Homework #3

due Monday, September 25, 10:00 PM

This assignment will focus on the concepts of collections and using/implementing iterators.

1 Concepts

1.1 Collection ADT

Soon after Java was released, the need for a standardized collections framework became apparent. Accordingly, Java 1.2 introduced a standard collections framework. Each of the collection classes implements a standard set of operations.

For this assignment, you will write a class that implements the standard Java Collection interface which has the following methods (in addition to ones every object has):

- **size()** Return the number of elements in the collection.
- **isEmpty()** Return whether the collection is empty.
- **contains(Object)** Return whether the collection contains the parameter.
- **iterator()** Return an iterator over the elements.
- **toArray()** Return an array of all elements.
- **add(Object)** Add an element to the collection and return true.
- **remove(Object)** Remove an element from the collection and return true, or return false if it was not found.
- **containsAll(Collection)** Return true if every element in the parameter is also present in this collection.
- **addAll(Collection)** Add all the elements from the parameter to this collection, returning true if anything was added.
- **removeAll(Collection)** Remove all the elements in the parameter from this collection, returning true if anything was removed.
- **retainAll(Collection)** Remove all the elements in this collection that do not also occur in the parameter; returning true if anything was removed.
- **clear()** Remove everything from the collection.
Now, as it happens, some of these operations are more fundamental than others. For example, if iterators are working, it’s easy to write contains:

```java
public boolean contains(Object o) {
    Iterator<E> e = iterator();
    if (o==null) {
        while (e.hasNext())
            if (e.next()==null)
                return true;
    } else {
        while (e.hasNext())
            if (o.equals(e.next()))
                return true;
    }
    return false;
}
```

(This code is copyright ©2004 by Sun Microsystems.)

Notice how the code calls iterator() which is the (unimplemented) iterator method in the same class. As it happens, if iterators are working, then many methods can be implemented in terms of iterators.

For this reason, the Java collections framework includes AbstractCollection, an abstract class\(^1\) that does precisely this: implement everything using iterators, with that crucial part omitted: the iterator() method is defined as “abstract,” that is unimplemented.

Now size() is also not implemented, even though it would be perfectly possible (albeit inefficient) to implement that method with iterators. It is left abstract because presumably each collection has a more efficient way to keep track of the size than iterating through the whole collection. The clear method is similar: it is easy enough to implement using iterators:

```java
public void clear() {
    Iterator<E> e = iterator();
    while (e.hasNext()) {
        e.next();
        e.remove();
    }
}
```

(This code is copyright ©2004 by Sun Microsystems.)

Indeed the abstract class includes this implementation, but notes that a more efficient implementation should be possible.

The add(Object) method is implemented in the AbstractCollection class, but the implementation is not useful: it simply throws an exception that the operation (add) is “unsupported.” Indeed, there’s no way one can add an element using an iterator. So, again, extenders are encouraged to override this method with a proper implementation.

\(^1\)We assume you were taught abstract classes in CS 251.
1.2 Iterators

An iterator is an object that enables a programmer to traverse a container (e.g. a collection) without violating the data encapsulation principle (i.e., declare data as public). Java's external iterators have the following methods:

- **hasNext()** Return true if there still remain elements to be returned.
- **next()** Return the next element in the container. If there is no such element (in which case hasNext() should have returned false), then throw an instance of **NoSuchElementException**.
- **remove()** Remove the last element returned by next() from the container. Throws an instance of class **IllegalStateException** if next() has not yet been called or if the element has already been removed.

The **Iterator** interface itself is generic (covered in later assignments) but we will implement ours to work specifically with Notes. To access all the elements of a Song, and decide whether to delete them, one can write:

```java
for (Iterator<Note> it = c.iterator(); it.hasNext();)
    Note b = it.next();
    if (we don’t want element b any more) it.remove();
```

Java has a special syntax of “enhanced” for-loops to make it easy to use iterators. The shortcut:

```java
for (Note b : c) {
    ...
}
```

is short for

```java
for (Iterator<Note> _secret = c.iterator(); _secret.hasNext();)
    final Note b = _secret.next();
    ...
```

1.3 Fail-Fast Iterators

When using standard Java collections, (external) iterators become “stale” if the collection changes, except by using the iterator’s own **remove** method. It is not legal to use a stale iterator for anything, even calling **hasNext**. In other words, if you request an iterator and later add an element to the collection, then you are not allowed to use the iterator again. If you want an iterator, you must request a new one.

