Mastery I — Data Structures (T. I/22–23)

Directions:

- This mastery examination starts at 10am on Thursday October 20, 2022. You have until 12pm (noon) of the same day to complete the following *four* problems. Each problem has been broken down into multiple goals. Each goal will be graded on an all-or-nothing basis by a machine grader. No collaboration of any kind whatsoever is permitted during the exam.
- You are expected to use an IDE.
- **WHAT IS PERMITTED:** The exam is open- book, notes, Internet, Google, Stack Overflow, etc. The Internet can only be used in *read-only* mode.
- **WHAT IS *NOT* PERMITTED:** Communication/collaboration of any kind. Obviously, asking a question online is strictly *not* allowed.
- We're providing a starter package. The fact that you're reading this PDF means you have successfully downloaded the starter pack.
- Your Java code must *not* be in any package. That is to say, your files must not contain the statement `package .....`.
- To hand in your work, zip all the relevant `.java` files as `m1handin.zip` and upload it to Canvas before it closes.

Problem 1: Devil Spectrum (10 points)

The devil spectrum of a number $k \geq 0$ is the histogram of the digits of the value of 7^k encoded in a particular way. More specifically, the *devil spectrum* of a number k is an array `a` of length 10, where `a[i]` stores a number equal to how many times `i` appears in the digits of 7^k.

For example, because 7^{11} is equal to 1977326743, the devil spectrum of 11 is the array

```
a = [0, 1, 1, 2, 1, 0, 1, 3, 0, 1]
```

where we note that here:

- 7 appears three times so `a[7] == 3`,
- 1 appears once so `a[1] == 1`, and
- 5 doesn't appear at all so `a[5] == 0`.

Your Task: Inside a public class `DevilSpect`, write a public static method

```
public static int[] spectrum(int k) { ... }
```

that takes in an integer `k`, promised to be ≥ 0, and returns an array of length 10 that is the devil spectrum of `k`. Hence, as an example:

```
spectrum(11) // ==> [0, 1, 1, 2, 1, 0, 1, 3, 0, 1]
```

Additional test cases are in the starter file. *Hint:* Consider `Character.getNumericValue`.

Grading: Each run for $k \leq 10,000$ must finish within 1 second. There are 2 goals:

- The code works correctly for $k \leq 11$.
- The code works correctly for $k \leq 22$.
- The code works correctly for $k \leq 2000$.
- The code works correctly for $k \leq 10000$.

Problem 2: Shopping (10 points)

In this task, you will be implementing a few Java classes that simulate grocery shopping. The starter code for this problem contains the class skeletons and unit tests. You must *not* modify the unit tests, but you're free to add more tests as you see fit.

Your Task: Complete the implementation and pass all of the provided unit tests.
Here is the description of each class/interface:

- `Sellable` – is an interface modeling a product that is sold at a store. It contains two methods:
  1. `String getBarcode()` – returns the barcode string.
  2. `double getAfterTaxPrice()` – returns the price after tax is applied, if any.
- `FoodProduct` – is a class that represents a food product. It must implement `Sellable`. All food products are non-taxable. We require the following constructor:

```
public FoodProduct(String barcode, double price) {...}
```

- `NonFoodProduct` – is a class that represents a non-food product. It must implement `Sellable`. However, non-food products are taxable. The tax amount should be included in the price after tax. We require the following constructor:

```
public NonFoodProduct(String barCode, double price, double taxRate) {...}
```

  For example, if a non-food product has `price=1` and `taxRate=0.01`, its price after tax should be `1.01`.
- `Store` – is a class that maintains a collection of products in the store. There are three methods that you have to implement:
  1. `public void addSellable(Sellable a)` – adds a sellable item to the store. This method must throw `DuplicateBarcodeException` when you try to add an item with the same barcode again.
  2. `public Sellable findSellableByBarcode(String barcode)` – looks up a sellable by its barcode. This method must throw `ItemNotExistException` when there is no item with the given barcode in the store.
  3. `public int getSize()` – returns number of sellable items added so far.
- `Cart` – is a class that represents a shopping cart. We require the following constructor:

```
public Cart(Store store) {...}
```

  There are 4 methods that you have to implement here:
  1. `public void addItem(String barcode)` – adds an item with the given barcode into this cart. If the item doesn't exist, this method does nothing (no errors). There can be multiple items of the same product in the cart (e.g., banana added twice).
  2. `public double getTotal()` – returns the total price after tax of the entire cart.
  3. `public int getItemCount()` – returns the number of items in the cart. For example, if we have 5 bananas and 4 teapots, this method should return 9.
  4. `int getUniqueItemCount()` – returns the number of unique products in the cart. For example, if we have 5 bananas and 4 teapots, this method should return 2.
- `DuplicateBarcodeException` – a runtime exception thrown when you add products with the same barcode to a store.
- `ItemNotExistException` – a runtime exception thrown when you look up for an item that doesn't exist.

