COMP 110 Programming Exercise: Simulation of the Game of Craps

Craps is a game of chance played by rolling two dice for a series of rolls and placing bets on the outcomes. The background on probability, random numbers, simulation, and the rules of the game are given below. A simulation is a different kind of computer application than those considered so far. Rather than a problem to be solved, a simulation is a computer reenactment of a real process that may include probabilistic (rather than deterministic) behavior. By reenacting the process in a computer program, the process can be studied in detail, and the factors that influence its outcome can be hypothesized and tested.

Background: Probability of Dice Throws
A single six-sided die is assumed to be fair, which means that when thrown once, all possible roll values 1 through 6 are equally likely.

<table>
<thead>
<tr>
<th>Value</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/6 = 16.7%</td>
</tr>
<tr>
<td>2</td>
<td>1/6 = 16.7%</td>
</tr>
<tr>
<td>3</td>
<td>1/6 = 16.7%</td>
</tr>
<tr>
<td>4</td>
<td>1/6 = 16.7%</td>
</tr>
<tr>
<td>5</td>
<td>1/6 = 16.7%</td>
</tr>
<tr>
<td>6</td>
<td>1/6 = 16.7%</td>
</tr>
</tbody>
</table>

When two dice are thrown, the possible roll values range from 2 through 12, but since each die throw is an independent event, the values 2 through 12 are not equally likely. Considering that there are 6*6 = 36 possible outcomes when throwing 2 dice, the probability of a given value is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Outcomes for That Value</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1: 1+1</td>
<td>1/36 = 2.78%</td>
</tr>
<tr>
<td>3</td>
<td>2: 1+2  2+1</td>
<td>2/36 = 1/18 = 5.56%</td>
</tr>
<tr>
<td>4</td>
<td>3: 1+3  2+2  3+1</td>
<td>3/36 = 1/12 = 8.83%</td>
</tr>
<tr>
<td>5</td>
<td>4: 1+4  2+3  3+2  4+1</td>
<td>4/36 = 1/9 = 11.1%</td>
</tr>
<tr>
<td>6</td>
<td>5: 1+5  2+4  3+3  4+2  5+1</td>
<td>5/36 = 13.9%</td>
</tr>
<tr>
<td>7</td>
<td>6: 1+6  2+5  3+4  4+3  5+2 6+1</td>
<td>6/36 = 1/6 = 16.7%</td>
</tr>
<tr>
<td>8</td>
<td>5: 2+6  3+5  4+4  5+3  6+2</td>
<td>5/36 = 13.9%</td>
</tr>
<tr>
<td>9</td>
<td>4: 3+6  4+5  5+4  6+3</td>
<td>4/36 = 1/9 = 11.1%</td>
</tr>
<tr>
<td>10</td>
<td>3: 4+6  5+5  6+4</td>
<td>3/36 = 1/12 = 8.83%</td>
</tr>
<tr>
<td>11</td>
<td>2: 5+6  6+5</td>
<td>2/36 = 1/18 = 5.56%</td>
</tr>
<tr>
<td>12</td>
<td>1: 6+6</td>
<td>1/36 = 2.78%</td>
</tr>
</tbody>
</table>

So 2 and 12 are the least likely values, and 7 is the most likely value.
Pseudo-Random Numbers on the Computer

Craps is an example of a game of chance. There is no “algorithm” to determine the outcome of throwing two dice, there are only probabilities of specific values turning up. So there is no way to write a computer program to “solve” the Craps “problem.” Instead, what is of interest is to write a computer program to “simulate” or reenact the playing of the game and gather data and draw conclusions by observing and measuring the reenactment. Simulation of a probabilistic event on a computer is often implemented with the help of a special feature called a **pseudo-random number generator**. Most programming languages provide a library function or method for this. A pseudo-random number generator is a function or method that, when called, returns a number that appears to be random. The word “pseudo” implies that the generated numbers give the appearance of being random, but are not truly random. If a sufficiently large series of pseudo-random values is generated, a pattern will emerge that reveals the numbers are not truly random. But if the scope of the simulation is constrained, the underlying pattern will not be detected, and the difference between pseudo-random and truly random numbers on the accuracy of the simulation will be minimal.

