Computer Laboratory 9

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday November 15th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs have explicit instructions about useful topics to review/overview which will help you excel in this lab.
- For more rules see the lab rules document on canvas

UPDATES

- Clarification: The NumberGuesser class should be not consider negative numebrs as "valid" inputs.
- Clarification: Guesses are to be counted in the processMove function in both games – an invalid guess is not counted

2 Learning Objectives

This lab is designed with a few learning objectives in mind. By doing this lab you will:

- Practice working within the context of a provided parent class
- Yet more practice building functionally interesting classes
- Get a little background with Abstract classes
- Implement fun games
- See polymorphism in action
- Build something **fun!**

3 Introduction

"Game packs" are a common way to program and distribute groups of similar games. While they are less common these days than they used to be, game packs remain a great way to design and share groups of related games. A great, and enduring, example is Simon Tatham's Portable Puzzle Collection (`https://www.chiark.greenend.org.uk/~sgtatham/puzzles/`)

What makes "Game packs" particularly appealing to programmers is that, when well-chosen, the games can share common game engines, or other game code. By looking actively for this code re-use we can develop games much faster, while also producing a more varied and appealing end-product. The added structure of common game code can even make it easier to come up with new game ideas!

In this lab we will be building two simple text-based java games: a "number guesser" game (where the computer picks a random number and then gives you "higher" and "lower" feedback to help you guess the number) and a "hangman" word-guessing game (where you try to guess a word letter-by-letter). On it's own a number guesser would be tedious at best for programmers of your skill, and hangman, while more interesting, ultimately has little educational value. Therefore to make things more interesting we will begin by giving you a base Game class which provides a structure and basic "game engine" to your code. The challenge of this lab, therefore, is not "how do I program number guesser" but instead "what is the structure Game.java provides, and how do I use it?".

4 Requirement: Code Reading

You will need to begin by downloading the provided files from canvas. This section walks you through the purpose and design of these classes. Reading these classes is a requirement of this assignment, and you should do your best to understand how they work/are designed. Make sure you're asking questions if, after reasonable effort, the design of these classes is still unclear – the required programming will be nearly impossible if you don't understand these classes.

4.1 Game class

The first thing to understand is the basic "Game Framework" provided for this lab. The Game class is an abstract class that will be a parent class to both games you should write. As an abstract class it provides *structure* for future classes without being a complete class on it's own. In this case the Game class provides two things:

1. A concrete "play" method which structures what it means to "play a game"
2. A collection of abstract methods which are used to break the "game" process up into small bite-sized pieces:

- `getName()` – this function returns the game's name – and is used by the GameGrabber menu-class.
- `prepToPlay()` – this function is called before starting up a round of the game. Since games can be re-used (I.E. one Game object can be used to play it's game many times) this function resets variables to represent a new game. This function also returns a string with basic instructions for the player.
- `isValid(String move)` – this function takes a string the user entered and says if it's a valid move or not. For example, in number-based-games only Strings like "14", "-10", or "136752" are valid.
- `processMove(String move)` – this function is called after each valid move is input. It should process a move and update any variables. It can also optionally return a string to indicate changes to the game to the user. For example, in many games this method should check if a correct answer is given and update variables to mark the game as over.
- `isOver()` –this function queries the game-state variables and indicates if the game is over. In our example game this is a simple boolean variable, but you may find it useful to have multiple variables checked here (as some games have more than one way to end-the-game, such as a separate win-condition and lose-condition)
- `finalMessage()` – this function provides a final message to the player – often this indicates what the secret number or word might be.

The game's play function uses the above functions to play a game. **You should, at this point, download and read the provided Game.java file** In particular, make sure you review the "play" function and understand how it uses the above-functions together. You will also likely find it useful to read the provided "example" math test game to see how this plays out in practice. The provided math test game can also be run as a program to play a single ground of the game – you may want to try this out now too! **do not continue if you do not understand the game-structure Game.java defines!**

4.2 Math Test Game

A provided Math Test game can be found on canvas to help you understand how to program subclasses of the Game class, and how the behavior of a game is split into files. The constructor to this class takes "game settings" – a Random object to use when generating random numbers, minimum and maximum values for the randomly chosen numbers, and a char indicating which operator to test (plus, times, minus, or division) You will find this class useful at demonstrating how to implement a game using the Game class as structure. I recommend reviewing this code – you will not need to make any changes to it.

4.3 GameGrabber

The provided GameGrabber class operates as a menu system over Game objects. This class is provided complete, and you largely can ignore it's inner workings. You may find it useful to try out. As a final step of this lab you will be asked to modify the Main function to add your Games to the GameGrabber's menu.

