CSED332 Assignment 1

Due Tuesday, September 19

Objectives

- Make sure you can install and run IntelliJ IDEA.
- Create your GitLab account on `https://csed332.postech.ac.kr`.
- Use Gradle
- Learn Java programming language

Getting Java

- We will use IntelliJ IDEA in this class. You will need Java, or more specifically, the Java Development Kit (JDK). Get the latest release (20.0.2) from:

  `https://www.oracle.com/java/technologies/downloads/`

Getting IntelliJ IDEA

- Download and install the latest version of IntelliJ IDEA Ultimate (version 2023.2.1) from

  `https://www.jetbrains.com/idea/download`

  You may need to *apply for a student license* at `https://www.jetbrains.com/student`, unless you already have a (free) license.

Microsoft Teams

- Make sure your are invited to the `csed332-2023-Fall` team (if not, please contact TA).
- Leave any message in the `Homework1` channel.

Create your GitLab account

- Create your GitLab account using your Hemos ID and your name in English (the same as those in POVIS) at the GitLab repository server

  `https://csed332.postech.ac.kr`

- Create SSH keys, following `https://csed332.postech.ac.kr/help/user/ssh.md`.
- Create a *private project* with name homework1, and clone the project to your machine.

IntelliJ IDEA and Gradle

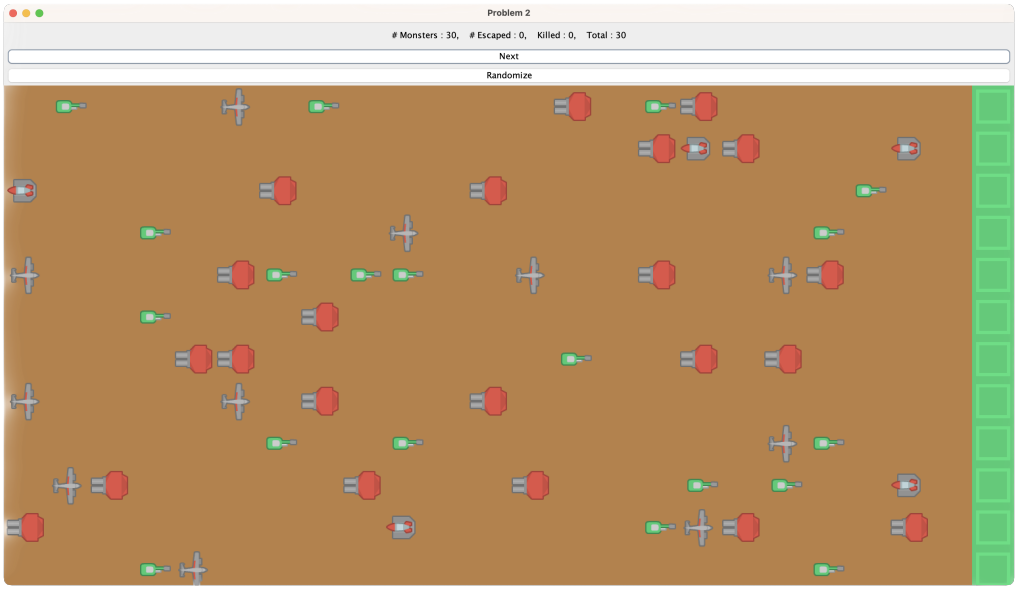
- Download the attached file `homework1.zip`, which contains two subdirectores `hw1-problem1` and `hw1-problem2`. Each of them can be imported as a separate project into IntelliJ IDEA.
- In this assignment, your code need to be compiled using only Gradle in a command line. You can use IntelliJ IDEA, but your code will be graded using Gradle.
- To compile your code using Gradle, you can go to where your `build.gradle.kts` is for each problem, and execute the command `gradle compileJava`.
- You may need to install Gradle to run it in a command line. For download instructions and tutorials, see `https://gradle.org/guides`. Use the latest version: Gradle 8.3.

Problem 1

- The goal is to create a simple account management system for banking as follows. Check more detailed information in the skeleton code.
  - A *bank* maintains two types of *accounts*: *simple interest* accounts and *compound interest* accounts. You can deposit money into an account, withdraw money from an account, and transfer money from one account to another.
- The `src/main` directory contains the skeleton code. You should implement all methods with *TODO* in the following classes:
  - `Bank`
  - `AbstractAccount`
  - `SimpleInterestAccount`
  - `CompoundInterestAccount`
- The `src/test` directory contains some test methods in `BankTest.java`. You can run the test cases by running `gradle test`.
- Your code will be graded by Gradle, using extra test cases (different from `BankTest.java`) written by teaching staff. Make sure the command `gradle test` works.
- Do not modify the existing interfaces, the class names, and the signatures of the public methods. You can add more methods or classes if you want.

