Software Development

Higher Diploma in Science in Computing

Project-Based-Learning Project Report

Report Authors in Alphabetical Order:

Liz Bourke/19149310

Alexia Falcoz/19144733

Laurine Rolland/19181965

Contents

[Coding Responsibilities 3](#_Toc26125305)

[Input, Processing, and Output 4](#_Toc26125306)

[Class Diagram 5](#_Toc26125307)

[Decisions Made 6](#_Toc26125308)

# Coding Responsibilities

Every member of the project team undertook to code the entire project prior to discussing implementable solutions, to ensure that everyone understood the principles at work. Based on the group discussions, we decided to divide the coding responsibilities for the final product in the following manner:

**Alexia** took primary responsibility for coding solutions to store the game history, for implementing the arrays, and for checking that all numbers generated by the Random Number Generator (the lottery numbers) and all numbers input by the user are unique values.

**Laurine** took primary responsibility for coding the instantiable class, its setters and getters, and the function to calculate winnings.

**Liz** took primary responsibility for coding repetition and selection statements inside the application class.

# Input, Processing, and Output

## Input

There are two main inputs provided by the user:

1. The number of lines (between one (1) and three (3)) that the user plays in a given game, and
2. The six (6) numeric values between one (1) and forty (40) which the user provides for each individual line.[[1]](#footnote-1)

There is one main input that must be generated at the beginning of each iteration of the game:

1. The six (6) random numeric values between one (1) and forty (40) against which the user-generated values must be checked.

There is one additional input at the end of each iteration of the game, provided the victory condition has not been reached:

1. The user inputs a value which will allow them to either continue or exit.

## Processing

There are five (5) main pieces of processing to be accomplished:

1. Six random numbers between 1 and 40 must be generated.
2. For each line, the user-generated numeric values must be checked against the random numbers, and the number of matches recorded.
3. For each line, the number of matches recorded must be checked against the number of matches needed to win a sum of money (3, 4, and 5 matches) or the whole lottery (6 matches).
4. For each game, the sum of money won must be calculated and recorded.
5. For the total of all games, the total sum of money won must be calculated.
6. For the total of all games, the average sum of money won per game must be calculated.

## Output

There are 6 main outputs:

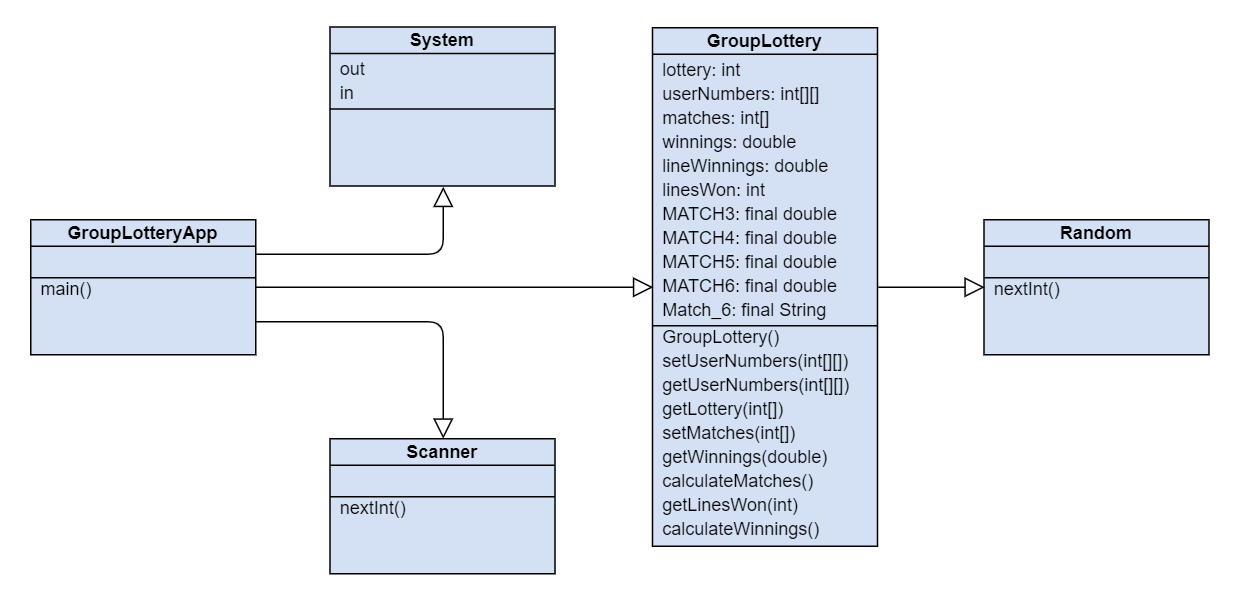
For each game, the following must be output to the user:

1. the total number of matched numeric values per line played;
2. the number of lines played in the game;
3. the lines where the user won a sum of money. (That is, the lines where the user matched between 3 and 6 numbers.)