5 Requirement: Number Guesser

You are required to make a game "NumberGuesser" The number guesser game is a traditional game for guessing numbers. The game is common enough that you may have even programmed it before! (This is one of Kluver's go-to games to program when learning a new programming language)

The game play starts by selecting a secret number. In the real world one person would pick the number and respond to guesses, we will have the computer do that. After that, players are allowed to guess numbers until they guess the secret number (or hit a maximum number of guesses). After each guess they are told "too high" or "too low" to indicate if the guess was above, or below the secret number.

An example playthrough can be seen below:

```
I've picked a number 1 to 100. You get 8 guesses to guess it
Enter Your Move or 'quit' to quit> 50
Too High
Enter Your Move or 'quit' to quit> 25
Too Low
Enter Your Move or 'quit' to quit> 37
Too Low
Enter Your Move or 'quit' to quit> 44
Too High
Enter Your Move or 'quit' to quit> 41
Too High
Enter Your Move or 'quit' to quit> 39
That's it!
The number was: 39
```

Formally, the NumberGuesser class must extend the Game class. It should have the following public constructor:

- `public NumberGuesser(Random rng, int maxNumber, int maxGuesses)` The parameters to this constructor are the Random object to use when generating the secret numbers, the maximum number to use when generating the secret number (inclusive) and the maximum number of guesses. The idea here is that, because a Game object should be re-usable, the settings we pass to the constructor best represent *difficulty settings*. By giving a larger or smaller maximum number, or changing the number of guesses allowed we can make an instance of the game that is more- or less- difficult.

You will need two *groups* of private variables for this class:

- Group 1: "game-setting" variables – these match the variables passed to the constructor above (a Random object, the maximum number, and the maximum guesses)
- Group 2: "each-round" variables – these describe the record-keeping needed by one single round of the number guesser game:
  - a variable to track the "Secret number"
  - a variable to track how many guesses have been used this round.
  - A variable to track if the last guess was correct or not.

You will also need to implement each of the abstract functions from the game class. Here is information to help implement those functions.

- The getName function should return "NumberGuesser"
- The prepToPlay function should pick a new secret number between 1 and the maxNumber (inclusive), as well as updating the guess-count variable and the "was the last guess correct" variable. It should return a message indicating the number range and number of guesses. Example: `I've picked a number 1 to 100. You get 8 guesses to guess it`

  Important notes

  - **Make sure you get the message correct** – while we can be flexible in manual grading – automatic grading will be looking for the exact correct message here (to the letter, symbol, and capitalization!)
  - **secret number generation** – to allow testing we've done some silly things with the Random object. To make your code work with our tester, you will need to use a line similar to: `hiddenNumber = rng.nextInt(maxNumber) + 1;` (Since nextInt includes 0 and does not include maxNumber – we need to add 1 to exclude 0 and include maxNumber in our possible random numbers)
- The game is over if the player has guessed the correct number (they won) OR if they have reached their maximum guess count (the lost).
- A move will only be valid if it encodes a **POSITIVE** number. You can look at the MathTestGame code for an example of how to check that (there is no "Easy" way to do this, you just gotta check what letters are in the string) You will, however, have to modify the code from MathTestGame, as that code accepts negative numbers, which are not valid in this case.
- Processing a move will involve first parsing the move string as a number. Thanks to the isValid method you can be sure it is a valid integer – so we don't need to

worry about errors there. To convert that string to an integer I recommend the `Integer.parseInt(String)` method. From there, you should update the turn-count and "was the last guess correct" variables, and return either "Too High" or "Too Low" to indicate if the guess was greater than, or less than the secret number. A correct guess should get the message "That's it!"

- The final message should indicate what the secret number was. For example:
  `The number was: 19`

5.1 Requirement: Hangman

"Hangman" is the rather morbid name for a traditional "word learning" game, commonly played in grade school. The game starts by selecting a word in secret (this would be done by a teacher in a school setting, but will be done by the Hangman class in our case) (Your code can use the provided WordList class to do this) The players are told how long this word is, often by showing a series of blanks:
`_______` (a 7 letter word)
The players then are asked to guess letters that may be in the word. If the user guesses a letter that is in the word, then all of the places in the word with that letter are revealed:
`_a___a_` (guess "a")
`_an__an` (guess "n")
`han__an` (guess "h")
If the user guesses wrong, then no new information is revealed:
`han__an` (guess "d")

Game continues in this manner until either 1) all of the letters are guessed, revealing the hidden word, or 2) a maximum limit of guesses is reached and the players lose. Rules differ around this "guess limit", while some people play where only wrong guesses count against the limit, our game will maintain a simple "total guess" limit – where correct and incorrect guesses both count against the limit.

Formally, the Hangman class must extend the Game class. It should have the following public constructor:

- `Hangman(WordsList words, int minWordLen, int maxWordLen, int maxGuesses)` The parameters to this constructor are a WordsList object which should be used to get words (this class is provided, and should be self-explanatory how to use), a minimum and maximum word length, and a maximum number of guesses. These last three integers serve to allow specifying a difficulty for the hangman game, and should change how each round of the game is played. (This might be deceptive, but typically smaller words are harder than longer words) The min and max word length are both inclusive.