In Java, the pseudo-random number generator is named `Random` and is provided in the `java.util` package, which must be imported at the top of the program. The generator must be instantiated or initialized before its first use. Once it is instantiated, the `nextInt()` method is used to generate a new pseudo-random integer. The method can accept an optional input parameter to restrict the range of the number generated.

`nextInt()`: generates a pseudo-random number on the range \(-2^{31} \text{ to } 2^{31}-1\) (-2147483648 to 2147483647).

`nextInt(n)`: generates a pseudo-random number on the range 0 to n-1.

The numbers returned are distributed uniformly. There are techniques to convert uniformly distributed values to some other distribution (exponential, normal, etc.). Here’s a short example of code to generate and print out 100 pseudo-random numbers on the range 0 to 49:

```java
import java.util.*;
...
Random r = new Random();
for (int i=1; i<=100; i++) System.out.println( r.nextInt(50) );
```

We can improve the code by representing the number of values to generate by n and the range by range:

```java
import java.util.*;
...
int n = 100;
int range = 50;
Random r = new Random();
for (int i=1; i<=n; i++) System.out.println( r.nextInt(range) );
```
Initializing or “Seeding” a Random Number Generator

When Random is instantiated, it is automatically given an initializing value called a seed. The seed determines the starting point for the series. For many simple simulations, the user does not need to worry about the seed, the Java environment will provide an acceptable seed automatically. But if multiple simulation runs are required, with each run treated as a separate sample, the user might want to change the seed for each run, and manage the values themselves. Look at the following code:

```java
Random r = new Random();
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) );
```

If we run this code several times we get a different series of 10 random numbers from 0 to 999 each time.

```
C:\> java R
607 164 2 344 329 177 550 298 587 818
C:\> java R
23 775 14 353 525 606 760 251 776 477
C:\> java R
940 72 500 748 158 519 724 56 947 551
C:\> java R
972 25 122 910 125 417 574 232 309 949
C:\> java R
966 572 498 654 546 348 794 456 738 198
```

Each time the program is run, a different (unknown) seed is selected automatically, and the series of numbers is different from the previous run. Note that your series will not be the same as mine. Now look at the following examples:

```java
Random r;
r = new Random(4712516793795885967L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) );
r = new Random(6823933401192084983L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) );
r = new Random(5807747892661018243L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) );
```

In these cases, Random is instantiated with a long int value chosen by the user. The letter “L” appended to the end of the number is required for the Java compiler to treat the number as a long int rather than a regular int. The outputs are:

```
11 549 980 989 698 452 202 636 560 904
518 7 691 987 117 767 942 477 410 798
784 850 72 825 983 927 518 134 583 449
```

As in the first example, anytime the seed is changed, the resulting series of values will be different. But now run the experiment by repeating the seed value several times:

```java
Random r;
r = new Random(4712516793795885967L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) + " ");
r = new Random(4712516793795885967L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) + " ");
r = new Random(4712516793795885967L);
for (int i=1; i<=10; i++) System.out.println( r.nextInt(1000) + " ");
```

The outputs are now:

```
11 549 980 989 698 452 202 636 560 904
11 549 980 989 698 452 202 636 560 904
11 549 980 989 698 452 202 636 560 904
```

In this case, the seed value is reused three times. Each time Random is initialized with the same seed, the same series of numbers is generated. In other words, the seed value completely determines the series of number, which reveals them to be pseudo-random rather than truly random. By remembering or recording the seed, the programmer can regenerate the same series of values for multiple runs of the simulation. This technique is useful if the programmer wants to rerun a specific simulation case for checking results or debugging the code for a specific series.
Pseudo-random numbers are a useful tool for computer simulations. There are many tests from statistics that can be performed on pseudo-random numbers to demonstrate that they are behaving sufficiently randomly (the Chi Square test, the Kolmogorov-Smirnov test, etc.). For many simple simulations, these additional tests are not necessary.
**Rules of the Game**
Craps is played by a group of players surrounding the craps table. The player currently rolling the dice is called the shooter. Other players can place bets on the outcome of the rolls. Although the overall game and the strategies for betting can be complex, for the sake of this exercise, we will just focus on simulating a single series of rolls, or one *round* of play.

