CS 211, ALL SECTIONS

LAB EXERCISE-7

DUE MONDAY, MARCH 11TH AT MIDNIGHT

The objective of this lab is see an example of intefaces, and the way they allow us to refer to classes abstractly. We create an interface, and use apply it to two different types of classes which nevertheless have similar properties.

OVERVIEW:

1. Write an interface, `Splittable`, which describes some type of entity which can be split into two halves.
2. Write the class `SplittableString` which is essentially a string with the characteristic that it can be split into a first half and a second half.
3. Write the class `Interval` which defines an integer interval on a number line.
4. Write the class `SplittableArray`, which is an `ArrayList` of strings which can be split into two smaller subarrays.
5. Download and use a tester module to ensure that your programs are correct.
6. Prepare the assignment for submission and submit it.

This is a lab exercise on which collaboration (including discussing the solution on Piazza or searching online sources) is explicitly permitted. It is still in your interest to understand and prepare your final submission yourself.

SPLITTABLE TASK:

(*2pts*) We understand the notion of splitting something into two parts. If we have a string, we can split it into a first half and a second half, so `"programmer"` can be split into `"progr"` and `"ammer"`; an interval on a number line can be split into two smaller intervals; and an array can be split into two smaller subarrays, each containing half of the elements in the list. So we will define an interface called `Splittable` which we use to describe a type of class which can be split into two halves. Any `Splittable` class is guaranteed to do three things: it has a size, it has a method to retrieve its first half, and it has a method to retrieve its second half.

Why are we making `Splittable` an `interface` instead of a `class` which we can derive from? Because the types of objects which are splittable may be conceptually very different. We can make a splittable array or a splittable string or a splittable interval, but these are relatively different types of classes: they may already derive from different existing classes, and even the notion of how splitting is supposed to work is different for each of the three. Instead of committing ourselves to a common base class, we only promise that each one has a notion of splitting.

Our `Splittable` interface should include three public methods:

- `public int size()` The size of the `Splittable`.
- `public Splittable firstHalf()` Returns the first half of the `Splittable`. If the halves are uneven in size (for example, an odd-length string), then we will adopt the convention that the first half is the larger of the two halves.
- `public Splittable secondHalf()` Returns the second half of the `Splittable`.

All method signatures should match the ones shown exactly.

SPLITTABLESTRING TASK:

(*2pts*) Now, we will write a `String` wrapper which is splittable. We will call the class `SplittableString` and it will implement `Splittable`. If our splittable string contains a string value, then the first and second half will be determined by substrings containing the first and second half of the string, respectively. Thus, if our string is `"noon"`, then our first half is `"no"` and our second half is `"on"`.

Note that since the `String` class itself is `final`, we are not allowed to create subclasses of `String`. However, we can create a wrapper instead - make a a class which contains a `String` member and which has a few methods which look and behave like `String`'s methods.

The class should implement the following `public` methods:

- `public SplittableString(String s)` Initialize the splittable string with the given string value.
- `public int size()` Returns the length of the string (this is one of the interface's methods).
- `public int length()` Returns the length of the string (in order to behave the same as `String`).
- `public char charAt(int i)` Returns the character at position `i` in the string.
- `@Override public String toString()` Returns the contents of the string.
- `public SplittableString firstHalf()` Returns the first half substring. If the string is odd-length, then the first half string is the larger of the two (for example, the first half of the string `"program"` is `"prog"`).
- `public SplittableString secondHalf()` Returns the second half substring. If the string is even-length, then the second half string is the smaller of the two (for example, the second half of the string `"program"` is `"ram"`).

