Homework # 5 (v1.1)
due Monday, February 25, 10:00 PM

In this assignment we implement "endogenous" "doubly-linked" lists using "dummy" nodes. See the "Linked List Variations" handout to see what these terms mean. These lists will be used in ADTs that follow the standard Collection ADT. You will also make an ADT for hexagonal game boards that also follows Collection.

1 Implementing the Collection ADT

As seen in the documentation Collection has a large number of methods (at least 13 even before Java 8 added more). 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`:

```
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 `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:

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

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

[1] We assume you were taught abstract classes in CS 251.

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

The `add(...)` 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.

2 Concerning the HexBoard ADT

In Homework #3, we used a collection of hex tiles as the hex board. This worked fine for many operations, but when the user was adding or removing tiles, it required looping through the whole collection, which was tedious to program and inefficient to run.

Thus in this homework, we implement a HexBoard ADT that keeps the same collection interface, while adding a single new operation `terrainAt` which returns the terrain of the hex tile (if any) at the given hex coordinate. This new operation is more convenient to the client, allowing `HexBoardEditor`'s click listener to avoid any loops.

The `add` method of a hex board shouldn't work the same as a list: if you add the same hex tile again, it should make no change and return false. Furthermore "null" is not allowed to be added—your implementation should throw a instance of `NullPointerException`. Furthermore, the order isn't important. As explained below, you are expected to use a hash table implementation which means the order will be all but unpredictable. That behavior is fine.

At the implementation level, we will use Java's built-in `HashMap` class which implements the `Map` interface (See Chapter 5.7 in the textbook and Oracle's Map documentation). The implementation of `HexBoard` will thus have a map from hex coordinates to terrains without storing *any* hex tiles. The iterator will use the iterator of the map's "entry set" (use `entrySet().iterator()`) which is an iterator over entry objects (of type `Map.Entry<HexCoordinate,Terrain>`). The entry objects have `getKey()` and `getValue()` methods. These are the only methods of the entry objects that the nested `MyIterator` class will need. Since the hex board class won't store any hex tile objects, the iterator class will create them as needed. The `HexTile` class will be made explicitly a value class without object identity (we override `equals`, which should have been done right back in Homework #1).

As mentioned above, `AbstractCollection` implements most methods correctly, albeit inefficiently, using iterators. Others are implemented to simply throw exceptions. When you implement `HexBoard`, you will need to

- Override some methods due to Java (to avoid compiler errors);
- Override some methods for functionality (to avoid run-time errors);
- Override some methods for efficiency (to keep code running faster).

Many of the methods can be left as provided by the `AbstractCollection`: `addAll`, `containsAll`, `equals`, `hashCode`, `toString`, `isEmpty`, `removeAll`, `retainAll`, `toArray` and all the new Java 8 methods (`parallelStream`, `removeIf`, `spliterator`, `stream`). Neither are you expected to override the protected method `clone`.

But for the other methods of `Collection` (which are ...?), you should either override the implementation (giving the reason from the above list) or add a comment indicating why you don't need to oevrride it. Make sure to use the `@Override` tag; every public method must be documented *unless* it has an `@Override` tag.