Grading: Your implementation should pass all of the provided unit tests. **Uncomment the unit tests to activate them.**

Problem 3: Enhancing `SLList` (10 points)

The starter code for this problem contains a rudimentary implementation of the singly-linked list (`SLList<T>`). The code uses a front sentinel. You will work to enhance it in this problem while retaining the front sentinel.

Your Task: You will implement the following methods in the `SLList` class:

- `public T[] toArray()` returns an array of all the elements of the list in list order. The length of the array must be exactly the size of the list (it doesn't include the sentinel node).
- `public void deleteIf(Predicate<T> p)` takes as an input parameter a predicate `p` and deletes all items in the list for which the predicate `p` returns true. Other than deleting items matching the criteria, everything else must remain untouched. This method should take time proportional to the length of the list.

  The predicate `p`, expressed using the `Predicate` interface, represents a higher-order function that takes in a type `T` parameter and returns a boolean (i.e., true or false). This is similar to other higher-order function (HoF) code we have done in the past. More specifically to this task, the interface has a method `public boolean test(T t)`. Hence, to test if an element `x` satisfies the predicate, we will look at the outcome of `p.test(x)`. This means inside your `deleteIf` method, you can call `p.test(x)`, where `x` is of type `T`, and it will return true or false indicating the outcome of the predicate.
- `public SLList<T> reversed()` returns a new `SLList<T>` that is the reverse of this list. For example, reversing `[3, 2, 7]` will give us `[7, 2, 3]`. This method should take time proportional to the length of the list.

Here are some examples:

```
class BanA implements Predicate<String> {
    public boolean test(String t) { return t.equals("a"); }
}
SLList<String> list1 = new SLList<>(List.of("J", "a", "v", "a", "S", "E"));
SLList<String> list2 = new SLList<>(List.of("J", "a", "v", "a", "S", "E"));

list1.deleteIf((String e) -> e.equals(e.toUpperCase()));
// list1 should become [a, v, a]
list2.deleteIf(new BanA());
// list2 should become [J, v, S, E]
SLList<String> list3 = list2.reversed();
// list3 is [E, S, v, J]
```

Ground Rules:

1. The only file you can modify here (aside from creating/writing tests) is `SLList.java`.
2. You are free to modify the `Node` class inside `SLList`.
3. You must *not* replicate the entire list using another collection. This means that you cannot, for example, make a temporary `ArrayList` or `LinkedList`. In other words, your logic must work directly on the `SLList`.

Grading: There are 3 goals:

1. Correctly implement the constructors and `toArray` as measured by what `.toArray` returns.
2. Correctly implement `deleteIf` as measured by what `.toArray` returns.
3. Correctly implement `reversed` as measured by what `.toArray` returns.

Problem 4: Range Iterator (10 points)

In Python, the `range` function has three forms:

- `range(n)`—a sequence of numbers $0, 1, \ldots, n-1$.
- `range(b, e)`—a sequence of numbers between `b` (included) and `e` (excluded). That is, it gives $b, b+1, \ldots, e-1$.
- `range(b, e, s)`—a sequence of numbers between `b` (included) and `e` (excluded), in increments of `s`. This means, for example, `range(1, 7, 3)` yields the sequence $1, 4$.

We will mimic the behavior of `range` in Java by writing a public class

```
public class Range implements Iterable<Integer>
```

so that it can be used as follows:

```
for (int i: new Range(10)) {
    System.out.println(i); // prints out 0, 1, ..., 9 on separate lines
}
```

Now this is only one example. Your class will support all three forms of `range`. Hence, it will have the following constructors:

```
public Range(int n) // corresponding to range(n)
public Range(int b, int e) // corresponding to range(b, e)
public Range(int b, int e, int s) // corresponding to range(b, e, s)
```

We promise to be nice to your code and will only test your code with `b` $\leq$ `e` and `s` ≥ 1.
Since the class implements `Iterable`, it must have a `public Iterator<Integer> iterator()` method.

Ground Rules:

1. The only file you can modify here (aside from creating/writing tests) is `Range.java`.
2. You must *not* use `IntStream`, `LongStream`, or `DoubleStream`.
3. You must *not* construct the entire sequence anywhere. This means that you cannot, for example, make an `ArrayList` of length `n` and return it.

Grading: There are 4 goals:

1. Programs compile clean and all three constructors can be invoked (i.e., won't crash).
2. Correctly implement `Range(n)`, as measured by the outcome of running for-each on it.
3. Correctly implement `Range(b, e)`
4. Correctly implement `Range(b, e, s)`