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

CS61B **P. N. Hilfinger**
Fall 2014

Project #0: 2048 (revision 2)

Due: Friday, 26 September 2014 at midnight

1 Introduction

This initial programming assignment is intended as an extended finger exercise, a mini-project rather than a full-scale programming project. The intent is to give you a chance to get familiar with Java and the various tools used in the course.

We will be grading *solely* on whether you manage to get your program to work (according to our tests) and to hand in the assigned pieces. There is a slight stylistic component: the submission and grading machinery require that your program pass a mechanized style check (`style61b`), which mainly checks for formatting and the presence of comments in the proper places. See `http://inst.eecs.berkeley.edu/~cs61b/fa12/labs/style61b.txt` for a description of the style it enforces and how to run it yourself.

To obtain the skeleton files (and set up an initial entry for your project in the repository), you can use the command

```
$ hw init proj0
```

which creates a working directory `proj0`. You will also find these files in `~cs61b/code/proj0`.

2 The Game

You've probably seen and perhaps played the game "2048," a single-player computer game written by Gabriele Cirulli, and based on an earlier game "1024" by Veewo Studio (see `http://gabrielecirulli.github.io/2048`). In this mini-project, you are to reproduce this game as a Java application. We have provided an API (Application Programmer's

Interface) for the actual mechanics of displaying the game board. You must use this API, since we will be using an instrumented implementation of it to test your project.

The game itself is quite simple. It's played on a 4×4 grid of squares, each of which can either be empty or contain a tile bearing an integer—a power of 2 greater than or equal to 2. Before the first move, the machine adds a tile containing either 2 or 4 to a random square on the initially empty board. The choice of 2 or 4 is random, with a 3:1 bias in favor of 2 (that is, there is a 75% chance of choosing 2 and a 25% chance of choosing 4).

On each move, the machine first adds a new tile containing either 2 or 4 to an empty square as for the initial configuration. The player then chooses a direction: north, south, east, or west. All tiles slide in that direction until there is no empty space left in the direction of motion (there needn't be any to start with.) If at this point there are two tiles bearing the same number that are now adjacent in the direction of motion, they merge into a single tile containing the sum of their values (that is, double the value of either one of them, and therefore still a power of two). The tiles then continue to slide in the direction of motion to eliminate the resulting empty space. Even if the merging and subsequent slide bring more tiles together with the same number, there is no further merging. When three adjacent tiles in the direction of motion have the same number, then the leading two tiles in the direction of motion merge, and the trailing tile does not. When there are adjacent four tiles with the same number in the direction of motion, they form two merged tiles.

For example, the board shown in Figure 1a, if tilted to the east, results in the board in Figure 1b. The tile marked with an asterisk in Figure 1b indicates a new, randomly chosen piece that the program then generates for the next turn.

Tilting does not cause a move, and a new piece is not generated, unless the tilt would change the board. For example, an attempt to tilt board (e) north again would not result in a change, the game would not generate a new piece, and the player's turn would not end.

Each time two tiles merge to form a larger tile, the player earns the number of points on the new tile. The game ends when the current player has no available moves (no tilt can change the board), or a move forms a square containing 2048.

3 The API

To display the moves in this game and to receive input, you will use some classes that we've provided. The class `game2048.gui.Game` represents a game, and provides the following methods:

addTile Places a tile with a given value at a particular row and column. Rows are numbered from 0 to 3 from the top, and columns from 0 to 3 from the left. There must be no pending, undisplayed moves (see `moveTile` and `displayMoves`) when you call `addTile`.

moveTile Moves a tile from one given position to another. It's OK to (and optional)

