# CIS 163
# Project 2 – 1024/2048 Game

## Due Date

- At the beginning of the lab, Tuesday 2/21/17.

## Before Starting the Project

- Review array, and Interfaces (Chapters 6, 7)
- Read this entire project description before starting

## Learning Objectives

- Apply the MVC design architecture to separate the model and view components of an application program
- Rearrange and update numerical values stored in a two-dimensional array, following specific rules
- Using ArrayList of objects
- Using Stacks to implement an undo feature
- Using Java exception mechanisms for notification of error conditions

## Introduction

The game that you are about to implement in this project is widely known as 1024 Game or 2048 Game. The two name variants are used depending on the winning objective the player must achieve. In this single-player game, the player slides numbered tiles on a grid (usually 4x4) to combine and add identical tiles to obtain a value of 1024 (or 2048). The best way to understand this game is to play it online.

**Game 1024 Rules**

- The rules of this game can be summarized as follows:

- The player slides the tiles only in one of the following directions: North, East, South, West
- A single slide action moves **all the tiles** in the requested direction as far as possible, eliminating any gaps between two tiles
- Two adjacent tiles of the **same value are merged** into one tile with a new value double its original.
- The tiles are merged beginning from the **same end of the slide direction**. Consider a row of three 4s and three blanks (dots represent gaps/blanks)

```
4 . . 4 . 4
```

- When the tiles slide *westbound*, the *first two on the west side* merged producing two numbered tiles and four blanks:

```
8 4 . . . .
```

- When the tiles slide *eastbound*, the *last two on the east side* merged producing two numbered tiles and four blanks:

```
. . . . 4 8
```

- **In a single slide action**, merging is applied only to the **current set** of tiles, and **not to the recently merged** tiles. For instance a row of four 8s: 8 8 8 8 would only produce two tiles of 16 (and two gaps): 16 16 . . (or . . 16 16). The two 16s **are not automatically merged** into one tile of 32. The player, of course, in **a subsequent action** may merge them into one tile of 32.
- At the beginning of the game, the board is initialized with two random values (must be $2^k$, i.e. 1, 2, 4, 8, 16, 32, …) at two random locations.
- After each slide a new random value is generated at a random location
- The player won if she is able to add up the tiles to a value of 1024. *This is the default winning value, but our game logic must be written to accept different winning value).* The player lost the game when the board is full with no tiles having the winning value.

<u>**IMPORTANT:**</u>

**The instructor will explain in class on how to solve the sliding portion of the assignment. In the explanation, the instructor will use a specific technique that will be required for this assignment (If you miss class during this explanation, please stop into the**

**office or gets notes from a fellow student) For this assignment you are <u>NOT</u> permitted to use an Arraylist to handle the sliding of the board. You must slide the board with loops, 1 and/or 2D arrays.**

## Your Assignment

To help you understand the clear separation between the “user facing components" versus the “number crunching elements" of an application, this project is deployed in two parts (Project2 and Project3):

- In project 2 you will:
  - Develop the game engine that complies with the NumberSlider interface
  - **You must NOT use an arraylist to solve the slide problem,** please see important comment above
  - Use the provided JUnit test file to verify that your game implementation passes all the test cases
  - Run (and play) the game using the text mode user interface provided by your instructor

- In Project 3 you will:
  - Develop a GUI program that interacts with the game engine implemented in Project 2

(Two separate demos will be required, one for each project2 and project 3)

## Programming Resources

Download the following starter files (in BB) for project 2:

1. Cell.java, GameStatus.java, SliderDirection.java
2. NumberSlider.java
3. TextMode UI files (TextUI.java)
4. Unit Tester (TenTwentyFourTester.java)

## Recommended Steps

**Step 1:** Create a New Project
Create an Eclipse or IntelliJ project and then create a new package under the project. Suggested package name: game1024

**Step 2:** Download the Starter Files
Download the following file(s) to your project

- Cell.java
- GameStatus.java
- NumberSlider.java
- SliderDirection.java
- TextUI.java (user interface in text mode provided by your instructor)
- The JUnit test file (TenTwentyFourTester.java)

**Step 3:** Implement the NumberSlider Interface
Under the package created in Step 1, create a new class that implements the NumberSlider interface.  For example: public class NumberGame implements NumberSlider {Write the require methods dictated by the interface, leave the method body unfinished for now (except for the required return statement). Be sure the class has no compile errors prior to moving on to Step 4.

**Step 4:** Run the JUnit Tests
Before running the JUnit test file (TenTwentyFourTester.java), locate the first TODO item in this file and use the class created in Step 3 to instantiate an object into the gameLogic variable. **Expect to see test failures**. Your goal (in the next step) is to turn all these failures to passing test cases.

