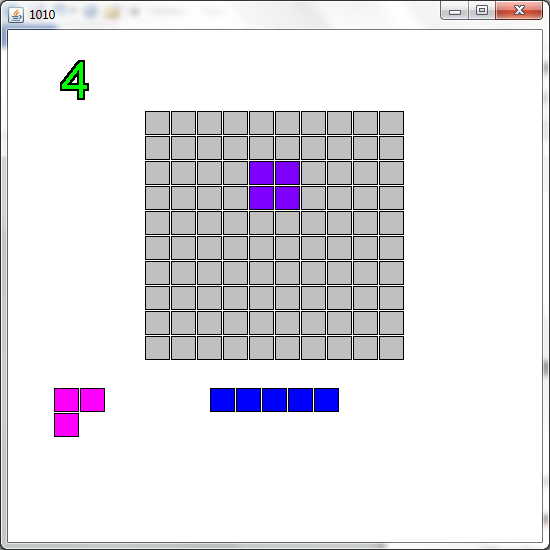


Click on one of the pieces and then move your mouse around. Make sure that the piece follows your mouse cursor. Click off the board to release the piece back to its starting place.

Exercise 5: Placing a Piece on the Board Part 1.

TenTenGame contains 2 functions called canPlace and playPiece.

When the user selects a piece and then clicks on the board, we have to determine if the piece can be placed in the location that the user selected. If it can, then we should fill in certain squares on the board corresponding to where the piece was placed. The functions canPlace and playPiece handle these tasks. Each consumes an int indicating the index (0 – 3) of the piece to be placed, and two more ints indicating the row and column coordinates of the square on the board on which the the top left corner of the piece should be placed. For example, in the game above, if the user selected the 2x2 square on the right (piece index 2), and then clicked on the square in the 3rd row from the top (row 2), 5th column from the left (column 4), then the board will end up looking like this:



Note that the top left corner of the piece is in the 3rd row, 5th column of the board. The other squares which make up the piece have been placed in adjacent rows and columns as you would expect. The code executed to make this happen is:

if ( game.canPlay( 2, 2, 5 ) )

{

game.playPiece( 2, 2, 5 );

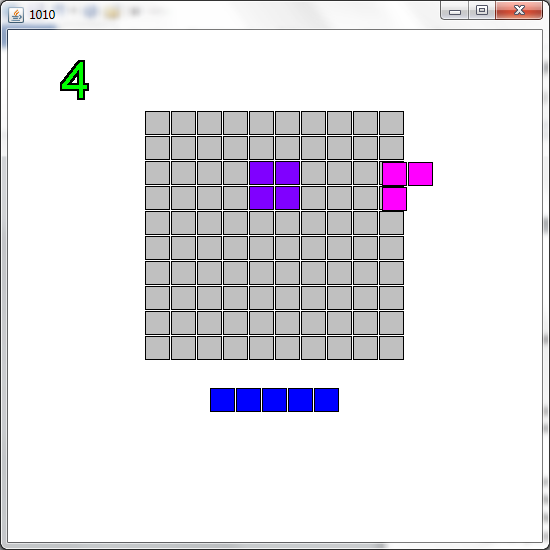
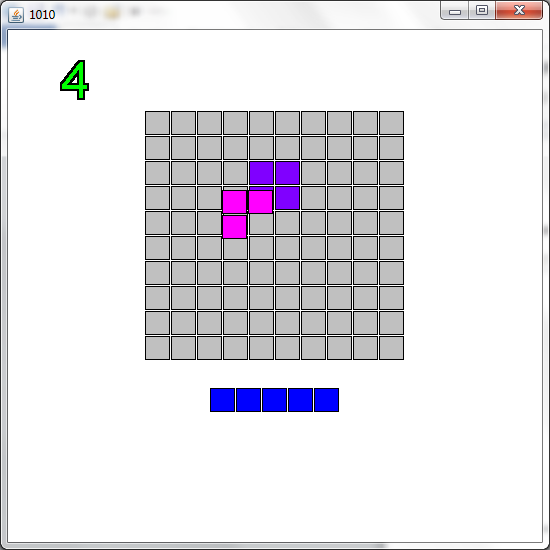
}

canPlay should consume the piece index and location and return true if the piece can be placed as required. That is:

1. Every part of the piece would be on the board (ie the row and column of the part will both be between 0 and 9 inclusive)
2. Every non-null entry of piece will fall on a null entry on board. It is OK if a null entry in the piece falls on a non-null entry of the board.