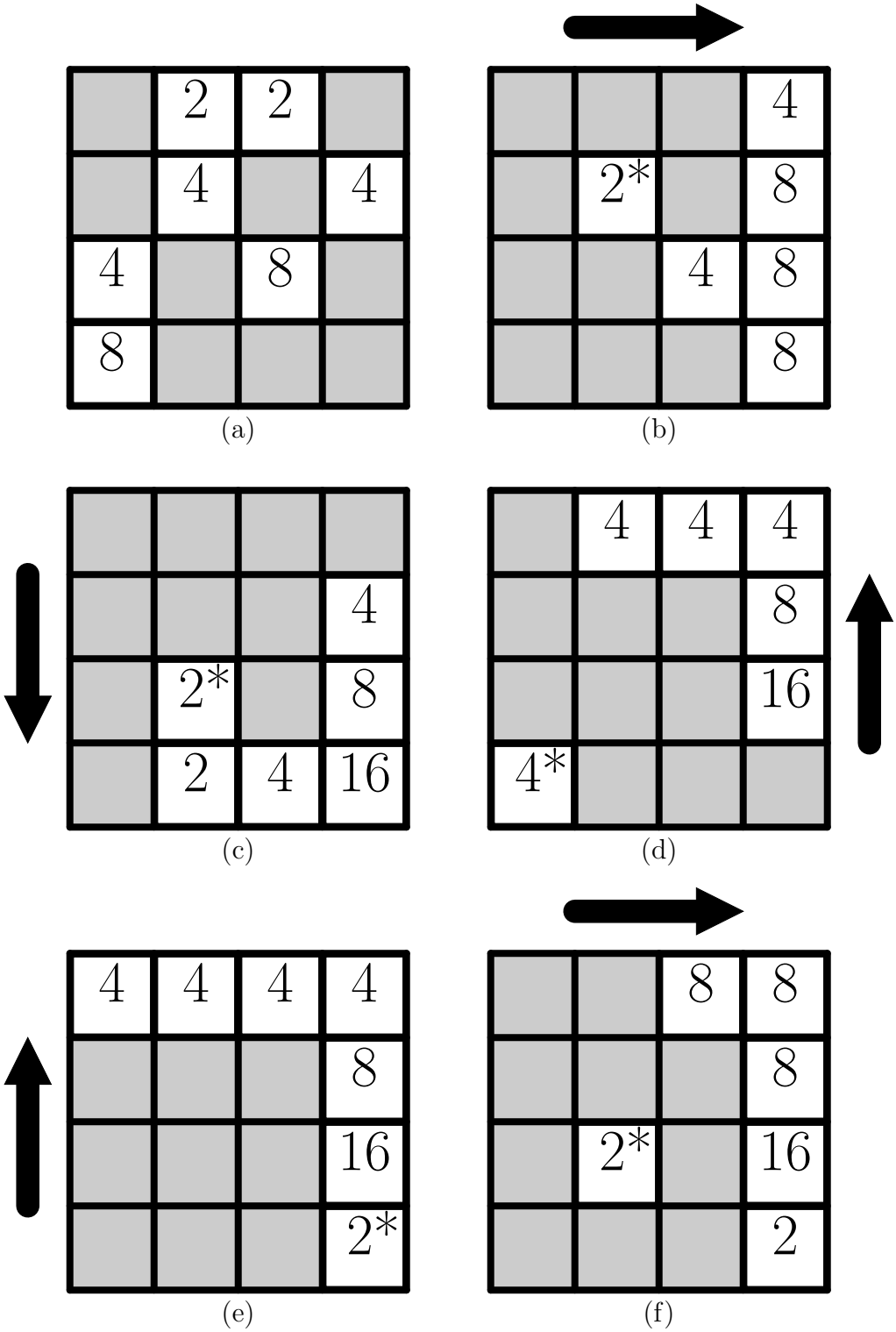

Figure 1: (a) Possible configurations in the 2048 game. (b)–(f) The results of sliding (a) to the right (east), south, north, north, and finally east, each time randomly adding a new tile to an empty square (indicated with an asterisk).

to "move" a tile to its current square. Move each tile at most once before calling `displayMoves`, and move tiles only to empty squares.

mergeTile Moves a tile from one given position to another, already occupied position, merging the two tiles. Do not move a merged tile—or merge a tile after calling `moveTile` on it—without first calling calling `displayMoves` in between.

displayMoves Displays the results of `moveTile` calls since the last call to `clear`, displayMoves, or the start of the game. Call it after moving all necessary tiles for a particular move.

getRandomTile Returns a random row number, column number (numbered from 0, with (0, 0) at the top left), and tile value (2 or 4). There is no guarantee this square is unoccupied. To use this to find a place to add a tile, therefore, you must call it repeatedly until it yields the row and column of an unoccupied square.

clear Clear the board for a new game.

readKey Returns the next key or button that the player has presses. Keys are returned as strings. The ones of interest for this game are the direction keys (`"Up"`, `"Down"`, `"Right"`, and `"Left"`), and the button "keys" `"Quit"` and `"New Game"`. When you see the "Quit" button, your program should exit (use the Java call `System.exit(0)` for this purpose), and when you see "New Game", clear the board, set the score to 0, and start a new game. Your program should ignore all other keys.

setScore Displays scores (that of the current game and a high score for a series of games). Make sure you have called this with the final score for a move before calling `displayMoves` for that move. You can call it several times (say after each merge) as long as the last is correct.

endGame Call this when the player has no moves or creates a 2048 key (either of which ends the current game), and after using `setScore` to display the final score and the maximum final score so far (that is, update the maximum game score whenever a game ends normmally). Don't use `endGame` or update the maximum score when a game ends prematurely (by "New Game" or "Quit").

To set up the board for the first move, add two tiles with `addTile`. To make a move, use `moveTile` and `mergeTile` to move all tiles to their ultimate destinations. To merge two tiles, move one of them (if necessary) to the final destination with `moveTile` and then move the second to the same final destination with `mergeTile`, indicating the new (summed) value for both. Then use `setScore` to update the score (if needed). Be sure to call `displayMoves` only after making all necessary calls to `moveTile`, `mergeTile`, and `setScore` for a move.

Use `randomTile` to select locations and values of new tiles. Do not call it unnecessarily (e.g., after a game ends). For testing purposes, we will feed in chosen "random" values, and you'll get incorrect results if you skip values.

