—Assignment #1—

Objectives

1. Create classes and use them
2. Use random generated numbers
3. Review your understanding of numeric primitive data types
4. Review loops and arrays of primitive data types

The code for the class `Random` is available on Blackboard in the same place this assignment document is posted. We also wrote the `Random` class at the end of last lecture. It is normal to be totally confused the FIRST time you read detailed instructions. Read the best you can once all the way through as soon as possible, take a bit of time away from it, and then read it again. If you are still having trouble understanding after multiple readings, I am very happy to help when you send me a clear, specific question. If you message me with comments like, "What do I do first?" or other questions that show absolutely no effort from you, I will simply tell you to read the assignment document.

You will need to create two more classes:

- `BinSort` — to sort a sample of random numbers into a bar chart.
- `RandomPlay` — contains a `main` method to run your program.

Make sure to also use Javadoc to document all three classes. Use doc comments to describe each class, and each method in the class. Keep it simple, and only use the `@author`, `@param`, `@return` tags for now. Don't worry about using any HTML. To keep it organized, compile your documentation web pages into a `doc` folder within your `RandomNumbers` folder, like we did in class.

If you need help understanding what a bar chart is, then visit the following web page:

https://www.mathsisfun.com/data/bar-graphs.html

To simplify printing, we will make our final output a horizontal bar chart. Because we will be generating integers, and not floating-point numbers, the categories of the bar chart will be groups of integers. If it helps you simplify things in your mind, then you can replace all further mentions of bar charts in these instructions with histograms, and think of generating random floating-point numbers instead (and we just happen to always end up with integers being generated).

The `BinSort` class needs five instance variables (ONLY declare them):

- `final int MAX` — the possible random numbers generated can be any integer from 0 to `MAX` − 1.
- `final int N_BINS` — the number of bins.
- `final int N_SAMPLES` — the total count of random integers that will be generated.
- `final float BIN_WIDTH` — the width of each bin interval.
- an `int` array called `binCount`

In a bar chart, we have a set of `N_SAMPLES` of random numbers generated between [0, `MAX`), and we divide up the interval [0, `MAX`) on the y-axis into smaller intervals:

- the categories (or bins) are each one interval on the number line of width `BIN_WIDTH`

- the horizontal length of the rectangle of each bin is the number of times any random numbers generated come from within that bin

---

Simplify things in your mind by thinking through specific numbers in an example. Suppose we have:

`MAX = 10`, `N_BINS = 2`, `N_SAMPLES = 12`

This would mean we will generate 12 random integers between 0 and 9, inclusive, and then categorize them into bins $[0, 5)$ and $[5, 10)$. Notice that `BIN_WIDTH = 5.0` is the length of the interval of each bin.

Suppose we generated the following 12 random integers:

$$4, 7, 2, 9, 3, 3, 7, 1, 4, 5, 0, 9$$

Then the generated numbers $0, 1, 2, 3, 3, 4, 4$ all belong to bin $[0, 5)$. There are 7 values of our sample in this first bin. Also, the generated numbers $5, 7, 7, 9, 9$ all belong to bin $[5, 10)$. There are 5 values of our sample in this second bin.

Then we can print a bar chart for this example. We'll keep printing as simple as possible for this assignment, and just print the same number of `*`s as the number of values in each bin. For our example:

```
12 random integers in [0, 10) sorted into 2 bins:
***** 5 [5.0, 10.0)
******* 7 [0.0, 5.0)
```

Print the number of stars for a bin directly after the stars. At the end of each line, print the bin interval. Notice that the lines of the bar chart are printed so that the bin intervals increase up the y-axis.

Draw this bar chart on paper with coordinate axis if you need to spend some more time absorbing this example to convince yourself of all the values and what each of them mean. Split up the y-axis into the two bins. Draw a horizontal bar for each bin with length giving the number of random integers in the bin.

