CPS 305 Project One: Union of Intervals

The Problem To Solve

The first project for the course CPS 305 Fall 2018 is to design an **immutable** class whose objects represent **unions of intervals of positive integers**.

An **integer interval** is defined by its `start` and `end` values so that `start <= end`, and it contains all integers `x` for which `start <= x <= end`, the start and end of the interval both being inclusive. An object that represents a **union of intervals** can contain any number of such individual intervals. However, inside this union, two integer intervals that either are contiguous or that overlap partially or in full are supposed to meld into a single integer interval. Between any two intervals that are part of this union, there must exist at least one integer value that is between them but not part of either interval.

For example, using the canonical string representation (as defined below in the section "Required Methods"), three separate examples of unions of intervals might be

```
"[1-5,10-12,27,33-40]"
"[4-15,20-31,35-42]"
"[]"
```

where the first instance consists of four separate intervals (1-5, 10-12, 27-27, 33-40), the second instance consists of three such intervals (4-15, 20-31, 35-42), and the third instance represents an empty union that contains nothing.

The **intersection** of two unions of intervals contains the integers that are included in both, encoded in the same representation. For example, the intersection of the first two example intervals would be (again converted to `String` representation)

```
"[4-5,10-12,27,35-40]"
```

Similarly, the **union** of two unions of intervals contains the integers that are included in either one or both. For example, the union of the first two example intervals would be

```
"[1-15,20-31,33-42]"
```

Automated Tester

The instructors provide an automated tester `IntervalUnionTest.java` (latest version August 31) that is used to check and grade your project. Used on the command line, it has the format

```
java IntervalUnionTest SEED N M EXPECTED
```

where `SEED` is the seed to initialize the pseudorandom number generator that produces the test cases, `N` is the number of initial single intervals to generate as test data, `M` is the number of operations to perform on these intervals.

If given, `EXPECTED` is the CRC-32 checksum computed from the results of the operations performed by your code. If `EXPECTED` is not given, the tester runs in *verbose* mode useful for debugging during development so that you can eyeball whether the results computed by your methods are correct. If `EXPECTED` is given, the tester runs silently and verifies that the computed checksum of the actual results equals the expected checksum. The tester will then print only the last line whose first item is the number of milliseconds elapsed in the test. If the computed and expected checksums are different, the running time does not matter ("If my program does not have to work correctly, I can make it run in zero time", as the famous retort once put it) and is defined to be 99999999 milliseconds.

An example run (**UPDATED SEP 16**) using ten pseudorandomly created (using `SEED` of 98765) intervals followed by twenty operations performed on them, without giving the `EXPECTED` checksum, looks like this, with the instructor's command line prompt in green followed by the actual command you should use:

```
matryximac:Java 305 ilkkakokkarinen$ java IntervalUnionTest 98765 10 20
  0: [11-15]
  1: [8-10]
  2: [0-5]
  3: [10]
  4: [14-18]
  5: [3-5]
  6: [15]
  7: [9-12]
  8: [5-10]
  9: [12-13]
 10: union of 0, 6 is [11-15]
 11: union of 4, 6 is [14-18]
 12: union of 8, 5 is [3-10]
 13: union of 9, 12 is [3-10,12-13]
 14: union of 10, 3 is [10-15]
 15: union of 3, 7 is [9-12]
 16: union of 11, 1 is [8-10,14-18]
 17: union of 4, 12 is [3-10,14-18]
 18: union of 0, 15 is [9-15]
 19: union of 11, 14 is [10-18]
 20: intersection of 15, 8 is [9-10]
 21: union of 18, 1 is [8-15]
 22: intersection of 0, 19 is [11-15]
 23: union of 0, 4 is [11-18]
 24: intersection of 10, 13 is [12-13]
 25: intersection of 2, 24 is []
 26: intersection of 7, 8 is [9-10]
 27: intersection of 15, 1 is [9-10]
 28: intersection of 0, 20 is []
 29: intersection of 12, 15 is [9-10]
202 123456789 Kokkarinen, Ilkka
```

When grading your submissions, the TA will use much bigger values of `N` and `M`, along with our private `SEED` and `EXPECTED` checksum that will not be revealed to students. For example,

```
matryximac:Java 305 ilkkakokkarinen$ java IntervalUnionTest 98765 5000 1000000 3657198581
1200 123456789 Kokkarinen, Ilkka
```

In the instructor's home computer, the above test using one million operations took a total of 1200 milliseconds. This includes all the string conversions and the bookkeeping work by the test environment, especially the CRC-32 checksum calculation. However, since all this bookkeeping work adds the exact same constant to everybody's running time, the resulting running times can be used to rank your project submissions fairly.

(If you wanted to cut the reported total running time in half, you would need to speed up your program to be way more than twice as fast, since the total bookkeeping time is the same constant regardless of the speed of your program. Even if you could make your code infinitely fast by some dark magic, the reported running time would still be nonzero, equal to the bookkeeping time of the automated tester.)

Grading

This programming project is graded for correctness and execution time. **First, to receive any marks at all, your project must work correctly in that it passes our tests with the expected checksums.** To check whether your solution is passing the automated tester, you can try out the following precomputed checksums.

SEED	N	M	EXPECTED
123456	100	500	2623176406
123456	1000	5000	1827768763

123456	10000	50000	2782781297
123456	100000	500000	1376321519

Here are some outputs from the instructors' model solution that you can compare the outputs of your code to. The links lead to Dropbox.