You should not modify the classes in the package `game2048.gui`. Your program must work properly with the original package.

Example. To display the results of tilting the board shown in Figure 1a to the right, you would call the following methods on your `Game` object (denoted `G` here). Let's assume that the maximum game score so far is 1200 and the score for this game is 50 before this move.

```
// 2-tile at row 0, column 2 goes to (0, 3).
G.moveTile(2, 0, 2, 0, 3);
// 2-tile at row 0, column 1 goes to (0, 3), merging into the tile
// that just moved there.
G.mergeTile(2, 4, 0, 1, 0, 3);
// The 4-tiles at (1, 1) and (1, 3) merge into an 8-tile.
G.mergeTile(4, 8, 1, 1, 1, 3);
// 4- and 8-tiles on row 2 simply move right.
G.moveTile(4, 2, 0, 2, 2);  G.moveTile(8, 2, 2, 2, 3);
// 8-tile on the last row moves right.
G.moveTile(8, 3, 0, 3, 3);
// Display the score for the moves above (you can also call this at
// other points above with intermediate values if you want).
G.setScore(62, 1200);
// Display the moves above.
G.displayMoves();
// Add the new tile at (1, 1) for the next move
G.addTile(2, 1, 1);
```

4 Instrumentation and Testing

To facilitate automated testing of your work, the `gui` package has a few features that you can use to record sessions and to play back moves for testing or debugging purposes. The skeleton is set up so that when you start your program with

```
java game2048.Main --log
```

you'll get a record on the standard output of all of the keys returned by `readKey` and all the results returned by `randomTile` in the order that your program called them. You can capture this log using redirection, like this:

```
java game2048.Main --log > script1
```

The `--seed` option will allow you to prime the random number generation so that you can get the same set of random numbers each time:

```
java game2048.Main --seed=42
```

The same seed produces the same random sequence.

Finally, the `--testing` option reads in a script produced by `--log` and uses it (in place of user clicks and random numbers) to supply the results of `readKey` and `randomTile`. It also prints out data about what calls on the API your program makes (which we use to test the program). For example, the read back the file `script1`, use

```
java game2048.Main --testing < script1
```

5 Algorithm

The obvious way to keep track of the board is to use one or more 2D arrays to keep track of the values of the tiles in each location. It would be easy if the only key the user pressed during play was "Up" (or north). All pieces on row 0 (the top row) stay put, and you can proceed row-by-row up from 0, computing how far each tile can go (and which merge), since if you process in that order, tiles will not have to move again when you go to later rows.

The only problem is that you then have to do the same thing for the other three directions. If you do so naively, you'll get a *lot* of repeated, slightly modified code, with ample opportunity to introduce obscure errors. Therefore, we've included in the skeleton a couple of methods that will allow you to re-use code that works for "Up" on all the other directions:

```
/** Symbolic names for the four sides of a board. */
static enum Side { NORTH, EAST, SOUTH, WEST };

/** Return the row number on a playing board that corresponds to row R
 *  and column C of a board turned so that row 0 is in direction SIDE (as
 *  specified by the definitions of NORTH, EAST, etc.).  So, if SIDE
 *  is NORTH, then tiltRow simply returns R (since in that case, the
 *  board is not turned).  If SIDE is WEST, then column 0 of the tilted
 *  board corresponds to row SIZE - 1 of the untilted board, and
 *  tiltRow returns SIZE - 1 - C. */
int tiltRow(Side side, int r, int c) { ... }

/** Return the column number on a playing board that corresponds to row
 *  R and column C of a board turned so that row 0 is in direction SIDE
 *  (as specified by the definitions of NORTH, EAST, etc.). So, if SIDE
 *  is NORTH, then tiltCol simply returns C (since in that case, the
```

```
     *  board is not turned).  If SIDE is WEST, then row 0 of the tilted
     *  board corresponds to column 0 of the untilted board, and tiltCol
     *  returns R. */
    int tiltCol(Side side, int r, int c) { ... }
```

Therefore, one approach to tilting the board would be:

- Transfer all of the current contents of the array you use to represent the board's contents to a temporary array, using `tiltRow` and `tiltCol` to turn the board in the process.
- Do your computation on the temporary board as if it were tilted north (up). Be sure in doing so that any calls to `moveTile` and `mergeTile` translate their coordinates with `tiltRow` and `tiltCol`.
- When done, transfer your results from the temporary array back to your board array.

6 Submitting Your Work

When you are in your working `proj0` directory, the command

```
hw submit
```

will submit your assignment. Be sure to respond to all prompts and to make sure the messages you get indicate that the submission was successful. Don't just "say the magic words" and assume that everything's OK.

We strongly suggest that you use the command

```
hw commit
```

as often as possible (also while in your `proj0` working directory). This saves a snapshot of the current state of your project that you can restore at any time in the future (should your dog eat your laptop or some other disaster occur). If you lose all your work and have not been committing it regularly, don't expect much sympathy from the staff; we will consider it a "valuable learning experience."