CodeCrunch

CS2030 (2310) Lab #6: Java Stream

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Java Streams

Topic Coverage

- Application of Java Streams

Requirements

- Java Streams are to be used for the method implementations as specified in this assignment

The Tasks

There are several tasks in this assignment.

For each task, you are to define the appropriate method(s) within the `Main` class in `Main.java`. We do this so that you can have your code compiled before running in `jshell`.

Task 1: Twin Primes

A prime number is a natural number greater than `1` that is only divisible by `1` and itself. A twin prime is one of a pair of prime numbers with a difference of `2`. For example, `41` and `43` are twin primes.

Define the method `twinPrimes` in class `Main` which takes in an integer `n` and returns an `IntStream` comprising of distinct twin primes from `2` to `n`.

```
static IntStream twinPrimes(int n)
```

Save your `Main` class in the file `Main.java`.

```
jshell> Main.twinPrimes(100).toArray()
$.. ==> int[15] { 3, 5, 7, 11, 13, 17, 19, 29, 31, 41, 43, 59, 61, 71, 73 }

jshell> Main.twinPrimes(100)
$.. ==> java.util.stream.IntPipeline$9@42e26948

jshell> Main.twinPrimes(100).toArray()
$.. ==> int[15] { 3, 5, 7, 11, 13, 17, 19, 29, 31, 41, 43, 59, 61, 71, 73 }

jshell> Main.twinPrimes(100).count()
15

jshell> Main.twinPrimes(2).forEach(x -> System.out.println(x))
```

```
jshell> Main.twinPrimes(2).count()
0

jshell> Main.twinPrimes(3).forEach(x -> System.out.println(x))
3
```

Task 2: Reverse String

Complete the method `reverse` in class `Main` that takes in a `String str` and returns the reverse of `str`.

```
static String reverse(String str) {
   return Stream.<String>of(str.split("")).reduce("", (a, b) -> b + a);
}
```

Save your `Main` class in the file `Main.java`.

```
jshell> Main.reverse("orange")
$.. ==> "egnaro"

jshell> Main.reverse("one two three")
$.. ==> "eerht owt eno"

jshell> Main.reverse("")
$.. ==> ""

jshell> Main.reverse("the quick brown fox jumps over the lazy dog.")
$.. ==> ".god yzal eht revo spmuj xof nworb kciuq eht"
```

Task 3: Counting Repeats

Define the method `countRepeats` in class `Main` that takes in a `List<Integer>` of digits of digits `0` to `9` and returns the number of occurrences of adjacent repeated digits. You may assume that there are at least two elements in the array.

```
static long countRepeats(List<Integer> list)
```

For example,

- the array {`0, 1, 1, 1, 1, 2`} has one occurrence
- the array {`0, 1, 2, 2, 1, 2, 2, 1, 3, 3, 1`} has three occurrences of repeated digits

Save your `Main` class in the file `Main.java`.

```
jshell> Main.countRepeats(List.<Integer>of(0, 1, 1, 1, 1, 2))
$.. ==> 1

jshell> Main.countRepeats(List.<Integer>of(0, 1, 2, 2, 1, 2, 2, 1, 3, 3, 1))
$.. ==> 3

jshell> Main.countRepeats(List.<Integer>of(0, 1, 2, 2, 1, 2, 2, 1, 2, 2, 1))
$.. ==> 3
```

Task 4: One-Dimensional Game of Life

A one-dimensional Conway's Game of Life is a population of cells represented in a list where each element represents a cell organism. Each cell may be alive or dead.

Let's assume a population of nine cells with only one living organism (marked `1`; `0` being dead) in the initial state (or generation `1`):

```
[0, 0, 0, 0, 1, 0, 0, 0, 0]
```

For each generation, a cell is alive or dead depending on its own previous state and the previous states of the two neighbour cells. We adopt the following rule:

- For each cell in the population,
  - If a cell is alive in one generation, it will be dead in the next.
  - If a cell is dead in one generation, but has one and only one live neighbour cell, it will be alive in the next generation.

Applying the above rules to the initial generation above, we obtain the next generation `2`:

```
[0, 0, 0, 1, 0, 1, 0, 0, 0]
```

Applying the rules again to obtain generation `3`:

```
[0, 0, 1, 0, 0, 0, 1, 0, 0]
```

Define a `generateRule()` method that returns the above rule in the form `UnaryOperator<List<Integer>>`.

```
static UnaryOperator<List<Integer>> generateRule()
```

Define the `gameOfLife` method that takes in a `List<Integer> list` as the starting population, the rule in the form `UnaryOperator<List<Integer>>` and the integer number of generations `n` of the game. The method returns a `Stream<String>` of the game of life for `n (> 0)` generations where a list element `0` is represented as `.` and `1` is represented as `x`.

```
static Stream<String> gameOfLife(List<Integer> list, UnaryOperator<List<Integer>> rule, int n)
```

```
jshell> Main.gameOfLife(List.of(0,0,0,0,1,0,0,0,0), Main.generateRule(), 4).
   ...>    forEach(x -> System.out.println(x))
....x....
...x.x...
..x...x..
.x.x.x.x.

jshell> int[] arr = new int[63]
arr ==> int[63] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  ... , 0, 0, 0, 0, 0, 0, 0, 0 }

jshell> arr[31] = 1
$.. ==> 1

jshell> List<Integer> list = Arrays.stream(arr).boxed().toList()
list ==> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  ... 0, 0, 0, 0, 0, 0, 0, 0, 0]

jshell> Main.gameOfLife(list, Main.generateRule(), 32).forEach(x -> System.out.println(x))
...............................x...............................
..............................x.x..............................
.............................x...x.............................
............................x.x.x.x............................
...........................x.......x...........................
..........................x.x.....x.x..........................
.........................x...x...x...x.........................
........................x.x.x.x.x.x.x.x........................
.......................x.............x.......................
......................x.x...........x.x......................
.....................x...x.........x...x.....................
....................x.x.x.x.......x.x.x.x....................
...................x.......x.......x.......x...................
..................x.x.....x.x.....x.x.....x.x..................
.................x...x...x...x...x...x...x...x.................
................x.x.x.x.x.x.x.x.x.x.x.x.x.x.x................
...............x...............................x...............
..............x.x.............................x.x..............
.............x...x...........................x...x.............
............x.x.x.x.........................x.x.x.x............
...........x.......x.......................x.......x...........
..........x.x.....x.x.....................x.x.....x.x..........
.........x...x...x...x.................x...x...x...x.........
........x.x.x.x.x.x.x.x...............x.x.x.x.x.x.x.x........
.......x.............x.............x.............x.......
......x.x...........x.x...........x.x...........x.x......
.....x...x.........x...x.........x...x.........x...x.....
....x.x.x.x.......x.x.x.x.......x.x.x.x.......x.x.x.x....
...x.......x.......x.......x.......x.......x.......x...
..x.x.....x.x.....x.x.....x.x.....x.x.....x.x.....x.x..
.x...x...x...x...x...x...x...x...x...x...x...x...x...x.
x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x
```

Submission (Course)

Select course: CS2030 (2023/2024 Sem 1) - Programming Methodology II

Your Files:

BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.