Homework #3
due Monday, September 23, 10:00 PM

In this homework, you will use a collection class with an interface based on Java's library collection system.

1 Concepts

In this week, you will work with several concepts that don't have their own chapter in the textbook.

1.1 Interfaces

A Java "interface" is a special kind of "abstract class." A Java *interface* specifies a set of (public) operations ("methods" in Java) that an ADT might be expected to implement. An ADT implementation signals its adherence to the interface by adding "`implements` *I*" to the class header, where *I* is the name of the interface.

A Java variable (field, parameter or local) may have an interface as a type. Of course the variable is *not* an instance of this type, since interfaces are abstract—they don't provide any implementation. Instead they are instances of some class that implements the interface. In Java, it is considered better to use interfaces for the type of variables or method returns instead of concrete classes since it makes the program more general.

1.2 Generics

Many of the library classes are *generic* so that they work with different reference types (e.g., classes). In this homework, we will work with several generic interfaces and classes. If you fail to give the actual generic type parameter (inside angle brackets, e.g., `List<String>`), Eclipse will warn you that you are using "raw types." Raw types are not allowed for CS 351 homeworks, so make sure to take heed of the warnings.

1.3 Collections

The Java standard collection framework includes an interface `Collection` with a number of methods. See the textbook Figure 5.7, page 301 (3rd. ed. Fig. 5.6, p. 290), or look at the online documentation. There is an abstract implementation of this interface called `AbstractCollection`. This class provides a simple implementation for some of the methods. You will write a class that inherits from `AbstractCollection`. See the online documentation.

When you implement a class that inherits from an abstract class like `AbstractCollection`, you must decide which methods to override. Some reasone to override include:

1. The abstract class does not provide a functional implementation for this method.

2. The provided implementation is inefficient for your data structure.

3. The desired semantics differ from the provided implementation.

If none of these apply, you may choose to use the inherited method.

1.4 Iterators

`Collections`, as well as some other classes, provide iterators, which are an interface allowing access to each element, one-at-a-time, and once each. We implemented something similar as part of the `Sequence` ADT, but iterators are more powerful than such "cursors." The big advantages are (1) that one can have multiple iterators (whereas there's just one cursor for each Sequence) and (2) then changing the state of an iterator (e.g., advancing it) does not affect the collection itself. But since iterators are separate objects from the collections, if the collection changes, the iterator can go "stale." Implementations try to prevent the usage of stale iterators, throwing an instance of `ConcurrentModificationException` but the checks are not foolproof.

Iterators have three methods:

hasNext() Return true if there are elements that have not yet been accessed by this iterator.

next() Advances the iterator, accessing and returning an element that had not yet been accessed (now the current element). If there is no such element (in which case `hasNext()` should have returned false), then throw an instance of `NoSuchElementException`.

remove() Remove the current element (the one *previously* returned by `next()`). If `next()` has not yet been called, or if the last accessed element has already been removed, then this method throws an `IllegalStateException`, since there is no current element.

Each iterator moves separately through the container. Our iterator is a nested class, so it is able to access private fields and methods of the collection.

The `Iterator` interface is generic; the type of the elements in the collection needs to be specified. To access all the elements of a collection, and decide whether to delete them, one can write:

```
for (Iterator<T> it = c.iterator(); it.hasNext();) {
    T x = it.next();
    if (we don't want element x any more) it.remove();
}
```

Java has a special syntax of "for" loops to make it easy to use iterators. The shortcut:

```
for (T x : c) {
    ...
}
```

is short for

```
for (Iterator<T> _secret = c.iterator(); _secret.hasNext();) {
    final T x = _secret.next();
    ...
}
```

1.5 Fail-Fast Iterators

When using standard Java collections, 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. A "fail fast" implementation is not an absolute requirement.

2 Concerning the `Gallery` program

You will need to complete a `Gallery` class that implements a collection of `Painting` objects.

Note: there is a misleading comment in the invariant. It should say "there are no null elements," not "no null values in the data array."

3 Files

In the starting repository `homework3.git`, we provide the skeleton file for `Gallery.java` and also the `Painting` class. You need to complete the `Gallery` class, overriding methods to implement the desired interfaces and achieve the desired function. Do not add or change any fields in the classes.

There are questions in the comments that require answers. Make sure you answer them. *They will be graded.*

Use `TestGallery` and `TestInternals` to test your implementation, and `UnlockTests` to test your understanding.