Computer Science 2 Lecture 5
Java Collections Framework

02/11/2018

1 Prolog

In these notes the notion of "type" is often mentioned. That term refers to both interfaces and classes (and in Java, technically, enums and annotations, too). Other things that Java considers types are annotations like `@Override` and enumerations like

```
enum Color { RED, BLACK }
```

The term "subtype" can refer to an interface that extends another, a class that implements an interface, or a class that extends another.

2 Introduction

Java offers many classes in which to collect other objects. Here are some of their characteristics.

All the classes are generic, meaning you have to specify at least the element type you are storing. It doesn't mean that the collections are completely homogeneous, since when you specify a specific type for the type variable "E", you are really also allowing instances of all of its subtypes.

Refer to the Collection hierarchy in the collection-hierarchy diagram file. Keep in mind that one interface can *extend* another. This means that the former one imposes additional requirements on any implementing class.

Collection is a basic interface. It mandates methods for adding and removing elements, and computing the size, or number of elements currently in the collection.

Collection extends the Iterable interface. All collections offer two ways of iterating over their elements. The newer way is when you ask a Collection for its stream. You can then chain together operations that modify that stream of elements, accumulate them, and many other things. Streams will only be touched on in a limited way through later examples.

The traditional way of iterating is through a class that implements an Iterator. An iterator can be viewed as a cursor, or generalization of the array index. At any given point in time, an iterator is referring to one particular element in the collection from which it was created. Normally iterators are started at the "first" (whatever that means) element of the collection and they advance to each succeeding element until the end is reached. In Java, collections must be able to create Iterators for their clients through a method called, well, "iterator". As you might expect, an instance of Collection<SomeClass> will give you an instance of Iterator<SomeClass>. Iterators have only four methods.

- next() gives you the next element and advances the iterator further into the collection.
- remove() may not be implemented (more on that later), but if it is it gets rid of the next element. The new state of the iterator depends on the Collection class that made the iterator for you.
- hasNext() tells you if there are any more elements.

- forEachRemaining() is more like a stream operation in that it takes a "Consumer" to do something with the elements that the Iterator has not yet gotten to. We'll talk about this stream technique later when we discuss other interfaces' "forEach" methods.

The first and third operations mentioned allow you to iterate over the elements of a collection using a traditional "for" loop:

```
Collection< Foo > coll = .....;
for ( Iterator< Foo > iter = coll.iterator(); iter.hasNext(); ) {
    Foo f = iter.next();
    // Do something with f
}
```

Note the empty third clause in the "for" header. That's because next() automatically advances the iterator. (Not all collection frameworks do that, so beware when you learn a new one.)

However, because the Collection interface extends the Iterable interface, where that iterator() method comes from, there is a simpler for loop that you've also learned:

```
for ( Foo f: coll ) {
    // Do something with f
}
```

In fact the language standard specifies that the object after the colon must either be a primitive array, or an instance of a class that implements Iterable.

For example ArrayLists are iterable. Follow the inheritance chain: ArrayList<E> extends AbstractList<E>, AbstractList<E> extends AbstractCollection<E>, AbstractCollection<E> implements the interface Collection<E>, and finally Collection<E> extends the interface Iterable<E>

3 The Hierarchy

Refer again to the graphics in the collection-hierarchy file **collection-hierarchy.pdf**.

See how each subtype of Iterable adds more and more functionality as you go down the hierarchy.

All Collection subtypes describe one-dimensional collections of elements. That is implied by the fact that they are Iterable. Behaviorally, we can classify collections as being in one of threee categories:

1. Unorderable collections: You never know what order an iteration or stream will reveal.
2. Manually orderable collections: Elements have order, but it can be changed.
3. Automatically ordered collections: Data structure mandates an order.

HashSets belong to the first category. Lists belong to the second. TreeSets and PriorityQueues belong to the third. (Technically, all Queue implementers and Stack belong to the third group, but their underlying class methods may have capabilities to manually order their instances.)

Here are essential highlights of many of the types in the Collection hierarchy. Slightly more detail is shown in the hierarchy diagram.

- Sets do not allow duplicates.
- HashSets (extend Set) are implemented using hash tables.

