Computer Science E214

Tutorial 1

2015

1 The Argument Clinic

Write a program `ArgumentClinic` that takes two `String` parameters, which are the names of the two souls who will engage in an argument. Your program should behave in the following way:

```
$ java ArgumentClinic Palin Cleese
Palin is having an argument with Cleese.
```

Thus the first soul specified should be arguing with the second.

Note that using the "Run" button in DrJava should give you an error, since your program should be expecting command line arguments which are not provided when you click on the button. Instead, try typing the `java ArgumentClinic Palin Cleese` command into the `Interactions` pane, and it should execute. Modify your program until it works correctly for this example. Also ensure that you can execute the program with command line arguments at the command prompt. (This also requires typing the `java ArgumentClinic Palin Cleese` command if you have already compiled your program in DrJava — otherwise you will need to compile the program on the command line first.)

Note that, if you wish to use names and surnames, the following should work (given that your program is correct).

```
$ java ArgumentClinic "Michael Palin" "John Cleese"
Michael Palin is having an argument with John Cleese.
```

Without the quotes, this might happen:

```
$ java ArgumentClinic Michael Palin John Cleese
Michael is having an argument with Palin.
```

This is because the command-line has no way of knowing that the space between a name and surname is any different from the spaces between names of the argumentative individuals. Using the quotes signals that the everything inside should be treated as one command-line argument.

If you haven't made special provision yet, your program will do the following when only one name is given:

```
$ java ArgumentClinic Michael
java.lang.ArrayIndexOutOfBoundsException: 1
at ArgumentClinic.main(ArgumentClinic.java:NNN)
```

This is because there is only one parameter, but your program probably uses `args[1]`. Note that NNN will be the line number of showing where in your program the program stopped.

You can ask `args` how many arguments it contains using `args.length`, which returns that number as an integer. Use that, and `if` statements, to change your program to behave in the following way.

```
$ java ArgumentClinic
*crickets*
$ java ArgumentClinic Palin
Palin is arguing with themself.
$ java ArgumentClinic Palin Cleese
Palin is having an argument with Cleese.
$ java ArgumentClinic Palin Cleese Idle
Much hilarity ensues.
$ java ArgumentClinic Palin Cleese Idle Chapman
Much hilarity ensues.
```

[If the sentence for one argument causes your eyes to bleed, consider http://blog.oxforddictionaries.com/2013/01/themself/, though it may not entirely stop the bleeding.]

Now, obtain the file `ArgumentClinic.java` made available with the tutorial, and place it in a new directory. (This is to ensure you do not overwrite your own solution!) Verify that compiling and running it behaves exactly as specified above. Next use this version to run `java ArgumentClinic Idle Gilliam`. Is the output what you would expect? Open the program in DrJava, and try to understand why the program behaves as it does.

Conclusion: getting the right output for the examples provided does not necessarily mean your program is working as intended.

2 Book 1.2.18 (modified)

Write a program `SideLength` that takes two floating point numbers as parameters, which are the x- and y-coordinates of a point in the Cartesian plane. The program must then print out the distance from the origin to the point. Example usage:

```
$ java SideLength 3.0 0.0
3.0

$ java SideLength -3.0 4.0
5.0
```

3 Book 1.2.14 (modified)

Write a program `DividesEvenly` which takes two positive integers as command-line arguments and prints `true` if either evenly divides the other, or `false` otherwise. Example usage:

```
$ java DividesEvenly 5 10
true

$ java DividesEvenly 5 6
false
```

4 Book 1.2.30

Write a program `UniformRandomNumbers` that prints five uniform random values between 0 and 1, their average value, their minimum value and maximum value. Use `Math.random()`, `Math.min()`, `Math.max()`.

5 Example: Estimating Probabilities and Averages

This example first aims to give you some guidance around estimating probabilities from simulations. The question is:

Ford Prefect throws a single six-sided die. Estimate by simulation the probability that the top number will be either a 2 or a 4.

The answer is fairly obvious for this question, but you will answer this in an "overkill" fashion which generalises nicely to more complex simulation problems.

Roughly speaking, there are three things we need to define when analysing a problem statistically. They are *the experiment*, the possible *outcomes* of the experiment, and the *event* of interest.