When you implement the iterator for each collection, you are not expected to implement a data structure invariant, but should override the necessary methods. You are *not* required to implement "fail fast" semantics for the iterator for `HexBoard`. When implement the methods, you will use an iterator over the underlying structure. It will save you coding if you realize that if your code is expected to throw an exception under certain situations, it's perfectly acceptable if you let code that you call throw that exception. It makes little sense to catch the exception and rethrow the same exception.

3 Concerning the ADT Piece

Each Piece instance represents a different physical game piece. Each piece is on a team, has a "rank" (a simple enum) and a current location (if any). The class defines getters these three aspects, but a setter only for the position, since the first two are immutable characteristics of a "piece." A piece also has "prev" and "next" pointers which are used only by the `Piece.Collection` class explained below.

A "piece" has object identity—it is not equal to any other piece, even when that piece has the same characteristics. That's because a piece represents a physical game piece. On the other hand, we do have methods to print pieces as strings and create them from strings.

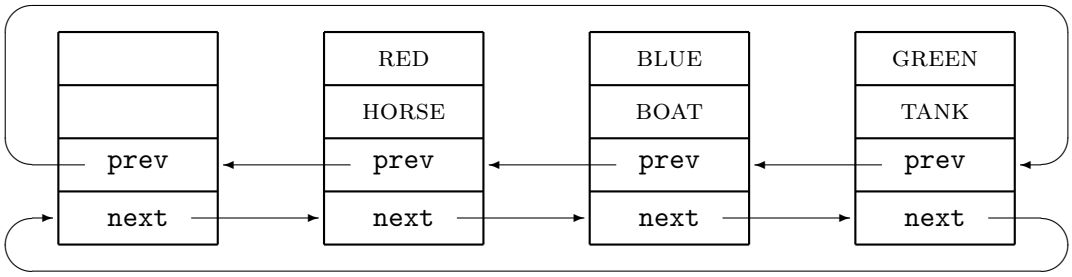
The Piece ADT is already implemented for you, but you should read the code to understand the implementation.

4 Concerning the ADT Piece.Collection

This ADT models a group of pieces, all on the board, or all captured by a single team, or all in reserve by a single team. Unusually, this ADT is implemented by a class (named "Collection") *nested* in the Piece class.

Since the data structure is endogenous, your code should permit a piece to be added to only one collection at a time, and null pointers can never be added (or `NullPointerException`). If an attempt is made to add it again to the same or another collection, an instance of `IllegalArgumentException` should be thrown. However, a piece collection should behave as a list: if a piece is added successfully, it is added to the *end* of the collection. The order will be visible when iterating.

The data structure used is *endogenous* (since the pieces themselves are connected), *cyclic* (the pieces are linked in a circle), *doubly-linked* (links both forwards and backwards) and uses a *dummy* node. When a piece is *not* in a collection, the previous and next pointers should be null.



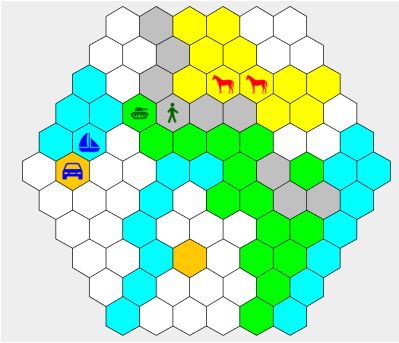
This data structure is redundant, and thus it has a non-trivial invariant. The pieces must be linked up in a cycle in which each card points to the next and previous pieces in the collection.

Finally, the size of the collection is stored redundantly (and does not include the dummy piece). It is part of this assignment to implement the invariant.

Unlike the iterator for hex boards, this ADT should also provide full "fail fast" semantics for iterators: an iterator should detect if it is out of "synch" with the collection. This is done using a "version" field which is incremented whenever the collection changes. When it is created, an iterator remembers what version it was created for and then checks this on every method call. When an iterator performs removal, it should update its version to follow that of the collection.

5 The Imaginary Game

These ADTs could be used for a game with pieces on a hexagonal board:



We don't actually define any rules for this game, but perhaps someone might make some sometime. This picture can be viewed by running the `Game` class class with "test/map.hex" and "test/sample.gam" as "Program Arguments."

6 Your task

Your task is to finish the `Piece.Collection` and `HexBoard` classes and their respective iterators. The `homework5.git` repository includes the following:

src/TestHexBoard.java Unit tests for the `HexBoard` class.

src/TestPieceCollection.java Unit tests for the `Piece.Collection` class.

src/TestInvariant.java A test suite for the invariant of `Piece.Collection`.

src/TestEfficiency.java A test suite for efficiency tests. If it takes more than a few seconds, you are implementing things inefficiently, or neglecting to override inefficient implementations.

src/edu/uwm/cs351/Piece.java The complete Piece ADT and incomplete `Piece.Collection` ADT.

src/edu/uwm/cs351/HexBoard.java Incomplete code for the hex board ADT.

src/edu/uwm/cs351/Game.java Display game boards.

test/ Some sample files.