Interval Task:

(*2pts*) An interval defines a portion of a number line. For example, the interval `[1, 3)` is the set of numbers from 1 to 3, including 1 but not including 3. The size of the interval is 2 (there is a distance of 2 between 1 and 3), and it can be split into two half-intervals: `[1, 2)` and `[2, 3)`. We will write an `Interval` class which implements `Splittable` and can return one of two half-intervals. The class should implement the following `public` methods:

- `public Interval(int a, int b)` Initializes the low (`a`) and high (`b`) bounds of the interval. The private variables which hold these values should be declared `final`, because we don't intend to change them once they're set.
- `public int low()` Returns the lower bound of the interval.
- `public int high()` Returns the upper bound of the interval.
- `public int size()` Returns the total size of the interval.
- `@Override public String toString()` Returns the bounds of the interval in the following format: `"[0, 10)"`.
- `public Interval firstHalf()` Returns the first half interval. Since the intervals are integer intervals, it's possible that one half-interval is larger and one is smaller, in which case the first half is the larger of the two (for example, if the interval is `[1, 10)` then the first half interval is `[1, 6)`).
- `public Interval secondHalf()` Returns the second half interval. Since the intervals are integer intervals, it's possible that one half-interval is larger and one is smaller, in which case the second half is the smaller of the two (for example, if the interval is `[1, 10)` then the second half interval is `[6, 10)`).

SplittableArray Task:

(*3pts*) Similarly to how we created a splittable string which let us split into two half-strings, we will now create a `SplittableArray` class which lets us split an array into two half-sized subarrays. We will extend this class from `ArrayList<String>`, so most of the array functionality will come pre-written for us, but we will also implement `Splittable`. Note that a class like this is one of the reasons we want to use interfaces: we would not have been able to derive from both `ArrayList` and `Splittable` if the latter had been a class. Also note that when we implement the `Splittable` interface, we will not need to implement a `size()` method because it already exists as part of the class!

The `SplittableArray` class should implement the following `public` methods:

- `public SplittableArray()` Initializes the class. Note that this constructor will need to do very little, and is possibly unnecessary (an empty default would be used instead).
- `public SplittableArray firstHalf()` Returns an array containing a subarray of the elements from the first half of the array.
- `public SplittableArray secondHalf()` Returns an array containing a subarray of the elements from the second half of the array.

Example:

```
> import java.util.Arrays;
> SplittableString str = new SplittableString("object-oriented");
> System.out.println(str.size());
15
> System.out.println(str.firstHalf());
object-o
> System.out.println(str.firstHalf().size());
8
> System.out.println(str.firstHalf().firstHalf());
obje
> System.out.println(str.secondHalf());
riented
> Interval i = new Interval(1,16);
> System.out.println(i);
[1, 16)
> System.out.println(i.size());
15
> System.out.println(i.firstHalf());
[1, 9)
> System.out.println(i.secondHalf());
[9, 16)
> System.out.println(i.firstHalf().size());
8
> System.out.println(i.firstHalf().firstHalf());
[1, 5)
> System.out.println(i.firstHalf().secondHalf());
[5, 9)
> SplittableArray arr = new SplittableArray();
> arr.addAll(Arrays.asList("the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"));
> System.out.println(arr);
[the, quick, brown, fox, jumped, over, the, lazy, dog]
> System.out.println(arr.size());
9
> System.out.println(arr.firstHalf());
[the, quick, brown, fox, jumped]
> System.out.println(arr.secondHalf());
[over, the, lazy, dog]
```

Testing:

Download the following:

- https://mason.gmu.edu/~iavramo2/classes/cs211/junit-cs211.jar (if necessary)
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/E7tester.java

This is a unit tester which is what we will use to test to see if your classes are working correctly.

When you think your classes are ready (you can try them even without the tester), do the following from the command prompt to run the tests.
On Windows:

```
javac -cp .;junit-cs211.jar *.java
java -cp .;junit-cs211.jar E7tester
```

On Mac or Linux:

```
javac -cp .:junit-cs211.jar *.java
java -cp .:junit-cs211.jar E7tester
```

In Dr Java:

- open the files, including `E7tester.java`, and click the *Test* button.

Submission:

Submission instructions are as follows.

1. Let *xxx* be your lab section number and let *yyyyyyyy* be your GMU userid. Create the directory `xxx_yyyyyyyy_E7/`
2. Place all of the `.java` files that you've created into the directory.
3. Create the file `ID.txt` in the format shown below, containing your name, userid, G#, lecture section and lab section, and add it to the directory.

   Full Name: Donald Knuth
   userID: dknuth
   G#: 00123456
   Lecture section: 004
   Lab section: 213

4. compress the folder and its contents into a .zip file, and upload the file to Blackboard.