# CSE 142, Autumn 2008
# Programming Assignment #5: Guessing Game (20 points)

**Part A (single game) due Tuesday, October 28, 2008, 11:30 PM ***
**Part B (complete program) due Tuesday, November 4, 2008, 11:30 PM**

This assignment focuses on `while` loops and random numbers. Turn in a file named `GuessingGame.java`.

Your program allows the user to play a game in which the program thinks of a random integer and accepts guesses from the user until the user guesses the number correctly. After each incorrect guess, you will tell the user whether the correct answer is higher or lower. Your program must exactly reproduce the format and behavior of the logs in this document.

This assignment will be due in two parts: an initial "Part A" that plays only a single game, and the second complete (multi-game) "Part B" a few days later. Part A will not be worth as many points as Part B. * **NOTE: Part A is <u>not</u> accepted late**, nor can you earn any "early days" for submitting it early.

## Program Behavior (Part A):

In Part A, a single guessing game is played. Several features that will be present in Part B, such as a prompt to play more games and a final display of overall statistics, are not included in Part A.

In the guessing game, the computer chooses a random number between 1 and 100 inclusive. The game asks the user for guesses until the correct number is guessed. After each incorrect guess, the program gives a clue to the user about whether the correct number is higher or lower than the user's guess. Once the user types the correct number, the game ends and the program reports how many guesses were needed.

```
I'm thinking of a number between 1 and 100...
(The answer is 46)
Your guess? 50
It's lower.
Your guess? 25
It's higher.
Your guess? 37
It's higher.
Your guess? 43
It's higher.
Your guess? 47
It's lower.
Your guess? 46
You got it right in 6 guesses!
```

Your program shouldn't always use 46 as the correct answer; you should choose a different random answer between 1 and 100 (inclusive) each time the program is run. Your output will have different random numbers depending on the random number chosen and depending what the user types, but your output's overall structure should match the output shown.

In Part A, the program should print a message that shows the game's correct answer (`(The answer is 48)` below). Obviously this ruins the challenge of the game, but the point of Part A is for your own testing to get the game logic working. By being able to see the correct answer, you can more easily debug the program by trying various guesses and seeing whether your game prints the correct clue. A message like this is sometimes called a "debug message" or "debug `println`". You will remove this debug message in Part B.

You should handle the case where the user guesses the correct number on the first try. Print the following message:

```
I'm thinking of a number between 1 and 100...
(The answer is 71)
Your guess? 71
You got it right in 1 guess!
```

Assume valid user input. When prompted for numbers, the user will type integers only, and integers in suitable ranges.

Part A will be graded on external correctness (correct output) only, not on internal correctness or any aspects of your coding style. You do not need to comment Part A, or worry about redundancy, proper static methods, etc.

## Program Behavior (Part B):

Part B is a more sophisticated version of the guessing game program that is able to play multiple games, as well as printing overall statistics about all games played. Unlike with Part A, it is possible to submit Part B late and also to earn early days for submitting Part B early. The differences between Part A and Part B are summarized below.

```
<< your haiku intro message here >>

I'm thinking of a number between 1 and 100...
Your guess? 50
It's lower.
Your guess? 25
It's higher.
Your guess? 35
It's lower.
Your guess? 30
It's higher.
Your guess? 32
It's lower.
Your guess? 31
You got it right in 6 guesses!
Do you want to play again? y

I'm thinking of a number between 1 and 100...
Your guess? 50
It's higher.
Your guess? 75
It's lower.
Your guess? 65
It's lower.
Your guess? 61
It's higher.
Your guess? 64
You got it right in 5 guesses!
Do you want to play again? YES

I'm thinking of a number between 1 and 100...
Your guess? 60
It's lower.
Your guess? 20
It's higher.
Your guess? 30
It's higher.
Your guess? 40
It's higher.
Your guess? 45
It's higher.
Your guess? 50
It's lower.
Your guess? 47
It's higher.
Your guess? 48
It's higher.
Your guess? 49
You got it right in 9 guesses!
Do you want to play again? no

Overall results:
total games   = 3
total guesses = 20
guesses/game  = 6.67
best game     = 5
```

First, the program prints a header message describing itself. This can be any message of your choosing, but we suggest that you write a haiku poem related to the guessing game. Your poem can be posted to Facebook.

Next, a series of games is played. Each game behaves identically to the game you wrote in Part A, except that the correct answer is not told to the user at the start of the game; the debug message from Part A is removed. You can leave the debug message in your code while working on Part B, but remove it before turning it in. (Consider commenting out the debug message so that you can turn it back on/off if you need it for debugging.)