Experiment: The thing of interest that has uncertain outcomes. In this case: throwing a six-sided die.

Outcomes: Possible results of running an experiment. In this case: 1, 2, 3, 4, 5 or 6. Each outcome has some probability of occurring, often this probability is unknown *explicitly*. In this case, each of the outcomes has a 1/6 chance of happenings (assuming the die is fair).

Event: A set of outcomes of interest. In this case: 2 or 4.

In order to **estimate** the probability of an event, we can run the experiment N times, record the outcome each time, and count how many times N_E the outcome corresponds with the event of interest E. Below is an example of running a series of experiments, recording their outcomes, and checking whether the outcome corresponds to the event of interest.

Experiment Number	Outcome	Event happened? (rolled 2 or 4)
1	4	Yes
2	1	No
3	5	No
4	3	No
5	6	No
6	2	Yes
7	4	Yes

To estimate the probability of the event, we divide the number of times the event happened by the number of experiments:

$$P(\text{number is 2 or 4}) \approx \frac{\text{Number of times event occurred}}{\text{Number of experiments}} = \frac{3}{7}$$

The correct probability is, of course, 2/6, so this estimate is not too far off. We can improve the estimate by increasing the number of experiments.

We can do the same in Java using something along the lines of:

```
int N = ... number of simulations ... ;
int N_events = 0; // Number of times event occurs

for (int i = 0 ; i < N ; i++) {
... Do the experiment ...
if (outcome corresponds with event) N_events++;
}

double prob = N_events / (double) N;
```

Note the use of the cast `(double)`, without which the result will be zero. (Be sure you know why!).

We can thus answer the question as follows:

```
int N = 10000;
int N_events = 0;

for (int i = 0 ; i < N ; i++) {
    int die = 1 + (int)(Math.random() * 6);
    if ((die == 2) || (die == 4)) N_events++;
}

double prob = N_events / (double) N;
System.out.println(prob);
```

Copy and paste the code into the Dr Java interaction window to test it out. This should give you about 0.33 (or 2/6).

In subsequent questions, the experiments you perform will be more complex, and the outcomes/events tested for a more complicated, but the principle is the same. Run N simulations, check how many outcomes correspond with the event you are interested in, and take the ratio of the number of times the event occurred to the number of experiments.

Averages: A simular argument to the above can be used for calcuating the expected value (average) of a quantity from simulations: run N simulations, total up the value of the quantity in each simulation, and divide the total by the number of repetitions.[1] An example question that you might tackle of this nature:

Ford Prefect repeatedly throws a single six-sided die until the top number is a 2 or a 4. Estimate by simulation the average number of throws Ford will make.

In this case, a simulation will not just be a single roll of the dice, but a sequence of rolls ending with a 2 or a 4. This leads us to the solution:

```
int N = 10000;
int total = 0;

for (int i = 0 ; i < N ; i++) {
    int numrolls = 0;
    int die;
    do {
        die = 1 + (int)(Math.random() * 6);
```

[1] In fact, estimating probabilities is a special case of this more general scheme — what is the quantity you are calculating the average of in this case?

```
        numrolls++;
    } while ((die != 2) && (die != 4));
    total += numrolls;
}

double avg = total / (double) N;
System.out.println(avg);
```

6 Web 1.3.20

Alice tosses a fair coin until she sees two consecutive heads. Bob tosses another fair coin until he sees a head followed by a tail. Write a program `CoinToss` to estimate the probability that Alice will make fewer tosses than Bob?

Your program should take the number of simulations as its first parameter and produce output like the following:

```
$ java CoinToss 3
Running 3 simulations.
0.6666666666666666
```

If your program is passed any other parameters (regardless of what they are), you should produce output like the following describing what is happening in each simulation:

```
$ java CoinToss 3 whateverlkewnfsohg
Running 3 simulations.
Alice:HTTHH Bob:HT
Alice:HH Bob:HHT Alice has fewer.
Alice:HH Bob:HT
0.3333333333333333
```

(Recall that the number of parameters passed to the program can be obtained from `args.length`, so you can use that to test whether more than one parameter has been passed.)

