SYSC2004 Fall 2018

Assignment 1

**Objectives:**

- Write one or more simple classes, to practice coding constructors, accessors and mutators.
- Write a client of these simple classes, to practice making objects and calling methods on those objects.
- Learn about random number generation.
- Single and double dimension arrays
- Use test-driven development to guide the writing of the simple classes.
- Defensive Copies

Provided Files: DiceTest.java, TicTacToeTest.java
Submit Files: Dice.java, DiceClient.java, TicTacToeEnum.java, TicTacToe.java

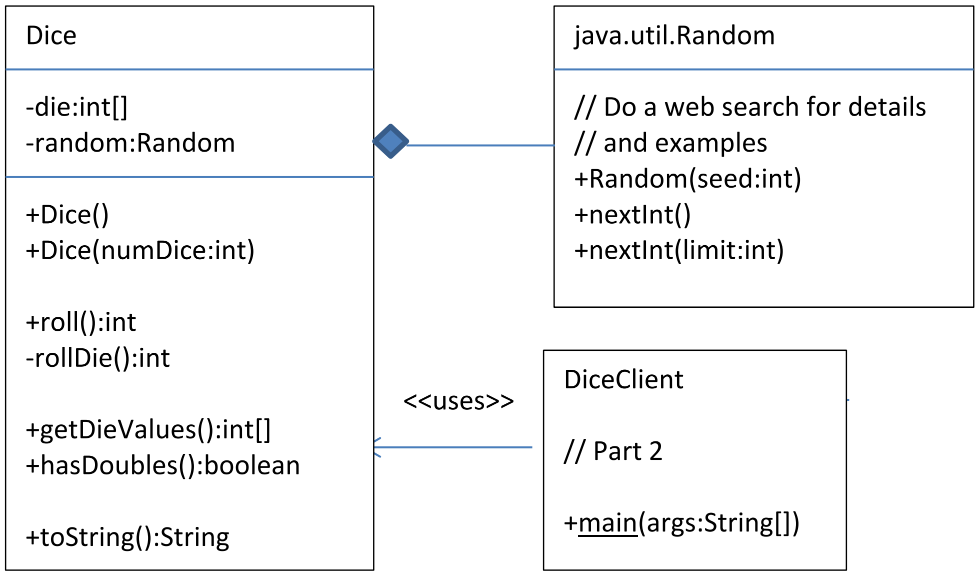
- If you do not complete the whole assignment, please submit the portion you have completed, submitting empty files for the incomplete portions.  You will receive part-marks.

This assignment to be too large to be finished in one or even two nights of programming. The intent is to prompt you to practice a lot early on in this course. Without practice, you will not master the beginning concepts and will be unable to keep up as the topics become more complex. Later assignments will not be this large.  **JavaDoc commenting is required – write as you go, to avoid lots of work at the end.**

## Part 1: Dice

In Assignment 2, we are going to need a pair of dice to play a game.  We will build that `Dice` class in this first assignment as a learning exercise in random number generation – the utility by which programs generate random numbers for use in simulations and testing.

The UML class diagram below must be followed exactly for full marks. By doing so, you will learn good programming practices.



The UML class diagram shows that the `Dice` class has-a `Random` object as an instance variable as well as an integer array of die. The default constructor will construct a default pair of die whereas the one-argument constructor will allow the construction of any number of dice (e.g. perhaps we want to play Yahtzee which uses five dice). The principle operation is the `roll`() method which will roll all of the dice at once, returning their total value. The implementation of this `roll`() method must use of (i.e. call) a private method `rollDie`() which rolls a single die and ensures that this rolled value is between 1 and 6, inclusive. The class has two getter-type methods: (1) `getDieValues`() returns an array of all the individual die-values and (2) `hasDoubles`() returns true if there are any double-values amongst the die-values. Finally, as you will see in this course, every good class has a `toString`() method. The format required for the `String` returned by this method is as follows: a space-separated list of all the individual die-values. Example: For two dice, the `toString`() method should return "4 6 ".

Special implementation requirements:

1. The one-argument constructor must ensure that at least one die is constructed. Place the following code at the beginning of this constructor to enforce this requirement. We will learn about exceptions after the midterm; for now, just copy.