If either of these conditions is violated, canPlay should return false.

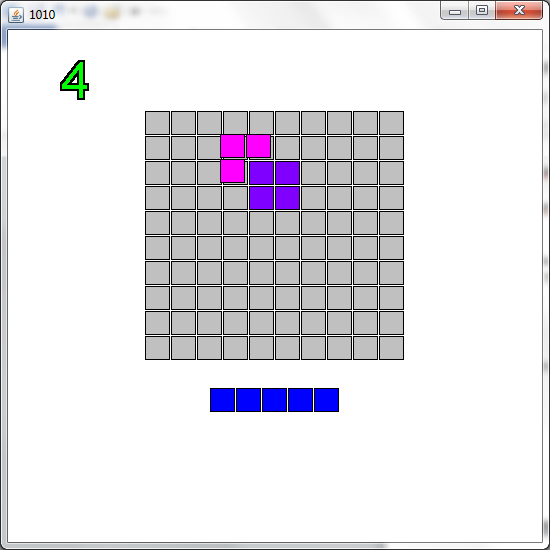
Here are some ILLEGAL moves:

In the first piecture, the piece is not completely on the board. In the second, the pink piece is overlapping with the purple piece.

playPiece is similar, but should just copy the piece’s grid to the board array. (Don’t forget that you can get the piece grid by calling getGrid). There are some other minor tasks to do (these are detailed in the comments for the function)

To test, run launchGame and place several pieces in the board. Be sure to test out placing pieces on top of each other (the game should NOT allow this) as well as placing pieces so that they extend beyond the edges of the board. You should also be sure to place several corner pieces so that the empty part of the corner piece is on top of an already placed piece (this SHOULD be allowed). For example:



Placing the pink piece here should be legal since none of its non-null parts overlap with an already placed piece.

Exercise 6: Resetting Pieces and Clearing Rows and Columns

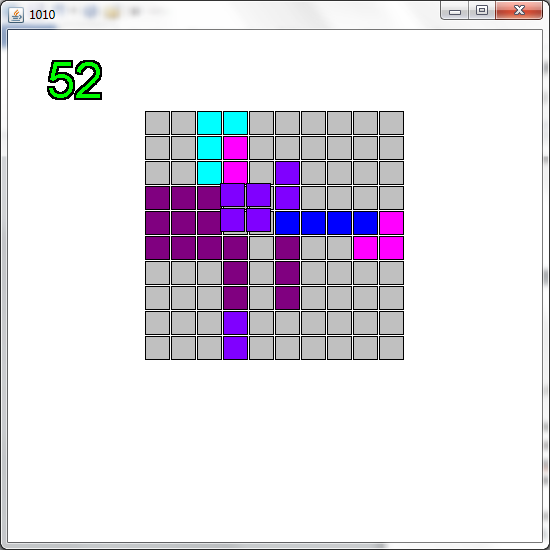
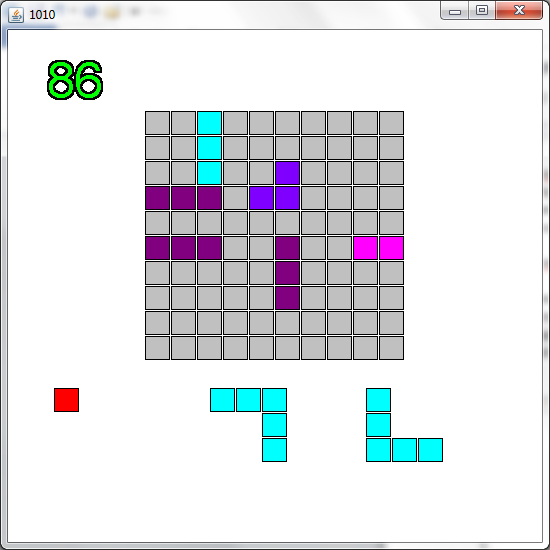
After all three pieces have been played, we need to get more pieces. Function allPiecesPlayed detects if all the pieces have been played. Recall that after a piece is played, its entry in the pieces array is set to null. allPiecesPlayed detects if all the entries in the pieces array are null. If they are, return true, if not return false. To test, place 3 pieces and see if more pieces come up.

