Lab 5 - Implementing Generics and Iterators

due September 29nd at 23:55

Objective

Today's lab will help you get familiar with

- Write your own generic class
- Further understand abstract classes and interfaces
- implement a linked list
- learn about and implement an iterator

Abstract Classes:

Abstract Classes are classes that are partially implemented, but some methods are indicated as abstract. These abstract methods are not implemented and anyone extending an Abstract Class must implement these methods. If one or more abstract methods remains unimplemented in the inheriting class, then that class must also be abstract.

Generic Classes:

The idea of a generic is to write an algorithm/method without specifying the data type that will be used. For example, in previous labs you wrote methods that only worked for a specific type. With Generic Classes you could have implemented your program to work for a variety of data types. In previous labs, you have used generic classes, e.g., LinkedList and ArrayList. In this lab, you will implement a generic class.
The code below illustrates how you can create your own generic class. In the example two generic types are specified:

```
public class Entry<K, V>{

        private final K mKey;
        private final V mValue;


        public Entry(K k,V v){
            mKey = k;
            mValue = v;
        }

        public K getKey(){
            return mkey;
        }

        public V getValue()   {
            return mValue;
        }

        public String toString()  {
            return "(" + mKey + ", " + mValue + ")";
        }
}
```

Inheritance can then be implemented in the following way:

```
public class DifferentEntry<K, V> extends Entry<K, V> {

        /* CLASS BODY */
    }
```

Interfaces:

Java provides a useful tool for specifying functionality in the Class Library. Interface is a structure that defines a method set that must be implemented to meet a specific set of functionality. Often interfaces are provided in conjunction with another class that is designed to manipulate classes implemented by other developers.
For example, let's say there is a class called College that simulates any college campus. The developers of the College class also provide an interface called Student, specifying specific methods that all classes that

implement the interface must implement/provide. Then developers can create a LafayetteStudent that inherits from the College class and implements the Student interface.
This technique is called contract programming because the programmer is entering into a contract by implementing a defined interface.

Iterators:

Iterators are helper classes that implement the Iterable interface. The Iterator is an interface that allows you to process the elements stored in the data structure you are using. The iterator provides three methods:

- **`boolean hasNext()`** - Returns true if the iteration has more elements.
- **`E next()` -** Returns the next element in the iteration.
- **`void remove()`** - Removes from the underlying collection the last element returned by the iterator (optional operation).

These methods can be used to iterate through for each element in the collection. Note, these iterators allow direct alternate access to the underlying collection of data. Below are three examples showing how you can loop through the data, using a : for-loop, while-loop, and for-each-loop.

```
for(Iterator i = list.iterator(); i.hasNext(); ) {
      System.out.println(i.next());
    }

    Iterator i = list.iterator();
    while(i.hasNext()) {
      System.out.println(i.next());
    }

    for ( Integer val : list ){
      System.out.println(val);
    }
```

Assignment:

1. Create a class MyLinkedList which is a singly linked, non-circular list where each node only has one link next and the list has a head and a tail link (think about how you would implement the node). The MyLinkedList class also implements the Iterable interface. The following should be implemented as well:
2. Add the methods to the MyLinkedList class:
    a. iterator() to implement the Iterable interface. This method returns an instance of MyLinkedListIterator initialized appropriately.
    b. addFirst(<E> value) that adds a value to the front of the list
    c. addEnd(<E> value) that adds a value to the end of the list - Warning: remember to deal with the boundary cases!!!
    d. A get method which given an index returns the element.
3. Create a class MyLinkedListIterator that implements the Iterator interface.
4. Similarly to Lab 2, create a class called MyListStringContainer which uses your new data structure. It only needs to implement the following methods:
    a. addToBack
    b. addToFront
    c. Search for a substring (this should search for a String as a substring of a String in the list and return the first index of such a String if it is present, if not return -1. **In this case do not assume that the array is sorted**). You should write two versions of this method. One which uses an iterator and one which does not.
5. Write an ExperimentController so that it now evaluates the performance of both searches against each other. As usual, summarize the results of your experiments in your report, including appropriate graphs.
6. Unit test your classes.

Submission:

In addition to your code you must submit a report (in PDF format). Save your report in the project folder before you compress it and upload it. Details about the report can be found on the CS150 class moodle page.

Grading:

1. unit testing - 2
2. Linked List implementation - 3
3. Iterator- 2
4. Experiment Controller - 1
5. Style/commenting – 1
6. project write-up - 1