As before, you will need two groups of variables:

- Group 1: "game-setting" variables – these match the variables passed to the constructor above (a WordList, min- and max-word lengths, and a maximum number of guesses)

- Group 2: "each-round" variables – these describe the record-keeping needed by one single round of the hangman game
  - A variable to track the "Secret word"
  - A variable to track how many guesses have been used this round.
  - A variable to track the "clue string" (I.E. `han__an`) in some way. You've got a few options here:
    * A string – which should start each game as repeated underscores (`_______`) and will need to be updated carefully as letters are revealed
    * A char array – same as above, this starts initially full of underscores and is updated as letters are revealed. You may find the array easier to manipulate.
    * A boolean array – similar to the above – but instead of storing letters – it can just store true/false to indicate if a letter is revealed or not.
    * Other options may exist – **You should think through how to store the "clue string" before you get started!**

You will also need to implement each of the abstract functions from the game class. Here is information to help implement those functions.

- The getName function should return "Hangman"
- The prepToPlay function should pick a new word based on the input parameters, reset the guess count variable, and reset/recreate your "clue string" variable (for example – if you are using the String approach – you will need to make an appropriate length string full of underscores.) the function should return a message indicating the length of the word, and the number of guesses example:
  `I've picked a 4 letter word. Guess letters you think are in the word. You get 20 guesses` **As before – be very careful with your message-text in these classes** (you may find it best to copy/paste from the tester code here.)
- The game is over if the player has guessed every letter in the hidden word OR if they have reached their maximum move limit. Testing if the player has guessed every letter in the hidden word will require slightly different logic depending on how you've defined that variable – but should in general involve looping over your "clue string" variable and checking for underscores.
- A move is valid only if it is a single letter long. Optionally, you can limit guesses to only alphabet letters (so you can choose "7" to be an invalid guess) but we do not require this.
- Processing a move will involve first recording that another guess has been made, secondly updating your "clue string" variable (replacing appropriate underscores with letters where the user's guessed letter appears) and finally, generating and returning the new "clue string" (I.E. the mix of letters and underscores "_")

- The final message should indicate what the secret word was. For example:
  `The word was: hangman`

6 Requirement: GameGrabber

The GameGrabber class has been provided on canvas. If you've made your Hangman and NumberGuesser games correctly you won't need to modify it at all. (and to be clear – we expect you to not modify any of the non-static code for this class) You will, however, need to update the Game array in the `public static void main` method to include your two new games. You can choose whichever difficulty settings you feel are most appropriate, but you are **required** to add instances of your new Game objects to the Game array in the main method. **This will not be automatically tested** – if you want to test this manually. You will know it's done right when you can run the main method on GameGrabber and play the 4 provided math tests or your two new games.

7 Submission

For this lab you need to turn in:

- `Hangman.java`
- `NumberGuesser.java`
- `GameGrabber.java`

7.1 Testing

Testing this code (in an automatic way) is harder than it appears. This is a general problem with user-interface code and not one we found a great solution for this semester. Therefore this lab will largely be tested using fake-random objects (which allow us to force specific results for the random number in number guesser or word in hangman) and then using a pre-scripted series of inputs. These "pre-scripted" games will play out, showing you only what your code output. For manual testing you should match your code's output against the provided output letter-to-letter.

While we understand that this type of testing does a poor job of isolating specific functions and helping you identify low-level issues, it's the best we can provide for the Game class design. As such, the automatic grading will be weighted less on this lab than other labs.

Grading on this assignment will follow the given general rubric:

- 18 points autograder – This is split somewhat chaotically across the methods.
- 10 points code style:

  - Your name (and the name of any collaborators) should be in a comment at the top of the file
  - Correctly formatted JavaDoc (this is the `/** ... */` comment blocks placed before each function that describes the function
    * We will consider this optional for constructors and 1-line functions.
    * This requirement stands for any function longer than 1 line.
  - You should also have a JavaDoc before each class that describes the class as a whole. You can look at the provided code from lab06 to see examples of this
  - Good variable names
  - Consistent tabbing (while java does not require this the same way python did – we **do** require it. Reading and grading poorly tabbed code is a painful experience)
  - Outside of these explicit requirements, as normal, we will generally be looking for (and giving feedback on) anything that makes the code harder/more painful to read, and will take away points if it's particularly bad.

- 33 points Number Guesser class manual review – note we will be looking for more than direct function requirements, good use of modifiers like public/private will also be checked here.

- 33 points Hangman Class manual review – note we will be looking for more than direct function requirements, good use of modifiers like public/private will also be checked here.

- 6 points Game Grabber Class – We will only be looking for appropriate updates to the main method