   ```
   if (numDice < 1)
   throw new IllegalArgumentException ("Put a nice msg”);
   ```

2. Below is some sample code for constructing a random object (Hint: Also put this in your one-argument constructor).

   ```
   this.random = new Random(new Date().hashCode());
   ```

   This code sample calls the one-argument `Random` constructor that takes in a seed value which is used to initialize the underlying math that ultimately creates your random sequences of numbers. Oftentimes, the current time is used as the initial seed value because the current time will always be different each time that the program runs, in turn ensuring that the sequence of numbers generated are different each time. The current time is found with this code: `new Date()`

3. When rolling a die, remember that the values should be from 1 to 6, inclusive.
4. The `getDieValues`() method must return a defensive-copy of the die-values. See your notes about defensive-copies (Lecture 5)

In this first part of the assignment, you are to write the implementation code for **Dice.java** as described and then test your implementation of the `Dice` class with the provided JUnit test class `DiceTest`.

- You are writing the service side. I have written client side already, as a test class. This is called Test-Driven Development.
- If you have questions and can't understand the preceding problem description, you can find clues by reading the test class, to see how the class will be used and what outputs are expected.
- Tip: Remember the small **How-To load a JUnit test** that is posted on CULearn.

- Tip: Other than perhaps temporarily commenting out portions of the test code to allow you to build your programs incrementally, never fix a bug by changing the test code. The bug will be in your source code.

## Part 2: Dice Client

Once you have a fully-tested bug-free version of your `Dice` class (all tests pass GREEN), you will write your own client program that uses the `Dice` class. This program will be an exercise in "using" a class but will also teach you more about random number generation, namely that it is not perfectly random.

Refer now back to the original UML class diagram. The `DiceClient` class has a single method called `main`().

```
public class DiceClient {

     public static void main (String[] args) {

           // You write ALL your client code here,
           // including all variables
     }

}
```

Your code must generate a statistical sample set of 2000 dice-rolls from which it will print out a histogram of the 4000 values (2000 rolls of a pair of dice = 4000 die values) as well as their average and standard deviation. Check out this URL for calculating the standard deviation: https://www.strchr.com/standard_deviation_in_one_pass

Sample output that you are to replicate in your program is shown below, to help you understand the task at hand.

```
The average roll was 3.48475
The standard deviation of the rolls was 1.721704805563369
The histogram of the rolls is:
1(697) :*********************************************************************
2(660) :******************************************************************
3(650) :*****************************************************************
4(671) :*******************************************************************
5(644) :****************************************************************
6(678) :*******************************************************************
```

In the histogram, there are 6 lines, one line for each possible value. On each line, the value of the die is printed, followed by the total number of rolls of this value in brackets, followed by a "visual representation" of this total. A * should be printed **for every 10 rolls**.

## Part 3: TicTacToe

(Create a new Java project)

You will implement a simple TicTacToe game (also called x's and o's). You are given the UML diagram for the code to be written.