- SortedSets (extend Set) maintain their elements in sorted order.
- NavigableSets (extend SortedSet) maintain their own element order, based on their natural ordering (Comparable) or a Comparator object, discussed later.
- TreeSets (implement NavigableSet) are implemented using *red-black trees*. This design is similar to the binary search tree, but a red-black tree has a lot more in it to keep the tree balanced so that it can achieve O(log N) performance.
- Lists allow access to elements via indices.
- ArrayLists internally use contiguous memory like primitive arrays do.
- LinkedLists use linked nodes.

For complete details, see the type's javadoc on line.

4 Code Examples: Person

The Person classes, versions 1-3, deal with creating Person instances, putting them into a collection, and ordering them one way or another. They demonstrate various ways of doing the same thing in Java. There are comments in the code. (Person versions 5-6 should not be looked at yet.)

The TreeSet, being a sorted set, depends heavily on the Comparable or Comparator class to do its job. The HashSet needs Object's hashCode method, which should always be overridden. Recall that hash codes are not unique, and therefore to make sure the right element has been found, an equality test also has to be performed. Therefore the equals method is also important here.

In the Person examples the equals method was defined from the very first version, becuase it is also very important that equals return true when compareTo returns 0, and that equals return false when compareTo returns something non-zero.

5 The Map Hierachy

Look at the Map hierarchy on the second diagram in collection-hierarchy.pdf.

There is a separate hierarchy for Map classes. The basic idea is that of a "Dictionary". You can view maps as lists of pairs of values, but it's more accurate to think of a map almost like a function that maps one value to another. We call the "input" value the "key", and the "output" value just "value". Thus the conventional type parameter names for these doubly generic types, K and V.

Recall that lists, or arrays, are a simple form of a map. Their keys are small integers (indices) and output values are whatever type was specified in the list's declaration. So a List<E> can be viewed as an implementation of Map<Integer,E> ... sort of.

Basically the Map hierarchy is a simplified version of the Collection hierarchy. Just remember that those variations in how the Collection classes worked now apply to the keys, not the values.

Version 4 is similar to the earlier versions, but it uses different kinds of maps to store Persons. The key is actually the Map, and the value associated with the key is a phone number. For TreeMaps the key has to be comparable, like TreeSets. For HashMaps the key has to be hashable, like

HashMaps. The hash code was specially chosen so that you could predict the order that the Persons will appear in the HashMap.

The Person version 5 example changes the Person class so that, instead of storing names and ages, it only stores a name. A map is used to look up the Person's age. You should see many similarities between this code and the previous Person examples.

6 Utility Classes

Two other classes are provided in this framework that contain many static methods. They are definitely worth review. Their names are Collections and Arrays (note the plural).

Collections contains methods that search, rearrange, and enhance various Collection objects. Arrays has methods for converting a primitive array to a List. Beyond that, its main purpose is to provide primitive arrays much of the same functionality as Lists. See their javadoc for details.

7 Optional Topics

There are times when a high-level type in the hierarchy promises something that a low-level class fails to deliver. Here are some examples.

- methods in interfaces that are allowed to raise UnsupportedOperationException
  Principle of "Design By Contract" states that this could be viewed as a violation of an interface's *contract*.
- why it's not SortedSet< T extends Comparable<T> >
  2 answers:

  1. the allowance of an external Comparator means that T does not really have to be Comparable.
  2. *covariance* – restricting the element type in a subclass – is bad. Sets allow any type element. Where does SortedSet, its descendant, get off restricting its elements? An analogy follows.

7.1 Practical metaphor for contracts and type covariance.

Let's say you hired Jake to rebuild your kitchen. You both signed a contract stating what was to be done, and what it will cost. Under cost, the contractor Jake wrote that the "type" of payments he accepted were cash, check, or credit card.

Then let's say that Jake hired a SUBcontractor Tony to actually do the work on your kitchen. Jake said that you could treat the subcontractor the same way you'd treat Jake. When the job was done, you offer Tony your credit card, but he says, "Sorry, I only take cash."