At the end of all the games, the following must be output to the user:

1. the total winnings. (The sum of all sums won in each game.);
2. the total number of games played;
3. the average winnings per game.

# Class Diagram



# Decisions Made

Decision 1: We chose to store the number of lines per game and the numeric values input by the user for each line as a 2D array, in order to simplify processing. The other option we considered was to:

1. use a variable to count the number of lines
2. take the numeric values as a 1D array
3. calculate the number of matches in that line
4. store the number of matches at index[i][linecounter] of a 2D ArrayList, where the outer ArrayList index[i] was equal to the number of the current game.[[2]](#footnote-2)
5. increment the linecounter variable
6. take the numeric values again
7. repeat until there are no more lines to calculate matches for.

But this approach meant that we were running three calculations to find matches-per-line every game. (Also, clarification of the project specification indicated that use of ArrayLists was deprecated.) Using a 2D array, we could run a (larger) calculation just once.

Decision 2: We chose to limit the array that holds game history to a theoretical maximum of 100 games, on the basis that this lottery game as a whole is not fascinating enough that anyone would play it more times than that in a single session.

Decision 3: We decided that the lottery jackpot would be a total of 1,000,000 euro. Anyone who "wins the lottery" (matches 6 numbers in a line) will receive 1,000,000 euro.

Decision 4: We decided that winning the lottery would automatically end the game loop and trigger the display of games history. While the random numbers are generated afresh for every iteration of the game, and therefore users could (theoretically) continue playing and winning the lottery for as long as they wanted, we decided that once the victory condition was met, it should trigger exit from the game. We made this decision for the following reasons:

* we chose to limit the array that holds game history to a maximum of 100 games, and thus we choose to discourage long games;
* the lottery game is not fascinating except to compulsive gamblers, so winning once should provide adequate player satisfaction.

Decision 5: After we had mostly coded the game, we decided to implement the calculation of the average game winnings in the instantiable class, rather than in the application class, to allow for greater reusability of code.

# Testing

We tested the following conditions:

Against input for number of lines:

1. inputting a negative number (no problem)
2. inputting zero or a number greater than 3 (no problem)
3. inputting a fraction or a String (returns runtime exception, because without an exception handler the Scanner can only handle integers)

Against input for user-generated numbers:

1. inputting a negative number (no problem)
2. inputting zero or a number greater than 40 (no problem)
3. inputting the same number more than once (no problem)
4. inputting a fraction or a String (returns runtime exception, because without an exception handler the Scanner can only handle integers)

Setting the Random Number Generator to generate numbers between 1 and 6 only to check if uniqueness is being verified and matches are actually being calculated:

1. Problems arose with verification of uniqueness. Uniqueness not achieved.

First solution attempted for validation of uniqueness resulted in "main" program not progressing into main game do-while loop. (See solution below. )

public GroupLottery(){

int[] array1 = new int[6];

for(int i=0; i<6; i=i+1){

Random myRandom = new Random();

int ranNum = myRandom.nextInt(6)+1;

array1[i] = ranNum;

}

int[] lottery = {0,0,0,0,0,0};

int next\_space = 0;

boolean duplicate = false;

while(next\_space<6){

int number\_picked = array1[next\_space];

for(int i = 0; i<lottery.length; i++){

if(number\_picked == lottery[i]){

duplicate = true;

break;

}

else{

duplicate = false;

}

}

if(duplicate = false){

lottery[next\_space] = number\_picked;

next\_space = next\_space +1;

}

}

}

**The while loop does not complete.**

**The second attempted solution:**

public GroupLottery(){

lottery = new int[6];

int[] compare = new int[3];

int[] compare2 = new int[3];

for(int i=0; i<6; i=i+1){

Random myRandom = new Random();

int ranNum = myRandom.nextInt(6)+1;

lottery[i] = ranNum;

for(int j =1; j<3; j++){

lottery[lottery.length-j] = compare2[j];

lottery[j-1] = compare[j];

}

}

boolean duplicate = true;

while (duplicate ==true){

for(int i =0; i<3; i=i+1){

if(compare[i] != compare2[i]){

duplicate = false;

}

else if(compare[i] == compare2[i]){

duplicate = true;

int k;

k = i;

Random newRan = new Random();

int ranNew = newRan.nextInt(6)+1;

compare[k] = ranNew;

}

}

}

lottery[0] = compare[0];

lottery[1] = compare[1];

lottery[2] = compare[2];

lottery[3] = compare2[0];

lottery[4] = compare2[1];

lottery[5] = compare[2];

}

This again failed to validate uniqueness, but at least did not cause a while loop to fail to complete.

The final solution to validate uniqueness is the solution in the file. Testing this (which proved to validate uniqueness) showed that we had made errors in the match calculation. The initial match calculation only registered matches if they were at the same position in the array. This proved to be an iteration error, and easily solved once recognised.

1. We chose to store this as a 2D array: see "Decisions." [↑](#footnote-ref-1)
2. This approach would mean we could have a theoretically infinite number of games. [↑](#footnote-ref-2)