Implementors of Java’s standard collections are encouraged to provide **fail-fast** implementations of iterators which “usually” throw an exception (an instance of **ConcurrentModificationException**) when a stale iterator is used.

Typically, this ability is handled by adding an integer version stamp to every collection and iterator. The version should be incremented inside every method that modifies the collection. If an iterator notices that the version doesn’t match, it throws the required exception before performing any method. This is **not** part of the invariant: if the invariant fails, it is because the ADT implementation contained a bug; if the version stamp doesn’t match it is because the ADT was misused. Therefore, if the versions don’t match, do not check anything in the invariant checker.

The iterator version only changes if the collection was changed under its control (i.e., using **remove**).
1.4 Nested classes

A (non-static) nested class is interesting: it is considered to be “inside of” the object in which it was created and thus has access to all the (private) fields and (private) methods as if they were its own. This is not inheritance: it doesn’t actually get any fields or methods from the surrounding class, but it can access them. It is even more confusing if the nested class extends some other class. But we won’t do that for this assignment.

Nested classes are often used to implement iterators since an iterator needs access to private internals of the collection class. Within the iterator, you can use the fields of the outer class directly. If there are two fields with the same name in the outer and inner classes, you can refer to the one in the outer class by the syntax OuterClassName.this.outerClassField. For example, in this homework, the iterator’s invariant checker will need to call (not assert!) the _wellFormed in the outer class (instead of the one within the iterator), by including the call Song.this._wellFormed().

2 What You Need To Do

You need to implement Song, a Note array-based implementation of Collection by extending AbstractCollection and overriding the following methods:

- **clear()** Remove all elements in the collection.
- **size()** Return the number of elements in the collection.
- **add(Note)** Add an element to the end of the collection.
- **iterator()** Return an iterator over the elements in the collection.

Your class should use the same style as in the previous homework assignments: it should assert the correctness of the invariant before and after the two methods (clear and add) that change the object’s state, and at the beginning of the two methods (size and iterator) that merely observe the object’s state. The invariant should also be checked at the end of the constructor and when the iterator invariant is checked.

You need to implement the nested iterator, and must provide the standard iterator methods described in previous section (repeated below) as well as the iterator’s own _wellFormed() method. For an array-based collection (as what we have), the iterator can simply use an int variable _currentIndex to keep track of the index of the current element within the array. (Note: While _currentIndex will serve a similar function as it did in the previous assignment, its value and behavior will hold different meaning in this one.)

- **hasNext()** Return true if there still remain notes to be returned.
- **next()** Return the next note in the collection. If there is no such note (hasNext() should have returned false), then throw an instance of NoSuchElementException.
- **remove()** Remove the last note returned by next() from the collection. Throws an instance of class IllegalStateException if next() has not yet been called or if the note has already been removed. (You will need to keep track of this condition in the iterator by using a boolean variable _calledNext.)

The public methods of the iterator should implement “fail-fast semantics” by throwing an instance of ConcurrentModificationException if the version of the iterator doesn’t match the version of
the collection. Note that this check should occur within the iterator methods, after checking the invariant. So a version mismatch will be handled by the iterator’s public methods and not the wellFormed() method.

Finally, you need to modify the Jukebox driver from using the previous version of Song’s Sequence API to using its newer Collection API. The current code will not compile, you need to change it. Don’t add new public methods to the Song class to match the old API! We recommend using the ”enhanced” for-loop syntax mentioned above to keep things looking neat, but remember that behind the scenes it is using your own iterator implementation!

3 Files

The directory homework3.git repository contains the following files:

src/TestSong.java This class contains the test suites for the Song class and its external iterator class.

src/TestInternals.java Test driver for invariants of collection and iterator.

src/UnlockTest.java Test driver that can unlock all the tests without executing them.

src/edu/uwm/cs351/util/Song.java This is the skeleton file you should work with.

src/edu/uwm/cs351/util/Jukebox.java This is the driver from Homework 2, extended to support stretch/transpose.