The exact answer is 39/121, so you can check your solution for many simulations — note that examples above may give poor estimates, because the number of simulations is very low!

7 Book 1.3.40

Note that this is numbered Book Exercise 1.3.41 on the web.

In the 1970s game show *Let's Make a Deal*, a contestant is presented with three doors. Behind one of them is a valuable prize. The contestant chooses a door, but does not open it. The host (who knows which door contains the prize) then opens one of the other two doors that does not contain the prize (there must be at least one such door) and shows the contestant that the prize is not there. The contestant is then given the opportunity to switch to the other unopened door. Should the contestant do so? Intuitively, it might seem that the contestant's initial choice and the other unopened door are equally likely to contain the prize, so there would be no incentive to switch.

Write a program `MakeADeal` that takes a command-line argument N, plays the game N times using each of the two strategies (switch or not switch), and prints the chance of success for each strategy.

Your program should take the number of simulations as its first parameter and produce output like the following:

```
$ java MakeADeal 3
Running 3 simulations.
Probability of win if stays:0.3333333333333333
Probability of win if changes:0.6666666666666666
```

If your program is passed any other parameters (regardless of what they are), you should produce output like the following describing what is happening in each simulation:

```
$ java MakeADeal 4 dsijfdsoihgsdl
Running 4 simulations.
winning_door=0 first_pick=1 opened_door=2 other_choice=0 Other choice wins!
winning_door=1 first_pick=0 opened_door=2 other_choice=1 Other choice wins!
winning_door=1 first_pick=1 opened_door=2 other_choice=0 First pick wins!
winning_door=2 first_pick=1 opened_door=0 other_choice=2 Other choice wins!
Probability of win if stays:0.25
Probability of win if changes:0.75
```

(Recall that the number of parameters passed to the program can be obtained from `args.length`, so you can use that to test whether more than one parameter has been passed.)

Note that examples above may give poor estimates, because the number of simulations is very low!

8 Book 1.4.4

Write a program `ReverseArray` that reverses the order of a one-dimensional array `a[]` of `double` values. These values are given as command-line arguments, so you should write code to place them in the array `a[]` and then to reverse `a[]` . Do not create any other arrays for intermediate processing or to hold the result. (We refer to array processing done in this fashion as *in-place*.) Hint 1: Use the code in the textbook for exchanging two elements of the array. Hint 2: `args.length` can be used to initialise `a[]`. Example usage:

```
$ java ReverseArray 40.7 35.2 21.4 13.0
13.0
32.4
35.2
40.7
```

9 Book 1.4.10

Write a program `Deal` which takes a command-line argument N, and prints N poker hands from a shuffled deck. Assume that we are playing *Five-card stud* where a hand consists of 5 cards. Example usage:

```
$ java Deal 2
4 of Spades
9 of Spades
Ace of Hearts
9 of Clubs
9 of Diamonds

6 of Spades
```

```
10 of Hearts
Queen of Hearts
8 of Hearts
King of Spades
```

10 Book 1.4.34 [Challenging]

In the game of bridge, four players are dealt hands of 13 cards each. An important statistic for bridge players is the distribution of the number of cards of each suit in a hand. Write a program `BridgeHands` to test which of the following distributions is the most likely: 5-3-3-2, 4-4-3-2, or 4-3-3-3. (Here the numbers represent the number of cards of each suit in the hand, ordered from longest to shortest suit.)

Your program must take an integer N as a command-line argument, where N is the number of times the players must be dealt a hand (i.e. how many games they play). You program should print the probability of observing each of the three distributions mentioned, and which is then the most likely distribution. Example usage:

```
$ java BridgeHands 3
Probability of 5-3-3-2: 0.16666667
Probability of 4-4-3-2: 0.083333336
Probability of 4-3-3-3: 0.083333336
5-3-3-2 is the most likely.
```

(If more than one of the distributions are equally most likely, your output may print any one as the most likely.)

Note that the estimates in the example above may be inaccurate, due to the small number of simulations. Hint: You can use the function `Arrays.sort(MyArray)` to sort the arrays which record the distributions. To use this function you have to import `java.util.Arrays`.