The contract has just been violated. Imagine a Payment interface, with implementing classes Cash, CreditCard, and Check. Your contractor provided you with something like

```
class Contractor {
    void acceptPayment( Payment p ) { ... }
    :
    :
}
```

However, along comes

```
class SubContractor extends Contractor {
    @Override
    void acceptPayment( Cash p ) { ... }
    :
    :
}
```

and you get an instance of that class instead.

```
Contractor myContractor = new SubContractor( "Tony" );
```

Using your "is-a" logic (formally known as the Liskov Substitution Principle), you expect nothing has changed, so you call

```
myContractor.acceptPayment( creditCard );
```

and "the system blows up". To prevent this in software, Java and many other languages do not allow you to further constrain, or "narrow" the types of any incoming values.

This is one reason why you have

```
Set< T >
```

and you can't have

```
SortedSet< T extends Comparable< T > >
```

because that adds a new constraint in the subclass.

In type theory this is known as "covariance". You are narrowing both the type and a parameter to the type simultaneously.

7.2 Streams and Lambda

Recall that Collection implements Iterable. The Iterable interface also provides another method:

```
void forEach( Consumer< E > )
```

Let's use Consumer as a way to introduce a very special group of interfaces called "functional interfaces". They are so called because they have only one abstract method, their "function". We can therefore look at classes that implement these interfaces as wrappers around simple functions. They are in the package java.util.function. For the case of Consumer:

```
interface Consumer< E > {
    void accept( E element );
    default Consumer<E> andThen( Consumer<E> after ) { .... }
}
```

Don't worry about andThen in this course. Since it has a default implementation, Consumer indeed qualifies as a functional interface.

Let's convert a normal collection loop into an application of forEach. We will print the elements of a collection of Integers, one per line.

```
Collection< Integer > numbers = .... ;
for ( int n: numbers ) {
    System.out.println( n );
}
```

How would you use forEach to do this? You're not going to like it ... at first. Create a class that implements Consumer.

```
class IntPrinter implements Consumer< Integer > {
    public void accept( Integer n ) {
        System.out.println( n );
    }
}
```

then apply it.

```
Collection< Integer > numbers = .... ;
numbers.forEach( new IntPrinter() );
```

That's a lot of work. Why stop using loops? Well, Java does allow some shorthand techniques. None of them make you explicitly declare a named class anymore. Let's go from the one that requires the most typing to the one that requires the least. Assume the numbers variable is already initialized.

```
numbers.forEach( new Consumer< Integer >() {
    public void accept( Integer n ) {
        System.out.println( n );
    }
});

numbers.forEach( ( Integer n ) -> { System.out.println( n ); } );

numbers.forEach( n -> System.out.println( n ) );

numbers.forEach( System.out::println );
```

These forms work because the Java compiler has gotten pretty clever at inferring what it needs to build the complete class from just a small amount of code. Now maybe it's starting to look easier than writing the loop. Maybe?

Note that, depending on what code you need to write in the body of the accept method, you may or may not be able to simplify the argument of forEach as much as was done here. When the body becomes too complicated, going back to plain old loops is the better choice. (But still, use the newer loop style whenever possible.)

The last three examples are called lambdas. They are named for a seminal computer science term that basically describes an unnamed function. Notice that those expressions do look like simple functions more than they look like classes.

Back to the Person examples, version 6 shows version 4 where all those small classes have been replaced with lambda expressions within the main code. You will probably think that it is worth it sometimes, but perhaps not others.

In this course, unless you're explicitly told otherwise in an assignment, you can write whatever form with which you're most comfortable.

As briefly stated at the start of these notes, Collections have a stream() method, which returns something called a "Stream" on the collection. Streams can also perform the forEach method:

```
numbers.stream().forEach( System.out::println );
```

Why bother making a Stream? Because Streams have lots of other methods that work with functional interfaces. We'll only introduce one other one: filter. The filter method takes a different functional interface called Predicate.

```
interface Predicate< E > {
    boolean test( E element );
}
```

filter() removes elements from the stream that fail the test. Let's look at a little example that uses these two methods: TryForEach. It generates the first few Fibonacci numbers, prints them, then using a stream filters out the even numbers and prints what's left.

7.3 Ranges in Java

If you want to do a traditional counting loop in Java that uses forEach, look at the BuiltInRange example. For the curious, another code example is the class Range that was designed to mimic as closely as possible the range class in Python. It is iterable.