Once several pieces have been played, we need to detect if columns and rows are complete and need to be cleared. There are several functions for this:

Functions rowComplete and colComplete detect if a given row or column is complete. That is, whether every entry in the row or column is non-null. They consume an int which indicates which row or column to check and produce a boolean – true if every entry in that row or column is non-null, false if any entry in the row or column is null.

Functions removeRow and removeCol set the entries in a given row or column to be null.

Finally, function clearFinishedRowsAndColumns uses the above functions to detect and remove complete rows and columns. This is not quite as simple as looking at each row and column and clearing it if is complete. A problem arises if a row and column are simultaneously completed since removing the completed row before detecting that the column is complete results in only the row being removed. For example:

Placing the 2x2 square as shown will complete both the 4th column and the 5th row. If we detect that the 5th row is complete and clear it, the 4th column will no longer be complete, so the rest of the squares in that column won’t be cleared. To solve this problem, we need to check all the rows and all the columns for completeness BEFORE we clear any of the rows or columns. Only after all the rows and columns have been checked can we start clearing. Of course, this means storing a boolean for each row and each column. Since there are 10 of each, we should use two 1-dimensional arrays. Each entry in these arrays will indicate whether the corresponding row (or column) should be cleared. Later, we’ll traverse these arrays of booleans and do the actual clearing. More details are in the comments for the function.

Once you have finished these functions, play the game and see that rows and columns are cleared. Try to create a situation in which multiple rows will be cleared at once. Create a situation in which multiple columns will be cleared. Then, try to create a situation in which both a row and a column will be cleared.

Exercise 7: Detecting Game Over

The game is over when none of the remaining (non-null) pieces can be placed anywhere on the board. Function detectGameOver detects this situation. The function should look at each piece in the pieces array. If the piece is non-null, it should check if the piece can be played (hint: there’s a function that will help with that) at any location on the board (that is, see if it can be played at 0,0 or 0,1, or 0,2 etc). If it can, then the game is not over (return false). If none of the pieces can be played at any location, then the game is over (return true).

To test this function, run the game and then press ‘e’. This should load a board position that is almost blocked. Place any pieces that you can being careful not to clear any rows or columns. Eventually, you should see the words “Game Over” in red when you can’t place any more pieces.

Exercise 8: Score

The score is stored in variable score. Up until now, the score has been 0. The score should increase in two situations:

1. A piece is placed. The score increases by the number of parts to the piece. For example, placing a 5x1 line piece is worth 5 points. Placing a 3x3 corner is also worth 5 points. Placing a 3x3 square is worth 9.
2. Rows or columns are cleared. Any time rows and columns are cleared, you score points according to how many rows and columns were cleared at once. If you clear 1 row/column you get 10 points. Each additional row/column is worth an extra 10 points. So 2 rows/columns at one time is worth 30 (10 + 20), 3 are worth 60 (10 + 20 + 30) etc. The maximum number of lines that you can clear at once is 6 (think about why). This is worth 150 points.

To implement the scoring scheme, you need to modify function playPiece to increment score by 1 for every non-null part placed in the board. Then, in clearFinishedRowsAndColumns, you need to count how many rows and columns were cleared and increase score by the appropriate amount (hint: there’s a quadratic function which will compute the appropriate number of points in terms of how many lines were cleared).