ICS3U Final Project: 2048

Overview

2048 is a single-player puzzle game originally developed in March 2014 by 19-year-old Italian web developer Gabriele Cirulli. The goal of the game is to slide numbered tiles on a grid to combine them and create a tile with the number **2048**. The original version of the game together with the rules can be found here: https://play2048.co/

Game Rules

- The game is played on a **4×4 grid**.

Game Play

- When the player presses one of the arrow keys (**up, down, left, or right**), all tiles slide in that direction.
- If two tiles with the same number collide while moving, they merge into one tile with a value equal to the **sum** of the two tiles.
- When three consecutive identical tiles are present, only one merge happens per move, starting from the direction of the slide.
  For example: Given three horizontal consecutive tiles of

2	2	2

  ← Swiping left combines the two left most tiles resulting in

4	2

  → Swiping right combines the two right most tiles resulting in

2	4

- After each move, a new tile appears in a random empty spot on the board. It will have a value of **2** (90% chance) or **4** (10% chance)

Ending Conditions

- **Win**: The player creates a tile with the value **2048**.
- **Lose**: No legal moves are possible—there are no empty spaces and no adjacent tiles with the same value.

Scoring

- The game keeps track of the player's score.
- Points are awarded whenever tiles merge, equal to the value of the newly created tile.

Project Goals

You are to create a program which enforces the rule of the 2048 game. You are NOT to implement the AI (artificial intelligence) to play the game. The following features must be implemented:

- Game begins with an empty 4×4 grid containing two randomly placed tiles.
- Player inputs a move by a key press: **up, down, left, or right**.
- All tiles slide in the specified direction, filling any empty spaces.
- Any two **adjacent tiles with the same value** will combine into one tile with a value equal to the sum of the two.
- A tile can only merge once per move, and no chain reaction within a single move.
- A move that does not result in any tile movement or merging should be considered invalid. No new tile should spawn in that case. No action is performed and another move should be expected.
- Points are earned after each move based on the values of the newly created tiles. The total score must be updated and displayed after every move.
- After each valid move, one new tile, 2 or 4, should be added to a random empty spot on the grid. 90% of the new tiles are 2s and 10% are 4s.
- The updated game board must reflect after any change of the grid.
- After each move, determine and display
  - "**Game won"** if a tile with the value **2048** is created.
  - **"Game over"** if the board is full and no valid moves remain (i.e., no adjacent tiles can be combined).
- Player can restart the game at any time.
- Player can **save** the current state of the game to a file.
- Player can **load** a saved game from a file.
- At the end of each game (win or lose), the player must be given the option to **restart or exit**.
- The game must have a **graphical user interface (GUI)**, and must be implemented using the provided GUI class.

Implementation Details

Your task is to implement the logic of the game. The graphic presentation of the game board is already implemented and included in the skeleton code, where your code will be built on. The program consists of three Java files:

`Game2048Listener.java`
This class detects and handles key presses, and button clicks by the users. Do not modify this file.

`Game2048GUI.java`
This class creates the GUI of the game. It represents the game grid using a 2D array with `NUM_ROW` rows and `NUM_COLUMN` columns, with index `[0][0]` representing the top left slot, and `[0][NUM_COLUMN-1]` representing the top right slot. This class provides several methods to update the GUI. Please refer to the API for the descriptions of the methods. You do not need to touch this file.
This file contains the `main` method, therefore, the program is run on this file.

`Game2048.java`
The file contains the implementation for the logic of the game, which you are responsible for.

Graphic Files

You will customize the look of your program with your graphics files. A set of 11 graphics files are required for each number tile and 1 for the logo banner at the top. The size (in pixel) of the number tiles (which are squares in shape) is specified in the constant `PIECE_SIZE` in the `Game2048GUI.java`, and sizes of other components of the game depends on this number. In particular, the size of a number tile is 75 X 75 pixels, and the dimension of the logo is 324 X 108 pixels. The files for the number tiles must be named `<icon><#>.jpg`, i.e., `icon2.jpg` … `icon2048.jpg`. The logo banner must be named `iconLogo.jpg`. All graphic files must be placed in a folder called `images` (specified by constant `ICON_FILE_FOLDER`).

