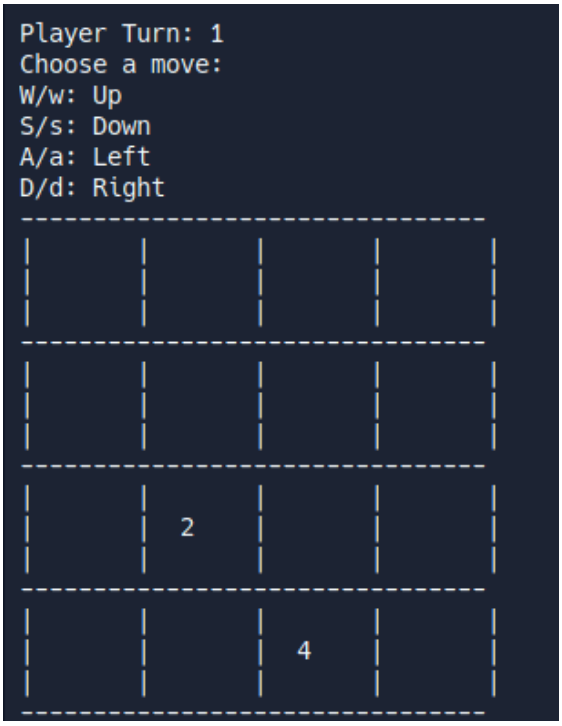
2048 Game Project

Congratulation on conquering the AP exam!

Now that we've grasped the basics of programming and Java, we have all the tools and knowledge we need to start building interesting things! In this project, you would re-create the popular game 2048 in Java



Game Rules

Let's first familiarize ourselves with the game rules by playing a few games. Keep the following questions in mind as you play:

1. How does the game start?

2. What happens to the board when you press one of the four movement keys: up, down, left, and right?

3. When do new tiles get added to the board?

4. How does the game end?

Puzzle pieces

> “The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.” - Frederick P. Brooks Jr

Building a software product is like collecting and piecing together a bunch of puzzle pieces, except we fabricate the puzzle pieces themselves!

To provide some structure on this puzzle quest, I've wired together some basic pieces, namely the game logic for handling the user input and the game view for showing the board. You can find a reference to them on the back of this sheet

Your main task here is to implement the class `Board`. This class represents the state of a board during the course of play and contains method for updating the state. Those methods are used by both the game logic and the game view

Checkpoints

Below is the recommended route for implementing the game

Checkpoint 0: Implement the basic methods and members of `Board` class

At this checkpoint, you should implement the constructor and the methods `setBoard` and `getBoard`

Here, you'd decide whether you like to represent the board as a 1D array or a 2D array of digits/tiles. There're pros and cons to each approach, so pick the version make the most sense to you

Checkpoint 1: Spawn new tiles

At this checkpoint, you should implement the method `spawnNewDigit`

The method would add a given digit/tile to the board (2 or 4). It should only add to an empty space on the board, and the empty space should be randomly chosen.

Checkpoint 2: Move and Combine tiles

This checkpoint is the main crux of the game

You would implement the four movement handling methods: `processLeftMove`, `processRightMove`, `processUpMove`, and `processDownMove`

Those four methods would update the board when the user inputs the corresponding movement key

The key is to break up how the tiles are **combined** and **moved** in the respective rows and columns

Checkpoint 3: Get Game State

Lastly, you would implement the method: `getGameState`

This would signal to game logic whether the game has been won, lost, or is still in progress

Test, test, test

A big part of software development is testing. Because as the thing you're building increases in complexity, so are the amount of things that can go wrong. Testing ensures the code you write actually works

There're two ways you can test: **Manual tests** and **Unit tests**

Manual Test

In manual test, you would run the game end to end and test the new code by manually providing the input

To do this, you'd un-comment the relevant portions in the class `Game2048` (indicated by the comments), click `Run`, and play the game

Unit Test

Many times, manual testing wouldn't spot problems early on in the system; this is because manual testing is end-to-end and cannot isolate each part of the system

Unit Testing is a key part to systematic testing. It comprises of test cases written for specific class components (`Board` class in this case) that are

executed automatically. Those test cases would call the the class' methods with a set of *input* and expect it to generate a set of *output*

I've written few test cases that you should run for each part of the checkpoint, you can find how to run the unit tests in the back of this sheet.

You can also try extending those test cases to cover more possible test scenarios