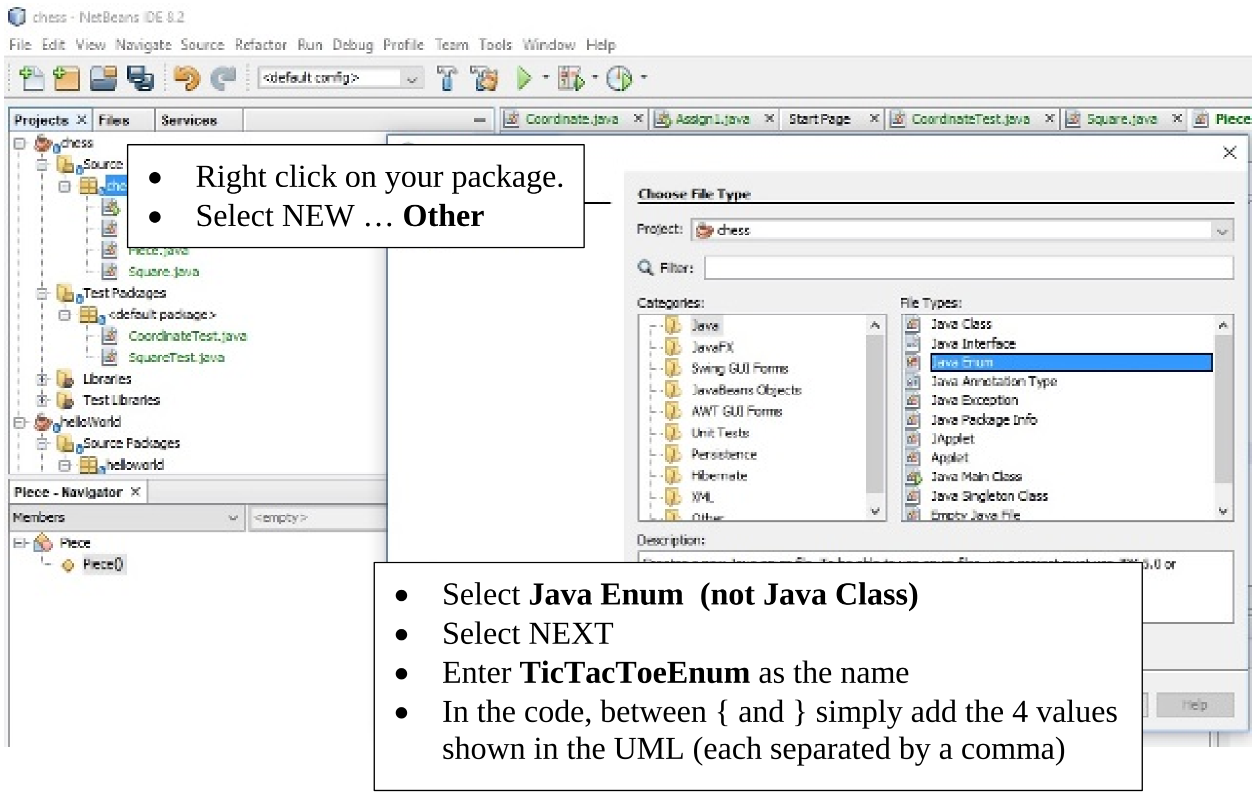
1. You shall write the code as described below and test it using the provide JUnit test TicTacToeTest.java.
2. After all unit tests have successfully ran with your code, you will write some client code that will actually let you play a game, using prompts and responses from the console.

```
TicTacToe
------------------------------------------------------------------
-nRows:int         // Default to 3
-nColumns:int    //  Default to 3
-numToWin:int  //  Default to 3 (i.e. Need 3 in a row to win
-grid[][]:char      // Either space ' ' or 'x' or 'o'
-turn:char;        // Alternate between 'x' and 'o'
-gameState:TicTacToeEnum  // Iniitally IN_PROGRESS
-nMarks:int      // Initial to 0; increment after each turn;
                      // use to detect DRAW (when grid is full but no winner)
------------------------------------------------------------------
+TicTacToe(initalTurn:char)   // Use defaults
+TicTacToe(nRows:int, nColumns:int, numToWin:int initialTurn:char)
+reset(initialTurn:char)     // Sets all grid locations to space ' ' and
      // resets other variables to their initial values

+getTurn():char
+getGameState():TicTacToeEnum

-charToEnum(player:char):TicTacToeEnum

+takeTurn(row:int, column:int):TicTacToeEnum
-findWinner():TicTacToeEnum

+toString():String
```

```
<<enumeration>>

TicTacToeEnum
------------------
IN_PROGRESS
X_WON
O_WON
DRAW
```

**Implementation Notes:**

1. Write the `TicTacToeEnum` class first. It is super simple and you need it to make the rest of your code compile.



- Right click on your package.
- Select NEW … **Other**

- Select **Java Enum (not Java Class)**
- Select NEXT
- Enter **TicTacToeEnum** as the name
- In the code, between { and } simply add the 4 values shown in the UML (each separated by a comma)

2. Two constructors are provided. Although most people play TicTacToe with a 3x3 grid (and you need 3-in-a-row to win, you could also play this game with a larger grid (e.g. 8x8).
   - The four-argument constructor should verify that the input parameters are not negative values. Hint: Refer back to your `Dice` class for code that will check for these IllegalArguments
   - As part of the construction of the game, these constructors probably should “reset” the board. So you probably want to write the code for `reset`() at this time, as well.

3. The method `charToEnum`() is a convenience utility that converts a player (either ‘x’ or ‘o’) into the corresponding value in the ENUM (either X_WON or O_WON). You will see its usefulness when you write the `findWinner`() method below.