Control Flow

When the program runs from the `Game2048GUI` class (which contains the main method), the following sequence of actions occurs:

1. GUI is initialized and displayed
2. `newGame()` method in the `Game2048` class is called to initialize the game logic.
3. Program waits for user input.
4. When the user presses an arrow key or clicks a button, the corresponding method in `Game2048` is called to handle the action.
5. Steps 3 & 4 repeats until the player either wins (by reaching 2048) or the game ends (no valid moves remain). At that point, the player can choose to restart or exit the game.

Steps 1 & 3 are already implemented in the provided skeleton code. You are responsible for implementing the methods that are called in step 2 & 4.

"Global" Variables

Variables declared outside all methods can be considered "global" variables. These variables can be accessed and modified in any non-static methods within the class. In the `Game2048` class, all core data that represents the game state should be stored as "global" variables. For example:

- the 2D array representing the play grid,
- the current level or stage of the game,
- the player's current score,
- any constants related to grid size or value representing an empty slot.

Defining these as "global" variables ensures they are accessible throughout the class and maintain their values between method calls without having to pass them as parameters.

Non-static variables and methods

In this program, all the methods you create should be non-static. You are not responsible for calling your methods from the main method, therefore you do not need to worry about creating an instance of the object to call them.

To call one non-static method from another within the same class, you simply call the method name. For example, inside your `newGame()` method, you can call another method like `initGrid()` directly, without using an object name:

```
public void newGame() {
    initGrid();  // valid because both methods are
                 // non-static and in the same class
}
```

Development Sequence

Here is a suggestion on the sequence of steps you should follow to proceed with your implementation:

1. Open the file `Game2048.java`, please note a list of constants as well as a method called `Game2048(Game2048GUI gameGUI)` has been declared. These are required for the program to function. Do not touch them (except changing values of the constants)
2. Define additional "global" constants and variables that are necessary outside all methods. Consider the constants, e.g., numbers used to represent empty slots, as well as core data required in this program.
3. Create the method `newGame()`:
    - Parameters: none
    - Returns: none
    - Descriptions: start a new 2048 game
        - Create the 2D array (representing the play grid) with dimensions defined by the constants. Please note the array has already been declared outside the method, so you only need to create it here. Re-declare the array inside this method will cause an error.
        - Initialize other values representing state of the current game
        - Add the two initial number tiles
        - Call corresponding methods of the GUI object (variable) `gui` to reset the game board visually

4. Create methods that implement the logic of the game. In particular, the following methods must be implemented as specified. They are called by the `Game2048Listener` class when keys are pressed or buttons are clicked:
   - `move`:
     - Parameter: `int`, representing the direction (integer representing the directions are defined as constants)
     - Returns: none
     - Descriptions: This method is called by the GUI code when player presses one of the arrow keys. The parameter indicates which arrow key is pressed. This is where the main logic of the game is. Every move by the players should update the 2D array according to the rule of the game. Make sure to call the appropriate method of the object (variable) `gui` to update the GUI. For example `gui.displaySlot(0, 3, 32)` will set the slot at row #0 and column #3 to the icon corresponding to number 32.
   - `saveToFile`
     - Parameters: `String`, representing the file name
     - Returns: `boolean` to indicate if the game status is successfully saved to the file
     - Descriptions: Write the information of the current game status to a text file with the given file name.
   - `loadFromFile`
     - Parameters: `String`, representing the file name
     - Returns: `boolean` to indicate if the game status is successfully loaded from the file
     - Descriptions: Read from a text file, with the given file name, the information needed to restore the status of a saved game. Be sure to call the method required to update the game board visually.

Hint to get started

The initial skeleton code does not compile. If you want to compile the code for testing purpose, you must first declare the following methods (in `Game2048.java`) with the proper header (correct parameters and return type); no implementation of the method is necessary for compilation.