SEED	N	M	File
101	100	10000	s101n100m10000.txt
102	100	10000	s102n100m10000.txt
103	100	10000	s103n100m10000.txt

(On the Unix command line, the diff tool is your powerful little friend to find the first line where the output of your code and the model solution are different, to help you discover what your code is doing wrong.) Note also that the methods `getPieceCount` and `contains` are also part of the final checksum calculation, even though their results are not printed out by the automated tester.

When grading your project submissions, the TA will use a different secret `SEED` and its corresponding `CHECKSUM` to verify that your project submission is working correctly. The values used for `N` and `M` will be 100 and 1000000 (**CHANGED**). Your code has a **time limit of thirty seconds** to complete the test. If the execution of the automated tester has not terminated by that time, your project code is considered to be too slow (or even worse, has a bug that causes your code to get stuck in an infinite loop) and will be rejected with a zero mark. The same will happen if your code fails to produce the correct checksum.

The projects that pass this first hurdle of working correctly are sorted based on their running time. This list is then reasonably divided into six portions based on the general clustering of these running times. The projects in these six portions will receive a project mark of 10, 9, 8, 7, 6 and 5 points, respectively.

As an added incentive and reward, for whatever it is worth, prof. Kokkarinen will write a letter of recommendation to the student whose project submission is the fastest. The winner is also entitled to refer to him- or herself with the title of "The Fastest Gun East of Mississauga" for the duration of the Fall 2018 term. In case that several project submissions at the top are too close to call, this photo finish will be resolved with another test using the value of `M` equal to ten million.

Required Methods

Your submission must consist of exactly one source code file `IntervalUnion.java` and no other files whatsoever. (Defining nested classes inside the class `IntervalUnion` is acceptable.) **Do not**

make any package declarations in your file, since otherwise the automated tester cannot find your class. Your code is freely allowed to use all data structures and algorithms available in the Java 8 standard library so that you don't need to reinvent any of these wheels. Any attempt to interfere with the behaviour and results of the automated tester is considered cheating and will be punished by the forfeiture of all project marks in this course.

Your class must have the following **exact methods** so that the tester can compile and run. How you choose to implement these methods, and whatever `private` data fields you choose to store whatever information you use to encode and represent a union of intervals, is entirely up to you. Do something intelligent based on what you have learned in this course so far.

```
public static String getAuthorName()
```

Returns your name in the format `"Lastname, Firstname"` exactly as it appears on RAMMS.

```
public static String getRyersonID()
```

Returns your Ryerson student ID as a string without any spaces. For example, `"123456789"`.

```
public static IntervalUnion create(int start, int end)
```

A **static factory** method to create and return an `IntervalUnion` object that contains a single interval from `start` to `end`. The automated tester does not assume the existence of any kind of public constructor, but will always call this factory method whenever it needs to create a new `IntervalUnion` object. (Since the `IntervalUnion` class is designed to be immutable, this method could theoretically return an existing object to save memory, if a suitable object with the correct contents already exists.)

```
@Override public String toString()
```

Returns the string representation for the `IntervalUnion` object `this`. To allow automated testing, **this string representation must follow the exact format** of listing the intervals in sorted order between square bracket character pair so that the start and end values of each interval are separated by a minus sign. If the `start` and `end` values are equal, only one number is used. The individual intervals contained in the union must be separated by single commas, with no silly trailing comma allowed. Whitespace and other extraneous characters should not be used anywhere inside the string to separate the intervals.

```
@Override public boolean equals(Object other)
```

The equality comparison between two Java objects. An `IntervalUnion` object can only be equal to other `IntervalUnion` objects and nothing else, and two interval unions are considered equal if and only if they contain the exact same integer intervals. It is essential that you implement this method to do this instead of using the automatically inherited version that merely compares the two objects for memory address equality, since this method is implicitly part of the checksum computation in the automated tester, in the part where the `IntervalUnion` objects are added into a hash table so that equal objects end up being there only once.

```
@Override public int hashCode()
```

Returns an integer hash code for the `IntervalUnion` object `this`. You can compute this hash code any way you want, as long as it satisfies the semantic requirements expected by the Java language that `a.hashCode() == b.hashCode()` holds whenever `a.equals(b)`, where `a` and `b` can be any `IntervalUnion` objects. Try to make this method spread the hash codes all over the possible integer values to make the hash table data structure run faster, since the automated tester will use both methods `hashCode` and `equals` as it builds up a `HashSet<IntervalUnion>` instance that contains all the pseudo-randomly generated interval unions.

```
public boolean contains(int x)
```

Checks whether the `IntervalUnion` object `this` contains the integer `x`. Note that the automated tester calls this method as part of checksum computation, but the results do not show up in the textual output, only in the final value of the computed checksum.

```
public IntervalUnion union(IntervalUnion other)
```

Computes the union of `this` and `other`, returning the result as an `IntervalUnion` object. Note that this method should not modify either one of the original interval union objects.

```
public IntervalUnion intersection(IntervalUnion other)
```

Computes the intersection of `this` and `other`, returning the result as an `IntervalUnion` object. Note that this method should not modify either one of the original interval union objects.

```
public int getPieceCount()
```

Returns the number of disjoint interval pieces contained in the `IntervalUnion` object `this`. Note that the automated tester calls this method as part of checksum computation, but the results do not show up in the textual output, only in the final value of the computed checksum.