COSC102 Spring 2015

Guessing Games Homework 1

Due date: **Monday February 2nd, at 11:55 p.m. on Moodle**

For this assignment, you implement a guessing-game in Java. The game gives users two options: in one the computer picks a number that the user subsequently tries to guess, in the other the user picks and the computer guesses.

A solution for the first option is explained in the online Javanotes textbook, the second is its "reverse", or dual. If you use code that isn't your own, such as more than a couple of lines from the textbook just mentioned, you have to cite your sources. I expect you to give **code credit** for any help you received. Doing so means in the comment header of the file or in a `readme` file explicitly list anything/anyone that helped you: an URL, a person or a textbook (you don't have to cite Java API, your classnotes and the class textbook). Within the file insert comments at the location(s) the source(s) came into play.

Starting Menu

The program starts with a menu on the output console asking the user to play by selecting an option or to exit the program. The menu items can be

- User guesses
- Computer guesses
- Quit

(Feel free to add simple ASCII to make the starting menu like a fun splash screen.)

If the user chooses to play, the chosen option starts. When the play is complete the game returns to the starting menu allowing the user to play again or exit.

When the User Guesses

Play starts with the user picking a range: two integers are entered in any order, one the lowest number in the range, the other the highest.

The computer then secretly picks a random integer from the range. Convince yourself that the following line of code picks any integer in the range, including the end points, with equal probability.

```
int pick = ( (int) ( Math.random() * (high - low + 1) )) + low;
```

The computer then enters the following loop.

1. Get and check the user's guess, which must be an integer in the range.
2. If the guess is correct, play is finished; the number of guesses required printed and the starting menu displayed.
3. If the guess is incorrect, tell the user if it is too high or too low.
4. Continue with step 1.

When the Computer Guesses

Play starts with the computer asking the user for a range, after which the user secretly thinks of a number in that range.

The computer then enters the following loop.

1. The computer guesses an integer in the range.
2. If the guess is correct, play is finished; the number of guesses required, printed and the starting menu displayed.
3. If the guess is incorrect, the user tells the computer if the guess is too high or too low.
4. Continue with step 1.

How should the computer generate its guesses? Starting with the low end of the interval and incrementing one by one is not efficient. It is lucky to hit the number in the first few tries. If the interval has n numbers, in the worst case n guesses could be necessary.

Instead the binary-search pattern is efficient since by proposing the middle number each time half of the possibilities are thrown away based on the user feedback. This guarantees no more than approximately $\log(n)$ guesses are needed—log() is $\log_2()$, the base-2 logarithm.

So, for an interval of 100 numbers, the linear search might take 100 guesses but, the binary search should never take more than 8 guesses. We will use this type of analysis later in the course when we talk about "efficiency", which usually mean the worst-case amount of running time and extra memory the procedure uses.

Helper Functions

To get keyboard input from the user a `Scanner` object is used, thanks to the import statement

```
import java.util.Scanner;
```

As the `Scanner` is a resource that uses memory, it is a good practice to use a single `Scanner` object by input stream. Thus the declaration of a global constant belonging to the class works well.

```
static final Scanner keyboard = new Scanner(System.in);
```

Write helper methods incrementally. Design helper methods that are useful in both options. Find methods in the `Scanner` and `Math` classes that are useful to make the program resistant to bad and invalid input (i.e. data type and range).

Grading

Your submission will be graded both on correctness [80%] and style [20%]. Therefore, test your program thoroughly and ensure that your code is readable and well formatted (e.g., follow textbook conventions included in Appendix A). Don't be grossly inefficient, but there's no need to over-optimize.

Submission

Include a header to your source code file that contains

- your student information,
- a short description of the program,

- if you have not fully implemented the assignment a description of each part missing and
- acknowledgement crediting people and resource(s) that helped you.

Using the upload link on Moodle submit **a `zip` file named `hw1`** that contains your **Java source code** file. The Java source code is the text file that has a `.java` extension (although this extension may not be visible on all computers). Do not submit the `.class` (compiled bytecode) file or `.java~` (backup) one: they are not useful to grading as they can't be read.

Your submitted `.java` file should contain a `main` function that, when invoked, begins the program as described above. You are expected to use helper functions and methods to structure your code and to include comments when required.

Submit early and often. Late assignments are not accepted, so make sure you have some version of the source code uploaded to Moodle early on. You can replace the uploaded file with a more up-to-date version until the deadline.

Tips

Plan out your program *before* you start coding: "Minutes of thinking will save you hours of coding." Consider writing an outline of your program using a combination of comments and function declarations for the different parts of the assignment as you see them. Then fill in the parts of code as you go. **Compile often**. If you submit a program that does not compile, then there are bugs and the program cannot be tested. A program that does not compile automatically receive a low grade.

I strongly suggest you to reproduce the process followed in lecture, which is

- to work on your code in small sections,
- to compile as you are working on each section to make sure that all the syntax errors are removed as soon as possible, and
- to test each small section of code before moving on.

A section can be two or three lines of code. Following this process prevents a massive debugging exercise to finish what you think is a full working program.

Design your program incrementally, consider the smallest tasks you have to do (printing messages, getting input, picking a random number, etc.) write and test each. Once they are right, you can put them together to form your whole program.

You can add print statements for the purposes of debugging to check the values of integers in your code. Comment these out before your final submission.

More Practice

Looking for more practice... For extra credit consider the Hangperson game you may have written in Python. (re)Writting Hangperson in Java

- require to use arrays of `String` and
- might be an opportunity to work/improve code design.