- `newGame`
- `move`
- `saveToFile`
- `loadFromFile`

Once you can compile the code, you can run the program from the `Game2048GUI` class and you will see the game board appears.

General Information

You should test your code continuously as you program, rather than waiting until all methods are complete before compiling and running it. Here are some effective strategies for testing your methods as you build them:

- Create temporary test code in a separate class
- Create a temporary method to display the content of the game board to the console

You are not required to follow the exact design stated above. You may choose to implement your program with different algorithms, methods, organizations. Methods list under section "**Hint to get started**" are required. Aside from the correctness of the functionalities, you will also be graded on the organization of your code, the efficiency of your algorithms, the proper application of methods, programming styles and documentations.

The project is worth 10% of the course mark.

Responsibility and Effort
This final project is the summative activity for this course. Most of the work must be done during class time. Exceptions are on tasks that need extra time to complete. Any unreasonable amount of work that is done outside class will not be accepted and will receive a grade of 0. Poor attendance and in-class effort will significantly affect your final grade. A responsibility and effort factor will be given and is used to adjust the final mark for the project.

Due Date of Final Project
Friday, June 13, 2025, 11:59pm

Rubrics

Program Design and Efficiency	**15**
☐ No redundant or repeated code ☐ Efficient use of data structures (e.g., avoiding extra arrays) ☐ Good choices of programming structures (e.g., while loop vs. for loop) ☐ Logical, clean implementation of core features ☐ Clear organization into methods that support reuse and testing	
Programming Style	**10**
☐ Clear and meaningful identifier names ☐ Consistent and logical indentation and vertical spacing ☐ Use of constants appropriately ☐ Logical ordering of variables, methods, and control flow	
Program Documentation	**10**
☐ Class-level program header comment ☐ Method headers comment that clearly state purpose, parameters, and return types ☐ Inline comments explaining complex logic or algorithms ☐ Clear labeling and explanation of any extensions (if any)	
Implementation	**65**
Game Initialization and Setup *(8 marks)* ☐ Necessary constants are declared and initialized (1) ☐ 2D array correctly declared, created and initialized (3) ☐ Other global variables declared where appropriate and initialized (2) ☐ Two starting tiles (2 or 4) placed randomly (2) **Game Mechanics and Logic *(28 marks)*** ☐ Tiles move correctly in the specified direction (4) ☐ Correct merging of identical adjacent tiles (6) ☐ After a move, all tiles shift to fill gaps properly (6) ☐ Each tile merges only once per move (no chain merging) (1) ☐ Invalid moves detected and no action occurs (2) ☐ Current level is updated correctly (2) ☐ Score is calculated and updated based on the merged tile value (2) ☐ New tile spawns after each valid move (3) ☐ New tiles follow 90% 2 and 10% 4 probability (2)	

End of Game Detection *(7 marks)* ☐ Win condition detected correctly (tile = 2048) (1) ☐ Lose condition detected correctly (no moves and full grid) (3) ☐ Game restarts properly when selected (2) ☐ Game terminates without error when exit is selected (1) **Saving and Loading *(7 marks)*** ☐ Information is saved to file correctly to keep track of current game status (3) ☐ Information is loaded from file correctly to restore game status (3) ☐ Basic error-checking for file access (for saving and loading) (1) **GUI Integration *(15 marks)*** ☐ Proper response to each directional key press (1) ☐ GUI reflects updated grid after at all points of the game (3) ☐ Score updates correctly on GUI (1) ☐ GUI reflects restart and load actions (2) ☐ Correct use of provided GUI class methods (2) ☐ Game status (win/lose) visually indicated and prevents further play (1) ☐ Player can select to restart game or exit after an end game (1) ☐ All required tile images are created and named properly (2) ☐ Logo banner images is present and named properly (1) ☐ All images placed in correct `image` folder (1)	
Total	**100**
Responsibility and Effort A score below 7 in this section will result in a reduced final mark	**10**
☐ Regular in-class progress ☐ Preparedness for class ☐ Appropriate use of class time and support ☐ Attendance ☐ Punctuality	