Computer Science E214

Tutorial 4

1 Counter Exception (Book 3.3.15)

Add code to `Counter` (Program 3.3.2 in the textbook) to throw a `RuntimeException` if the client tries to construct a `Counter` object using a negative value for `max`.

Note that exceptions are extremely important to know about as a professional programmer, since most error handling in current programming methodologies make use of exception throwing and handling. However, we will not consider advanced error handling with exceptions in this course.

2 Comparable Vectors (Book 3.3.6–3.3.8)

Consider the `Vector` API defined in Section 3.3 of the textbook, and partially implemented in Program 3.3.3. First, complete this implementation by providing the missing methods `minus` and `toString` specified in the API. Then add a sensible implementation of `equals(Object o)` to the class.[1] Next, have the data type implement the `Comparable<Vector>` interface by considering one vector larger than another if it has a larger magnitude. Write a client `VectorSort` which reads in k Vectors from standard input, and then prints them out in sorted order (from shortest to longest) by using the static method `sort(Object[] a)` method available in `java.util.Arrays`.

Some notes:

- This API and implementation has a weakness in that you can create `Vector`s of different dimensions, and then try to perform operations like addition on them, which lead to incorrect results or even exceptions being thrown. For this problem, assume that your operations deal with `Vector`s of equal length, and when creating your input files, make all the Vectors for one input run the same dimension.
- It is recommended to add a `main` method to your `Vector` class testing your modifications to the class.
- For your implementation of `minus`, use the existing implementations in the API rather than explicitly manipulating vector components directly.
- For implementing `equals`, you presumably want to compare the coordinate arrays in some way. However, the first step is gaining access to the coordinates array of the object being passed as a parameter. Since it could technically be any `Object`, we need to filter out non-`Vector`s, and after that convince Java's compiler to let us access the array for the `Vector`s that remain. You can do this with the Java keyword `instanceof`, and *casting*:

[1] Technically, we should also override another method `hashCode()` when overriding the `equals` method inherited from `Object`, but we will omit that here.

```
if (!(o instanceof Vector))
    return false;
Vector v = (Vector) o;
```

By casting, i.e. changing the declared type of the object, we are able to access fields or methods we could not earlier. Note that casting can cause a `ClassCastException` if the referenced object is not a valid value for the new declared type — hence our checking in advance using `instanceof`.

- After this, you may get some counterintuitive behaviour when comparing your arrays at first. *Ensure you understand why you get the behaviour you do, and how you can fix it.*
- When you implement the `compareTo` method, ensure it is consistent with your `equals` method — see the discussion on being *consistent with equals* in the documentation of the `java.lang.Comparable` interface.
- Read the documentation of the `sort(Object[] a)` method in `java.util.Arrays` carefully to see where the necessity for the `java.lang.Comparable` interface comes in. (The `sort` method actually casts the array elements to `Comparable` during the sort process in order to get access to each object's `compareTo` method.)
- It is up to you to decide on a sensible input format for your client (use previous examples in the course for inspiration). Document the format with comments in your client, and provide sample files that can provide input by using redirection in a folder called `input`.

3 Immutable Appointments (Book 3.3.22–3.3.23)

You are given the code `Appointment.java` (see the resources file) by your manager, and you point out to him that it could potentially create problems in the system because the data type exposes its fields and is mutable. He seems puzzled, and asks you what that means...

1. Modify `Appointment.java` to make the data type encapsulated and immutable. To illustrate potential ways in which the class may be mutable, see the methods in `AppointmentAttacks.java`. Your modifications should attempt to disable these attacks without renaming fields or methods of the `Appointment` class: specifically, if you have been successful, every attack should either cause a compiler error, or fail to modify the `Appointment`.
2. Add a block comment at the top of the class explaining in your own words what immutability is, and how it can be achieved in Java. Ask your classmates to critique your answer.

It is worth noting that many developers might consider extending the `Date` type when creating `Appointment`. However, this is generally not a good decision: it is probably more accurate to say that an `Appointment` *has-a* `Date`, than that an `Appointment` *is-a* `Date`. We thus include the `Date` as a field in the `Appointment` class. This is then an illustration of the important design principle: *Favour composition over inheritance.*

4 The Markov Hypothesis (Book 4.1.14)

Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of `Markov.java` (Program 1.6.3) as a function of the input parameters T and

N. Use the methods discussed in Section 4.1 of the textbook. Use the data files `tiny.txt`, `medium.txt`, `large.txt` from the resources to develop the hypothesis.

Note again, the resulting formula will be a function of both T and N. For each choice of N (this is chosen by choosing a file), you will need to run your program for a variety of T values. This will form a two-dimensional table, which you can use to develop your hypothesis.

Remember to verify your result! (Predict the time for `extralarge.txt` from the resources for various values of T using the model you developed, and see whether your predictions are accurate.)

Provide documentation of the above process and your findings in a text file called `Markov.txt` or a .pdf file called `Markov.pdf`.

5 The Happy Meal Shuffle (Book 4.1.17)

The following code fragment (adapted from a Java programming book) creates a random permutation of the integers from 0 to $N-1$. Determine the order of growth of its *average* running time as a function of N. Compare its order of growth with the shuffling code shortly before PROGRAM 1.4.1 in Section 1.4 of the textbook. (Note the similarities between this approach to shuffling and the coupon collector problem of SECTION 1.4.)

```
int[] a = new int[N];
boolean[] taken = new boolean[N];
int count = 0;

while (count < N) {
  int r = StdRandom.uniform(N);
  if (!taken[r]) {
    a[r] = count;
    taken[r] = true;
    count++;
  }
}
```

Submit your work on this problem in a text file called `HappyMeal.txt` or a .pdf file called `HappyMeal.pdf`.

Hint: if you repeatedly do something with independent probability of success p each time, the expected (i.e. average) number of repetitions until you succeed is $1/p$. (Challenge: Prove it!)

6 Quicksort (Book 4.2.33)

Write a method `public static void sort(Comparable[] a)` in a class `Quick` that sorts an array of `Comparable` elements. Recursively make use of a technique like that described in the Dutch National Flag problem (see EXERCISE 4.2.32): First, partition the array into a left part with all elements less than some array element v, then a middle part with any equal to v, and finally a right part with all elements greater than v. After performing this *pivot* step, recursively sort the two end parts.

Investigate the order of growth of the running time of your solution experimentally in the following cases:

- the input array is in a random order;

- the input array is in increasing order; and
- the input array is in decreasing order.

Try to explain what you see. Put your results and discussion in a text file, `Quicksort.txt`.