Problem 2

- The goal is to implement a mini tower-defense game. As shown in the following figure, a *game board* contains a number of *monsters* and *towers*.
  - A game board consists of $n \times m$ tiles, and the position of each tile is given by a pair (x, y), where $0 \leq x < n$ and $0 \leq y < m$.
  - (x, y) and (x', y') are *adjacent* iff (i) $x = x'$ and $|y - y'| = 1$, or (ii) $|x - x'| = 1$ and $y = y'$. E.g., $(2, 3)$ and $(1, 3)$ are adjacent, but $(2, 3)$ and $(1, 2)$ are *not* adjacent.
  - Monsters move towards the right end (marked as green squares). There are two types of monsters: *ground monsters* and *flying monsters*.



  - Towers attack (or kill) *all* adjacent monsters. There are two types of towers: *air towers* attack only flying monsters, and *ground towers* attack only ground monsters.
  - Two flying monsters cannot be on the same tile. Similarly, two ground objects (ground towers, air towers, or ground monsters) cannot be on the same tile.

- A game consists of a sequence of rounds. In each round, monsters and towers perform the following actions, and the game moves on to the next round.

  1. All monsters on the green tiles at the right end "escape" from the game board.
  2. Each tower attacks (and removes) all adjacent monsters of its type.
  3. The remaining monsters can move to an adjacent tile (or stay put).

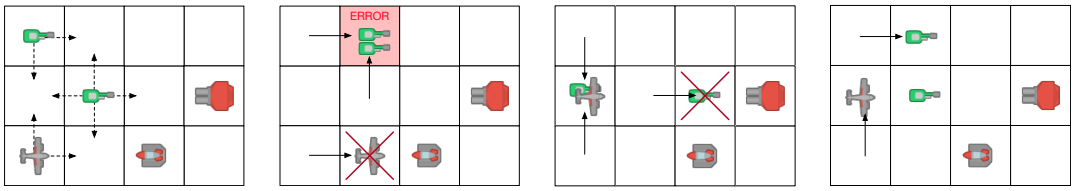
- In this assignment, you implement an object-oriented "model" of the game described above, given by the following classes.

  - `GroundTower` and `AirTower` implement the interface `Tower`, with the method `attack`, which returns the set of monsters to be attacked by that tower in the current round.
  - `GroundMob` and `AirMob` implement the interface `Monster`, with the method `move`, which returns an adjacent position to move to for the next round.
  - `GameBoardImpl` implements the interface `GameBoard`, maintains towers and monsters, and contains several methods for playing the game, such as the method `step`.

- As part of this assignment, students will define and implement a simple monster AI (the `move` method of `GroundMob` and `AirMob`).

  - The game should not fall into error states (e.g., two ground objects on the same tile) due to monster movements. See the method `isValid` of `GameBoard`.
  - Each monster should try to reach the goal tile as soon as possible, avoiding being attacked by towers, and not falling into error states.

  - For example, consider Board 0 below. Each monster can move to an adjacent tile or stay on the current tile in the next round. Consider three possible movements:



Board 0 Board 1 Board 2 Board 3

    * Board 1 is in an error state, because two ground monsters are on the same tile.
    * Board 2 is in a valid state, but *not* optimal since one ground monster is killed.
    * Board 3 is in a valid state and no monsters are killed by towers.
  - We will run your strategy against several game boards for evaluation. Students who develop better strategies will receive additional bonus points.

- We provide a simple GUI that runs the game on your model. It will not be used for grading, but you can inspect your implementation using this GUI.
  - The state of `GameBoard` is displayed in the GUI panel.
  - The NEXT button executes the method `step` of `GameBoard`.
  - If the game falls into an error state, NEXT will be disabled.
  - The RANDOMIZE button generates a new random game.
- The `src/main` directory contains the skeleton code. You should implement all classes and methods with *TODO* in the above classes.
- The `src/test` directory contains some test methods in `GameTest.java`. Your code will be graded by Gradle, using extra test cases written by teaching staff.
- The command `gradle jar` will create a jar file in the `build/libs` directory, which can be executed using the command: `java -jar hw1-problem2-1.0-SNAPSHOT.jar`
- Do not change the existing interface, class names, and public method signatures as they are used for the GUI and for grading.
- You can add more methods or classes if you like. In particular, you may want to add (abstract) superclasses for towers and monsters to avoid duplicate code.

Turning In

1. Commit your changes in your `homework1` project, which includes two directories `hw1-problem1` and `hw1-problem2`, and push them to the remote repository.
2. Tag your project with "`submitted`" and submit your homework. We will use the tagged version of your project for grading.

   (see `https://docs.gitlab.com/ee/user/project/repository/tags/`)

Java Reference

- Java Language Specification: `https://docs.oracle.com/javase/specs/`
- Beginning Java 17 Fundamentals 3nd by Kishori Sharan and Adam L. Davis, Apress, 2022 (available online at the POSTECH digital library `http://library.postech.ac.kr`)