**Step 5:** Develop the Game Engine
Instead of writing all the methods at once, it is highly recommended that you focus on one method at a time with the goal of passing all the relevant test cases related to that method.

1. Read the online documentation of each method specified in the interface. The description of each method should provide sufficient information about the logic of its body and *optionally* the **exception that must be thrown** when a logical error occurs.
2. Use a **2D array of integers** to represent the game board. *Please refrain from using data types other than integer for this 2D array, this is the simplest and most efficient representation for the game board*. This 2D array will be **one of the main** data structures for implementing the game logic. **Add other necessary arrays** (either 1D and 2D) and **variables** of other type as needed.
3. Add **private helper methods** to organize the overall logic. This is one of the required grading items
4. Implement each method as dictated by the interface. Pay special attention to the slide() method. The method is required to handle moving and merging the tiles in **four directions** (UP, RIGHT, DOWN, LEFT). *You may be tempted to write a solution for handling one direction and make three additional copies*. Carefully design your program (especially the private methods) in order to **avoid writing four copies of almost identical** code. To help you alleviate this problem, consider designing your private helper methods to take **appropriate method parameters**. Doing so makes those methods *generic enough* to perform several tasks. (*See the section "Method With(out) Parameters below*.)
5. **To assist during debugging**, consider adding a private method for printing the content of the game board 2D array. Invoke this method throughout the other methods for *easy visual inspection of the 2D array*.
6. Use a Stack for implementing the undo() method. Each element of the stack stores the current state of the game board. Prior to updating the game board in the slide() method, the **current state of the game board should be pushed** to the stack. Using this approach the undo() method would restore the game board to the topmost stack element and then pop it.

**Step 6:** Repeat Steps 4 and 5
You will continue working in this fashion until your game engine implementation passed all the test cases.

**Step 7:** Play the Game in Text Mode
Play the game using the text mode UI.  (CHANGE, the code provided to track the number of times the player wins the game out (percentage))

## Methods With(out) Parameters  (something to think about)

In CIS162 you learn how external data can be passed into a method via parameters. The main objective of designing such a method is to allow the method perform **several variants of similar tasks**. Consider the following progression of examples for printing a brief greeting.

```
public void sayGreetingToAndy() {
   System.out.println ("Andy, how are you today?");
}
```

The above method can only do **exactly one task**: printing a *specific greeting* ("how are you today") for a specific person (Andy). To make it more generic, we can parameterize the method with the greeting text:

```
public void saySomethingToAndy(String greeting) {
   System.out.println ("Andy, " + greeting);
}
```

At **runtime**, we just need to call this one method and supply a **variety of actual greetings**:

```
saySomethingToAndy ("glad you are here");
saySomethingToAndy ("nice to finally see you");
saySomethingToAndy ("hope to see you soon");
```

The next level of generalization is, of course, parameterizing the person's name so the greetings can be extended to other individuals besides Andy.

## Turn In

- Be sure to add proper **Javadoc comments** throughout your source code
- At the top of the source file, write a comment block that includes the @author and @version tags. Include your name **after** the @author tag.
- At the beginning of each method (public and private), write a comment block that describes what the method does. For each parameter, use the @param tag to explain how the parameter is used by the method. When the method returns a result, explain it using the @return tag.
- Keep each line (statements and comments) not to exceed the right margin at column 75. Break a long statement into several lines when it exceeds this margin.
- Be sure **your source code is properly indented**. Modern IDEs (like Eclipse and IntelliJ) provides a menu command shortcut for "Reformat Code". Learn how to use this editor command and *avoid indenting your source code manually* line-by-line.

# Project 2: “1024 Game" Program Rubric.

| | |
|---|---|
| Student Name | |
| Due Date | |
| Date Submitted, Days Late, Late Penalty | |

| Graded Item | Points | Comments and Points Secured |
|---|---|---|
| • Javadoc Comments and Coding Style/Technique<br>• (http://www.cis.gvsu.edu/studentsupport/javaguide)<br>• Code Indentation (auto format source code in IDE)<br>• Naming Conventions (see Java style guide)<br>• Proper access modifiers for fields and methods<br>• Use of helper (private) methods<br>• Using good variable names<br>• Header/class comments<br>• Every method uses @param and @return (1 sentence)<br>• Every method uses a /***************** separator<br>• Overall layout, readability, No text wrap<br>• Using /** … / for each Instance variable<br>• Has many inner “inner" comments | 10 | |
| **Basic Functionality**<br>Using JUnit testing<br>Using TextUI testing | 35<br>35 | |
| **Misc. (code design, MVC,)** | 20 | |
| **Total** | **100** | |

**Additional Comments:**