Assignment 6
Looping
4/5/19

Getting Started: Place work in "Problems" directory (i.e. "Assignment 6\Problems").

Note Well:

- In this doc, when a method description uses the word "we" or "our", it is talking about "this" object. In other words, the class (object) you are working on.
- Do the best you can. There is a lot of fun here. You may or may not be able to finish all the problems. Do the easier ones first.

Access Level Modifiers (Public, Private, Protected)

- All classes should be designated as **public**.
- All methods (defined in this doc) should be designated as **public.** Note that you may code your own "helper methods" that may have any access level modifier that you want.
- All instance variables should be designated as **private**.

General Hints:

See document *"Assignment 6 Hints.pdf"*.

Problem 1

Enhance "BrilliantNumber"

Take your BrilliantNumber from the previous assignment and enhance it. You probably do [not] want to remove any methods already in your class. You may find some of them useful for solving these problems.

"BrilliantNumber" objects will provide services for such things as isPrime, factorial, etc
so that (given a BrilliantNumber named "brilliant") we can easily do things like:

```
println("Number: " + brilliant.toString())
println("Is Prime: " + brilliant.isPrime())
println("Factorial: " + brilliant.factorial())
```

Instance Variables:

As defined in Assignment #5.

Assume BrilliantNumber's instance variable 'num' is a positive natural number (i.e. integer >= 1)

Constructor
As defined in Assignment #5.

Method Name: getNum
As defined in Assignment #5.

Method Name: setNum
As defined in Assignment #5.

Method Name: isPrime
Method Parameters: none
Return Type: boolean
Description:

Return true if our child "n" is prime. Else return false. A prime number is a number that cannot be formed by multiplying two smaller numbers. For example

7 --7 cannot be formed from multiplying two smaller numbers so is prime

10 --10 can formed from multiplying two smaller numbers (2, 5) so is not prime

For "10 = 5 * 2" we call 2 and 5 factors. One simple basic algorithm is to loop through candidate factors and check if our number "n" can be divided by any of the candidate factors. If it can be divided, that means it is not prime. If none of the candidate factors divides "n", then "n" is a prime. When you write down your algorithm (see hints), you'll be able to determine how to determine a set of candidate factors.

Hints:

- Jot down your algorithm on paper or a whiteboard before beginning

Method Name: factorial
Method Parameters: none
Return Type: int
Return Value: Return factorial for "n"
Description: Compute the factorial [without] using any Java Math methods.
notes:
A factorial is n*(n-1)*(n-2)...2*1
e.g.
4! = 4 * 3 * 2 = 24
The factorial of zero is defined to be 1

Method Name: isDivisibleBy
Method Parameters: an int, that is the candidate factor
Return Type: boolean
Description: Return boolean if our number is divisible by parameter value (i.e. the candidate factor)

Method Name: toString
Method Parameters: none
Return Type: String
Description: Return any “nice string” that you like. Suggestion: `return "" + this.getNum();`

Method Name: factorCount
Method Parameters: none
Return Type: int
Description: Return the count of all unique factors for our number
Hints: Reuse
For example:

Number	Unique Factors	Factor Count
6	1, 2, 3, 6	4
9	1, 3, 9	3
11	1, 11	2
12	1, 2, 3, 4, 6, 12	6
30	1, 2, 3, 5, 6, 10, 15, 30	8

Note: We include 1 and our number in the factorCount

Method Name: primeFactorCount
Method Parameters: none
Return Type: int
Description:
Return the count of all unique prime factors for our number.
If our number is a prime, return 0 for the count.
For example:

Number	Unique Prime Factors	Factor Count
6	2, 3	2
9	3	1
11	-	0
12	2, 3	2
30	2, 3, 5	3
31	-	0
210	2, 3, 5, 7	4

Hints: It can be useful to construct new objects in the loop action block, and sending the newly constructed object messages to get info you need. For example you may want to construct a new BrilliantNumber object.

Method Name: factorRatio
Method Parameters: none
Return Type: int
Description:
Return the rounded ratio of our factorCount divided by our primeFactorCount
NOTE WELL: If primeFactorCount=0, then return -1 for factorRatio.
Hints: See the previous assignment write-up for info on rounding. Think about reuse. Think about constructing new objects and sending messages to them to help simplify the solution.

Method Name: factorJKZ
Method Parameters: none
Return Type: int
Description:
If we are prime, then simply return our number as-is.
Otherwise, we'll calculate factorJKZ
It will take a few steps to show the calculation for factorJKZ.

1. If our number is even, let factorJKZ=2, otherwise let factorJKZ=1
2. Increase factorJKZ by our factorRatio (i.e. add "factorRatio" to factorJKZ)
3. If we are divisible by 3, increase factorJKZ by 3 (i.e. add 3 to factorJKZ)
4. If we are divisible by 5, increase factorJKZ by 5 (i.e. add 5 to factorJKZ)
5. If we are a square, double factorJKZ
6. If we are a cube, triple factorJKZ
7. If we are a palindrome, decrease factorJKZ by 1
8. Take the result of factorJKZ at this point. If this result (factorJKZ) is divisible by our number "num", then multiply factorJKZ by 0.

Return the final computed result for factorJKZ
Hints: Definitely think reuse.