The behaviour of `BinSort` needs:

- a constructor with 3 parameters `max, nBins, nSamples` used to initialize:
  - `MAX`,
  - `N_BINS`,
  - `N_SAMPLES`, and
  - calculate the value of `BIN_WIDTH` as `MAX / N_BINS` (use explicit casting to avoid integer division!)
- a void instance method `generateBins` that uses no parameters
- a void instance method `printBins` that uses no parameters

Pseudo-code for the method `generateBins`:

1. Initialize the `binCount` array to have `N_BINS` number of elements (an element for each bin). Let Java default the value of all elements to zero (you don't need to set any values for the elements here).

2. Loop with counter variable `i` for each integer from 0 to `(N_SAMPLES - 1)`

   (a) Declare a variable `rNum` and set it to a random number generated with the class method `rand` from `Random` (pass in the argument values `i` and `MAX`).

   (b) Declare an `int` variable `bin` and set it to the index of which bin `rNum` belongs to. You can get the correct bin index by first dividing `rNum` by `BIN_WIDTH`, and then (because it will never be a negative result) use `Math.floor()` to ignore the fractional part. Finally, you will have to cast the return value of `floor` to an `int`.

   (c) Increment `binCount` for the `bin`. This keeps track of the number of random integers generated inside each bin interval.

   (when `generateBins` executes, the `binCount` array holds the bar chart data we need to output)

Pseudo-code for the method `printBins`:

1. Loop with counter variable `i` for each integer from `(binCount.length - 1)` to 0. (one of the advantages of arrays in Java is that they are objects with field variable `length` that gives the number of elements in the array)

   (a) Print the number of stars that matches `binCount[i]` on one line. You can use a loop to do this and `System.out.print` one star at a time.

   (b) Declare a `float` variable `binMin` and set it to `i * BIN_WIDTH`. This is the lowest value in the i^{th} bin interval.

   (c) Declare a `float` variable `binMax` and set it to `binMin + BIN_WIDTH`. All values in the i^{th} bin are strictly below `binMax`.

   (d) Use `binCount[i]`, `binMin`, and `binMax` to print the data after the stars for the i^{th} bin as shown in the example output. Use interval notation in your output.

   (e) Make sure you advance to the next line to print the data for the next iteration of the loop. Don't print any blank lines. The output should match style with the examples of output given in this document.

   (when `printBins` executes, the bar chart should be output to the console window)

Once you get `BinSort` written, you can then start to check your work against examples in this document. Inside the `main` method of `RandomPlay`, you can try the following code:

```
System.out.println("12 random integers in [0, 10) sorted into 2 bins:");
BinSort sorter = new BinSort(10, 2, 12);
sorter.generateBins();
sorter.printBins();
```

And your output should look something similar to:

```
12 random integers in [0, 10) sorted into 2 bins:
******* 7 [5.0, 10.0)
***** 5 [0.0, 5.0)
```

Notice for this output, the number of random values generated in each bin is slightly different. This is okay. The goal is to check our set of generated random values. The bar chart helps us answer the question, "Is the randomness fair?" In other words, is each possible value we want to generate equally likely to happen?

Finally, add more code to `main` so that you get similar output to the following example output below. You can reuse the `sorter` variable and set it to a new instance with arguments passed into the `BinSort` constructor that match with the examples in the output below. The bar charts will have slightly different lengths for each bin, but all bins should be fairly close to the same length (because we are designing fair outcomes in our randomness). View the bar charts without line-wrapping in your console window.

```
12 random integers in [0, 10) sorted into 2 bins:
****** 6 [5.0, 10.0)
****** 6 [0.0, 5.0)

500 random integers in [0, 100) sorted into 10 bins:
************************************************** 52 [90.0, 100.0)
****************************************************************** 64 [80.0, 90.0)
********************************************** 47 [70.0, 80.0)
******************************************** 45 [60.0, 70.0)
******************************************* 44 [50.0, 60.0)
************************************************** 50 [40.0, 50.0)
********************************************* 45 [30.0, 40.0)
************************************** 40 [20.0, 30.0)
********************************************* 45 [10.0, 20.0)
************************************************************************ 68 [0.0, 10.0)

300 random integers in [0, 50) sorted into 20 bins:
**************** 16 [47.5, 50.0)
**************** 16 [45.0, 47.5)
*************** 15 [42.5, 45.0)
************************** 26 [40.0, 42.5)
************** 14 [37.5, 40.0)
***************** 17 [35.0, 37.5)
*********** 11 [32.5, 35.0)
***************** 17 [30.0, 32.5)
********** 10 [27.5, 30.0)
****************** 18 [25.0, 27.5)
************ 13 [22.5, 25.0)
********************** 22 [20.0, 22.5)
********* 9 [17.5, 20.0)
************ 13 [15.0, 17.5)
********* 9 [12.5, 15.0)
************ 13 [10.0, 12.5)
******* 7 [7.5, 10.0)
************ 13 [5.0, 7.5)
*********** 12 [2.5, 5.0)
***************************** 29 [0.0, 2.5)
```