Each round is divided into the first roll (the come out) and the remaining rolls (the point). The round starts by the player throwing the two dice for the come out roll. If the value is 7 or 11, the round is over and the shooter wins (as well as any other observers who placed bets that the shooter would win). If the value is 2, 3, or 12, the round is over and the shooter loses. If the value is any other value (4, 5, 6, 8, 9, or 10), the game moves into the point phase, and the shooter must keep rolling. The value of the first roll is remembered as the point, then the shooter must keep rolling until either the roll equals the point for a win, or the roll is 7 for a loss, whichever happens first. Note that rolling a 7 on the come out roll results in a win, but rolling a 7 during the point phase results in a loss.

**Example Rounds**
Here’s the result of running the simulation for 20 rounds. For each round, look at the roll values and make sure you understand the rules and you can explain why each round ends in a win or a loss.

<table>
<thead>
<tr>
<th>round</th>
<th>roll</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Lose!</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Lose!</td>
</tr>
<tr>
<td>3</td>
<td>8 5 8</td>
<td>Win!</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>Win!</td>
</tr>
<tr>
<td>5</td>
<td>8 5 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>6</td>
<td>4 10 8 8 9 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>7</td>
<td>5 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>Lose!</td>
</tr>
<tr>
<td>9</td>
<td>6 3 8 4 6</td>
<td>Win!</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>Lose!</td>
</tr>
<tr>
<td>11</td>
<td>4 3 10 6 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>Win!</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>Win!</td>
</tr>
<tr>
<td>14</td>
<td>5 6 6 9 8 5</td>
<td>Win!</td>
</tr>
<tr>
<td>15</td>
<td>6 5 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>16</td>
<td>6 4 6</td>
<td>Win!</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
<td>Lose!</td>
</tr>
<tr>
<td>18</td>
<td>8 6 7</td>
<td>Lose!</td>
</tr>
<tr>
<td>19</td>
<td>8 9 4 9 8</td>
<td>Win!</td>
</tr>
<tr>
<td>20</td>
<td>10 6 11 9 10</td>
<td>Win!</td>
</tr>
</tbody>
</table>
Writing a Craps Simulation Program
You will organize the code into two methods, roll() and round(). The skeleton of the code will be as follows:

```java
import java.util.*; // package for Scanner and Random

public class Craps {
    // static class variables
    // declare variables to be shared between methods here

    public static Random r;
    public static Scanner in;

    // static block contains initialization
    // this code can be placed in the main method as an alternative
    static {
        r = new Random();
        in = new Scanner(System.in);
    }

    // this method returns a value between 2 and 12
    // to simulate the roll of two dice
    public static int roll() {
        return 0;
    }

    // this method implements the rules of the game for one round
    // it calls the roll() method to simulate throwing the dice
    public static boolean round() {
        return true;
    }

    public static void main(String[] args) {
        // simulation is run here
        // for example, to play one round:
        // boolean result = round();

        // another example, to play n rounds:
        // int n = 10;
        // for (int i=1; i<=n; i++) {
        //    boolean result = round();
        // }
    }
}
```
Skeleton for the round() method.

The `round()` method contains the core logic to be implemented. Read the rules of the game, and using the tools discussed – such as the random number generator, methods, and static variables that are shared between methods – organize your logic into executable Java statements. Here are some observations to help you get your code organized.