Problem 2

Add a class called "NumberRange"

"**NumberRange**" objects will provide services for "range" type behavior such as:

```
NumberRange range = new NumberRange(1, 1000);
println("Number of primes between 1 and 1000: " + range.countPrimes());
```

Instance Variables

an int named "start"
an int named "stop"

Note: The instance variables "start" and "stop" are **included** in the range. In other words, if our range is from start=1 to stop=10, that means our range is:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Constructor
Constructor Method Parameters: two ints (first is for our start value, second is for our stop value)
Return Type: Not Applicable

Method Name: evenCount
Method Parameters: none
Return Type: int
Description: Answer count of even numbers in our range

Method Name: oddCount
Method Parameters: none
Return Type: int
Description: Answer count of odd numbers in our range

Method Name: countPrimes
Method Parameters: none
Return Type: int
Description: Count the number of primes numbers in our range

Method Name: sumPrimes
Method Parameters: none
Return Type: int
Description: Sum the primes in our range and return the sum
For example, if our range is 1 to 10, then the sum of primes would be:

2 + 3 + 5 + 7 = 17

Method Name: countDivisibleBy
Method Parameters: (int, int, boolean)
Return Type: int
Description:
If the boolean method parameter is true:
Then return the count of numbers in our range that are divisible by **both** of the "int" method parameters – e.g. AND.
Else return the count of numbers in our range that are divisible by **either** of the "int" method parameters – e.g. OR

Method Name: includes
Method Parameters: (int)
Return Type: boolean
Description: Return true if the range includes the method parameter

Method Name: size
Method Parameters: none
Return Type: int
Description: Return the size (length) of the range *(number of elements in the range)*

Method Name: countFactorRatios
Method Parameters: an int that is the threshold that will dictate if we include a number in the count
Return Type: int
Description:
For our range of numbers, determine the count of numbers that have a factorRatio that is greater or equal to the parameter value passed into this method.
Note: The definition of "factorRatio" may be found in BrilliantNumber.
Hint: Reuse

Method Name: sumFactorJKZs
Method Parameters: none
Return Type: int
Description: For our range of numbers, determine the sum of factorJKZs.
Note: The definition of "factorJKZ" may be found in BrilliantNumber.

Method Name: countFactorJKZs
Method Parameters: an int that is the threshold that will dictate if we include a number in the count
Return Type: int
Description:

For our range of numbers, determine the count of numbers that have a factorJKZ that is greater or equal than the parameter value passed into this method (the threshold). Return the count.

Note: The definition of "factorJKZ" may be found in BrilliantNumber.

Method Name: countPalindromes
Method Parameters: none
Return Type: int
Description:

For our range of numbers, determine the count of all of the numbers that are palindrome numbers and return the count.

Note: The definition of a "palindrome number" may be found in BrilliantNumber.

Method Name: getStart
Method Parameters: none
Return Type: int
Description:

Simply return the “start” instance variable value, e.g.:

```
public int getStart() { return this.start; }
```

Method Name: getStop
Method Parameters: none
Return Type: int
Description:

Simply return the “stop” instance variable value, e.g.:

```
public int getStop() { return this.stop; }
```

Method Name: intersection
Method Parameters: a NumberRange type
Return Type: NumberRange
Description:

Return a new NumberRange object that is the intersection between us (this) and the parameter (which is another NumberRange). For example if our range is 1-100, and the parameter is 91-200, then we would return a new NumberRange with a range of 91-100. If there is no intersection, return a range with start and stop both equal to 0.

Hints:

- You may optionally want to add a helper method “intersectsWith” that would take another NumberRange as a parameter and would answer true if the two intersect.
- Also remember the “includes” method you coded
- These Java methods may be handy to use:

```
Math.max(a, b);   //returns max of a and b
Math.min(a,b);    //returns min of a and b
```

Method Name: shrink
Method Parameters: int the factor that we use to shrink our range
Return Type: none
Description:
Shrink the range size (length) by the passed in factor (compute a new length)
Preserve our "start" position.
Adjust the "stop" position to accommodate the newly computed length
Example:

- Given a passed in parameter shrink factor of 3
- Take our length and divide it by 3 to compute new length (round result)

Hints:

- Remember you have already coded a method that will round for you
- Your may optionally want to add a helper method something like setLength that would take a "newLength" parameter and it would simply compute a new stop value using the newLength parameter.

Method Name: countSquareAndCubeNumbers
Method Parameters: none
Return Type: int
Description: Count the number of numbers that are both squares and cubes
Hint: Reuse
Just for curiosity: How many of these numbers are there below one million?

Method Name: largestPrimeGap
Method Parameters: none
Return Type: int
Description:

> Return the largest prime gap between any two primes within our range. For example if our range is 10-20, then the primes are 11, 13, 17, and 19. So the largest gap between any two primes in our range is between 13 and 17, meaning we have a gap of no primes between 13 and 17. In other words, 14, 15 and 16 are not primes. **NOTE WELL --** compute the gap to exclude the two primes. E.g. (prime2 - prime1 - 1). For our example (17 - 13 - 1) = 3. Our "gap" is 3. So we would return 3. If our range does not have any gap (e.g. zero or one primes), then return 0.

Method Name: largestPrimeGapPair
Method Parameters: none
Return Type: String
Description:
Return a String for the largest prime gap. Also see method description for "largestPrimeGap". The string is:

> prime1 + "-" + prime2

For example if our range is 10-20, then the primes are 11, 13, 17, and 19. So the largest gap between any two primes in our range is between 13 and 17. So we would return:

> "13-17"

If our range has more than one gap that has the largest size, return the first gap. For example for a range of 1-20, return "7-11", not "13-17". If our range does not have any gap (e.g. zero or one

primes), then return a string “0-0”.

Method Name: toString
Method Parameters: none
Return Type: String
Description: Return any “nice string” that you like.
Suggestion:

```
return ”” + this.getStart() + ” - “ + this.getEnd();
```