After each game ends and the number of guesses is shown, the program asks the user if he/she would like to play again. Assume that the user will type a one-word `String` as the response to this question.

A new game should begin if the user's response starts with a lower- or upper-case Y. For example, answers such as "y", "Y", "yes", "YES", "Yes", or "yeehaw" all indicate that the user wants to play again.

Any other response means that the user does not want to play again. For example, responses of "no", "No", "okay", "0", "certainly", and "hello" are all assumed to mean no.

Once the user chooses not to play again, the program prints overall statistics about all games. The total number of games, total guesses made in all games, average number of guesses per game (as a real number rounded to the nearest hundredth), and best game (fewest guesses needed to solve any one game) are displayed.

Your statistics should present correct information for any number of games ≥ 1, and any number of guesses ≥ 1 in each game. You may assume that no game will require one million or more guesses, but beyond that, your code should work no matter how many games the user plays or how many guesses are needed to solve each game, even if this number is very large.

Part B will be graded on both external and internal correctness, according to guidelines specified on the next page. Your Part B code is required to have a particular class constant described on the next page.

NOTE: If you finish all of Part B before Part A is due, you may submit a working version of Part B as your Part A solution. But you must still submit a file into each of the Part A and Part B areas on the web site (even if it is the same file in both cases). The same due dates still apply for both Parts regardless of whether you choose this option.

## Implementation Guidelines:

```
<< your haiku intro message here >>

I'm thinking of a number between 1 and 5...
Your guess? 2
It's higher.
Your guess? 4
It's lower.
Your guess? 3
You got it right in 3 guesses!
Do you want to play again? yes

I'm thinking of a number between 1 and 5...
Your guess? 3
It's higher.
Your guess? 5
You got it right in 2 guesses!
Do you want to play again? Nah

Overall results:
total games   = 2
total guesses = 5
guesses/game  = 2.50
best game     = 2
```

In Part B, you must define a **class constant** for the maximum number used in the guessing game. The sample log on the previous page shows the user making guesses from 1 to 100, but you should be able to change just the value of the constant to cause the program play the game with other ranges, such as a range of 1 to 50, a range of 1 to 250, or any range from 1 to any maximum.

Use your constant throughout your code and do not refer to the number 100 directly. Test your program by changing your constant and running the program again to make sure that everything works right with the new value. For example, a guessing game for numbers between 1 and 5 would produce output such as that shown at left. The web site shows other expected output cases.

As with the Space Needle assignment, we strongly suggest that you add the constant to your code last. You may also want to re-enable your debug message so that you can see the correct answer when testing your constant and code.

Read the answer using the `Scanner`'s `next` method (not `nextLine`, which can cause strange errors when mixed with `nextInt`). To check for a yes/no user response, use `String` methods described in Chapters 3-4 of the book. If you get an `InputMismatchException` error, it means you are trying to read the wrong type of value from a `Scanner`. For example, you may be trying to read an integer when the user has instead typed a string.

Produce randomness using a single `Random` object, as described in Chapter 5. Remember to `import java.util.*;`

Produce repetition using `while` or `do/while` loops. You may also want to review fencepost loops from Chapter 4 and sentinel loops from Chapter 5. Chapter 5's case study is a particularly relevant example for this assignment. Some students try to achieve repetition without properly using `while` loops, by writing a method that calls itself; this is not appropriate on this assignment and will result in a deduction in points.

## Stylistic Guidelines (Part B):

For this assignment you are limited to the language features in Chapters 1-5 shown in lecture or the textbook.

Structure your solution using static methods that accept parameters and return values where appropriate. For full credit, you must have at least **the following two methods other than `main`** in your program:

1. a method to **play one game** with the user (not multiple games)
   This method should *not* contain the code that prompts the user Yes/No to play another game.
2. a method to **report the overall statistics** to the user (and nothing more)
   This method should *only* print the statistics, not do anything else such as `while` loops or playing games.

You may define more methods than this if you find it helpful, although you will find that the limitation that methods can return only one value will tend to limit how much you can decompose this problem.

You may define other methods if they are useful for structure or to eliminate redundancy. Unlike in some past programs, it is okay to have some `println` statements in `main`, as long as your program has good structure and `main` is still a concise summary of the program. For example, you can place the loop that performs multiple games and the prompt to play again in `main`. As a reference, our solution has 4 methods other than `main` and occupies between 90-110 lines total.

Use whitespace and indentation properly. Limit lines to 100 characters. Give meaningful names to methods and variables, and follow Java's naming standards. Localize variables whenever possible. Include a comment at the beginning of your program with basic description information and a comment at the start of each method. Since this program has longer methods than past programs, also put brief comments inside the methods explaining relevant sections of your code.