1. According to the rules of the game discussed above, one round of Craps begins with a roll of the dice. Based on the value of that roll, there are three possible outcomes: a win, a loss, or a period of continued play called the “point.”
2. In the point section, the value of the first roll is remembered as the point value. The player now continues to make additional rolls until either (1) the roll equals the point, or (2) the roll equals 7. If the user rolls the point value, the round ends with a win. If the user rolls a 7, the round ends with a loss.
3. After every roll, the value must be output to the screen so the user can observe the progress of the game.

Convert these observations into Java if-else statements for decisions, and Java while statements for repetition.

Skeleton for the roll() method.

The `roll()` method will produce a value that represents a random number between 2 and 12 to simulate rolling the dice. This is a very short method, but in order for the simulation to be accurate, the random number generator must be used correctly to reproduce the probabilities described above.

With the `roll()` method available, the `round()` method is simplified by calling the `roll()` method whenever a random dice roll is required. In this way the `round()` method does not have to repeat the code required to use the random number generator directly.

The random number generator only has to be created and initialized once for the whole game. That is why this code is placed into the common and static areas of the class definition.
Additional Modifications

1. Turning Printing On and Off Without Modifying Code and Recompiling
There are actually two versions of the program that would be useful in different circumstances. During the initial testing phase, the program should print out detailed information about each roll for each round. This is necessary for the programmer to confirm that the logic is correct.

But once the logic has been verified, the programmer might want to use the simulation to collect statistics on the numbers of wins and losses. To get accurate numbers, a large number of rounds must be simulated – 10,000 or 100,000 or 1,000,000. For such a large number of simulations, detailed output of every round will be a waste. The logic has already been verified, and the programmer is no longer interested in seeing the details of each round, but only the aggregate statistics at the end of the simulation.

An easy way to allow the user of the program to turn detailed printing on and off is to use a boolean flag as a print setting, then put an if-statement that checks the flag in front of every output statement.

Original Code
```java
int roll1 = roll();
System.out.print(roll1);
```

In this version, the value of roll1 is uniformly always printed out, with no runtime check.

Improved Code
```java
public static boolean print = ...;
...
int roll1 = roll();
if (print==true) System.out.print(roll1);
```

In this version, a boolean flag named “print” is declared, and initialized to either true or false. Then in the code, every output statement is placed inside a decision statement controlled by “if (print==true) ...” If the print variable is set to true, the output statement executes. If the print variable is set to false, the output statement doesn’t execute. In this way, the print and non-print versions of the program can be controlled by the value assigned to the print variable.

2. Controlling the number of rounds
Let the user input the number of rounds n, then from the main method, call the “rounds()” method n times from a loop.

3. Collect Win/Loss Statistics
For large n, turn off printing and collect statistics on the number of wins and losses. Create a variable counter “wins” initialized to 0. For each win, increment wins. At the end of all rounds, output wins/n as a percentage.
4. Using Command Line Input to Customize Program Execution

As an alternative to creating a Scanner, programs can receive a small amount of input from the command line. The main method provides all command line information (or arguments) as the String array “args.” All args are provided as strings. If a numerical value is needed, the string must be parsed. As an example, suppose the java Craps program is run from the command line or terminal with the command:

```
> java Craps 100 true
```

Then args[0] is “100” and args[1] is “true.” Inside the program, these values can be accessed as follows:

```java
public static void main(String[] args) {
    int n;
    boolean print;
    n = Integer.parseInt( args[0] );
    print = Boolean.parseBoolean( args[1] );
}
```

In JGrasp, the command arguments can be accessed from the “Build” menu by checking the “Run Arguments” menu item. A one-line text box near the top of the screen just under the menu bar will be displayed. Command arguments can be placed here.

Note that like any array, the args array should be checked for length for a more reliable program. If you write a program that expects command line args and you don’t provide any, the program will hit a runtime exception and crash.

```java
if (args.length == 2) {
    ...
} else {
    System.out.println(  
        "Please run again with correct command line inputs."  
    );
}
```