The bottom line of output here means: out of 300 random integers generated between 0 and 49, inclusive, 29 of them resulted in the numbers 0, 1, or 2.

Submit Instructions and Marking

The deadline is listed on Blackboard under Assignment 1, and you need to submit your work there. Compress (zip) your RandomNumbers project folder containing the `Random.java`, `BinSort.java`, `RandomPlay.java` source code classes, and the `doc` folder, and upload only the one RandomNumbers zipped file.

The assignment is out of 40 marks and will generally be distributed as follows:

- 0–10: your source code is not named correctly, does not compile, or the output is not anywhere close to a solution to the problem.
- 10–20: your source code compiles, but there are bugs that stop the program from running with the expected output, or your code is written in a grossly unreadable way.
- 20–30: your source code compiles, runs with most of the expected output, or there are still some output bugs that stop it from being a complete solution, or your code is written in a way that is difficult to read.
- 30–40: your source code compiles, runs with all expected output, and is a complete solution, maybe only some minor output bugs, and your code is written in a way that makes it easy to read. Your documentation also compiles and describes your code both accurately and concisely.

Further Thought

Making games often involves designing some kind of simulations. Either simulating real world situations, or an imagined world with made-up rules. If everything in a simulation follows quickly recognized patterns, then it is very difficult to make any part of that simulation stand out above any other part. The human brain is hard-wired to look for and recognize patterns when contrasted against chaos. Therefore, a simulation should include some chaos (randomness). Think of randomness as the empty space surrounding and emphasizing a point of focus/interest.

The numbers generated by the Random class for this assignment are only pseudo-random. Samples of these generated numbers do not exactly fit a perfectly fair probability distribution for each number in the sample. In other words, some numbers are generated more often than others. For simple games and simulations, pseudo-random number generators like this are a good start to learning how to "control" randomness.

Being able to create a bar chart and read the chart can give you a quick visual way of understanding what kind of pseudo-random samples you can generate. If your goal is perfect fairness so that each possible number has the same likelihood of being generated, then you want your whole bar chart to look as close as possible like one big rectangle (ideally for bins of width 1).

However, the order that numbers in the sample are generated is *completely* dependant on the order of values of `seed` in the calculations we used to write the static `rand` method. The parameter value `count` is needed so that we can generate different numbers for each successive call to `rand` that happens faster than the system `time` can change. Many hundreds (or more) of instructions can execute in a millisecond, and `time` would hold the same value during that span of instructions. But, the value of system `time` makes it possible to generate different pseudo-random numbers whenever we execute the program again later.

When debugging games, it is useful to remove the system `time` from the calculations in `rand` so that you can always generate the same set of pseudo-random numbers every time you run your game.

Not only is pseudo-randomness useful in simulating different behaviours, it is also immensely useful for procedural graphics. Small, indy developers can consider much larger content projects only limited by the computational power of the hardware they use and the mathematics to describe the content.

Ultimately, if you can design simulations and content through the control of both randomness and patterns together, then you can control contrasting both against each other to help direct the attention of people using your programs.