Object-oriented Programming (WIF/AAI)

Amra Ramic

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Exercise 06: Generics & Iterators

In this exercise we are going to modify our custom implementation of the *Set* data structure (see `StringSet` interface) from exercise 2. Our goal is to turn the set implementation for `Strings` into a generic implementation that can store objects of any type. As an example, we are going to store sweets in the generic set and build an iterator that provides sweets from our generic set.

Task 0: Preparation

Please create a new folder for this exercise in your own repository as you did in the previous exercises. At the end commit and push the modified files to your repository.

Task 1: Generic Set

You are given the solution of exercise 2 inside the package named `stringset` which implements a *Set* data structure for `Strings.`

a) Create a generic interface `Set` and its corresponding generic implementation `SetImpl` inside a package named `set`. Feel free to copy code from `StringSet` and `StringSetImpl` but modify it such that `Set` and `SetImpl` offer the same functionality but for generic types.
b) Create a Testcase `SetTest` analogous to the `StringSetTest`, which tests the generic implementation in the same way.
c) Make sure that the test `SetTest.testGenericSetOfStrings` finishes successfully.

Hint: Notice that the `StringSetImpl` relies on the `String`'s natural odering (`.compareTo`) to locate elements in the tree. In the same way, the generic set relies on its objects' natural ordering. If only there was a way to specify that the objects stored in the generic `Set<...>` must be `Comparable`...

Task 2: Sweets

Create a normal top-level class `Sweet` with the following attributes:

```
- name : String            // The name of a sweet
-sugar : int               // A score from 1-10
```

a) Implement the **natural ordering** of sweets such that the least sugary sweets come first. If two sweets have the same sugar content, the names determine their ordering.
b) Create a couple of sweets and store them inside an instance of your generic `SetImpl<Sweet>.`
c) Use the Debugger to verify that sweets are located where you expect them in the tree structure.

Task 3: Iterable Set

"After a long day, a couple of customers approach the set of sweets, hoping to indulge in some treats. However, the set of sweets hesitantly admits that it's unable to assist them directly. Instead, it suggests that they navigate through the assortment themselves. This proves to be a mistake.

"The set of sweets had painstakingly arranged all its delicacies inside a sophisticated tree structure, only to find chaos ensuing as customers freely roamed around, tearing open packs, rifling through shelves, and scouring the basement for treats. It's a disaster. The set of sweets realizes that once customers become accustomed to finding specific treats in certain places, it will be nearly impossible to tidy up, modify, or expand its refined tree structure.

In a bid to restore order, the set of sweets decides to hire an individual named I. Ator, tasked solely with preparing the next sweet and having it ready for service upon a customer's call of "next." The set of sweets grants I. Ator access to all the shelves. But from now on, I. Ator is the only one allowed to roam around the tree structure. And the clients? They better ask I. Ator to get the next sweet."

a) In your generic `SetImpl<..>`, implement the method `iterator()` that is prescribed by the `Iterable` interface.
b) Also implement an `Iterator` that visits all elements in the tree structure.
c) Make sure that the test `SetTest.testGenericSetOfSweets` finishes successfully.

Hint: The iterator needs to vist all nodes in the tree (=traverse the tree). During this process, it is necessary to remember which nodes are visited next. The list of nodes to be visited next is called the "frontier". You can use a java.util.List to maintain the nodes in the frontier. When the iterator is created, the frontier contains a single node: the root. Each call to next, removes a node from the frontier and adds its children back to the frontier."