4. The `toString()` method shall return a string representation of the TicTacToe board that looks like the following:

   When it is empty:

   ```
     |   |   |
     |   |   |
     |   |   |
   ```

   After some turns have been taken:

   ```
   X | O |   |
   X | O |   |
     |   |   |
   ```

   One each line:

   character-space-bar-space-
   character-space-bar-space-
   character-space-bar-space

5. The purpose of the `takeTurn()` method is to (1) put the appropriate character (i.e. 'x' or 'o') in the requested grid location (row, column) and (2) try to `findWinner()`
   - Notice that the `findWinner()` method is private. It is not meant to be called by a client; instead, it is an internal method used by the class as a utility. It is meant to help you organize your code to make it more readable. This method will be your longest method because it will contain your logic for determining whether the current player has won by filling a complete horizontal row or a complete vertical column.
     - **To reduce your workload, you do not have to find diagonals. Interested students are encouraged to do so, but it is not required by the marking scheme.**

After your code has successfully passed all the tests in the provided JUnit test code (TicTacToeTest.java), you have one final task to complete. Copy-paste the following code inside your `TicTacToe` class. **Right-click on TicTacToe and select RUN.** Have some fun by playing your game.

```
public static void main(String args[]) {
     TicTacToe game = new TicTacToe('X');
     Scanner scanner = new Scanner(System.in);

     do {
         System.out.println(game.toString());
         System.out.println(game.getTurn() +
             ": Where do you want to mark? Enter row column");
         int row = scanner.nextInt();
         int column = scanner.nextInt();
         scanner.nextLine();
         game.takeTurn(row, column);

     } while (game.getGameState() == TicTacToeEnum.IN_PROGRESS);
     System.out.println( game.getGameState());

 }
```

## Marking Scheme (Total : 68 = 17*4)

Assignments will not be marked if standard indentation is not used. A mark of zero will be given.

Instruction to TAs: Make sure your write your initials on each assignment that you mark, so that the student knows who to contact for concerns about their mark.

Instructions to Students: Afterwards, if you have a question about the marking of your assignment -

1. On the course webpage, study the schedule for the TAs and see when your TA is on duty in the lab. If you do not have a class/lab conflict with that time, visit the TA during their lab period.
2. Alternatively, email the TA that marked your assignment. Pose your question if it is simple; if not, request a meeting. If you do not receive a reply from the TA within 2 working days, re-send your email and now 'cc your instructor so that s/he can follow up.

Marking Values: Each criterion is worth 4 marks and will be marked according to the rubric levels of BDAE (0 is not present at all, or consisting only of the UML call signatures)

Beginning for 1 mark: A bit but mostly not.

Developing for 2 marks: Some but missing key points and/or ignoring requirements & conventions

- Maximum mark possible if the code does not work.

Accomplished for 3 marks: Mostly of the vital learning points demonstrate mastery

Exemplary for 4 marks: Professional level code that is even better than asked.

Part 1 (Unit test must pass for any part, for full marks)

/4 Dice constructors initialize all instance variables (& pass unit tests)

- Zero marks if ANY instance variable is initialized as part of the declaration.
- Exemplary – ONLY if constructor chaining is used; otherwise limited to Accomplished.

/4 roll(), rollDie() and hasDoubles() adhere fully to the UML and descriptions (& pass unit tests)

/4 getDieValues() is implemented correctly as a defensive-copy (& pass unit tests)

/4 toString() method passes unit tests (and does NOT use System.out.println)

Part 2 – DiceClient

/4 Samples are randomly generated as described

/4 Average of samples is calculated correctly

/4 Standard deviation of samples is calculated correctly.

/4 Histogram is printed correctly, with a single * printed for every 10 samples of a value.

Part 3 TicTacToe

/4 Dice constructors (and reset) initialize all instance variables (& pass unit tests)

/4 takeTurn() correctly implemented, including checks for out-of-range parameters as well as requesting an already-occupied square.

- Exemplary – ONLY if calls findWinner() as part of the implementation.

/4 findWinner() correctly finds any complete horizontal or vertical row AND is well-structured

// More on next page

Overall Code Inspection:

DOES the code adhere to the Java Style Guidlines? TBD
(https://learn.zybooks.com/zybook/CARLETONSYSC2004SchrammWinter2018/chapter/2/section/8 )

/4 - For Whitespace – Appropriate, consistent whitespace between lines of methods as well as spaces between operators and arguments (Example: x = 5;  instead of x=5; ).  Readability.

/4 - For Braces – consistent placement of { } for readability

/4 - Naming of any working variables using conventions

/4 – Efficient use of variables – avoidance of repetitive allocation of variables through judicious use of variable scope.

**/4 * 2 – Well-constructed JavaDoc for all methods in all classes.**