Collections and Iterators

Based on the notes from David Fernandez-Baca and Steve Kautz
Based on The Java™ Tutorial
(http://docs.oracle.com/javase/tutorial/java/)

Bryn Mawr College
CS206 Intro to Data Structures

Collections

- Collection (container): an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
  - a poker hand : a collection of cards
  - a mail folder : a collection of letters
  - a telephone directory : a mapping of names to phone numbers

Collections Framework

- A collections framework is a unified architecture for representing and manipulating collections.
- All collections frameworks contain the following:
  - Interfaces: abstract data types that represent collections.
  - Implementations: concrete implementations of the collection interfaces. They are reusable data structures.
  - Algorithms: methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Choosing a Collection…

- Is the collection bounded in size?
- Does it allow duplicates?
- Does it allow null elements?
- Is the collection linearly ordered?
- Do elements have *multiple successors*, like a directory tree?
- Is there *no ordering* at all, like a mathematical set?
- Is there *random access* to all elements? E.g., can
- you go just as easily to item 23 as to item 133,056?

Choosing a Collection… (cont.)

- Is the access sequence restricted somehow? Common restrictions are
  - **First-in first-out (FIFO):** We can only access the "oldest" element. Such a data structure is called a queue.
  - **Last-in first-out (LIFO):** We can only access the "newest" element. Such a data structure is called a stack.
  - **By priority:** We can only access the element of "highest priority". Such a data structure is called a priority queue.

Key Methods of Collection<E>

- boolean add(E item)
- int size()
- boolean contains(Object obj)
- Iterator<E> iterator()
- boolean isEmpty()

Methods of Iterator<E>

- boolean hasNext()
- E next()
- void remove()

We can access the elements by repeatedly calling next() until hasNext() returns false. All elements will be got exactly once.

No guarantees about ordering: if you iterate over the collection again, you could get the elements in a different order.

If you try to call next() when hasNext() is false, you get a NoSuchElementException.

Example

```
Collection<String> c = new ArrayList<String>();
c.add("Huey");
c.add("Louie");
c.add("Dewey");
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    System.out.println(s);
}
```

- This code uses the fact that ArrayList implements the Collection interface.

Foreach Loops

- Foreach loop: the pattern of iterating through the elements of a collection.
- For instance, suppose c is of type Collection<String>, then the code

```
for (String s : c) { System.out.println(s); }
```

is, literally, translated by the compiler into:

```
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    System.out.println(s);
}
```

The AbstractCollection<E> Class

- AbstractCollection<E> is a generic abstract class that implements of **all** the methods of Collection<E>, except size() and iterator().
- Serves as a starting point for concrete implementations of Collection.
- Some methods of Collection are optional; i.e., they are not required to be implemented by an implementing class.
- Optional methods in AbstractCollection are implemented in a simple fashion: Throw an UnsupportedOperationException.

```
public boolean add(E o) {
    throw new UnsupportedOperationException;
}
```

The AbstractCollection<E> Class

```
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;
}
```

Practice: an Array-Based Generic Collection

- A simple array-based implementation of Collection<E> called FirstCollection<E>.
- Structure:
  - A data array, which stores items, and a size field, which indicates how many slots of data are being used.
  - Two constructors:
    - One takes an initialCapacity argument which specifies the initial length of data.
    - The default constructor initializes data to DEFAULT_SIZE (= 10)
- Key methods:
  - boolean add (E item) : put the new item in the next available slot at the end of the data array.
  - int size()
  - Iterator<E> iterator()

FirstCollection<E>: Basic Structure

```
public class FirstCollection<E> extends AbstractCollection<E> {
   private static final int DEFAULT_SIZE = 10;
   private E[] data;
   private int size;
   public FirstCollection() {
      this(DEFAULT_SIZE);
   }

   public FirstCollection (int initialCapacity) {
      data = (E[]) new Object[initialCapacity];
      size = 0;
   }
   …
}
```

FirstCollection<E>: Adding an Element

```
public class FirstCollection<E> extends AbstractCollection<E> {
   …
   public boolean add(E item) {
      checkCapacity();
      data[size++] = item;
      return true;
   }
   private void checkCapacity() {
      if (size == data.length) {
         data = Arrays.copyOf(data, data.length * 2);
      }
   }
   …
}
```

FirstCollection<E>: Iterators

- Recall that Iterator<E> is an interface with three methods:
  - boolean hasNext()
  - E next()
  - void remove()
- hasNext() and next(): go through all the elements of a collection exactly once by instantiating an iterator for it and then repeatedly calling next() until hasNext() returns false.

FirstCollection<E>: Iterators – remove()

- Removes the element returned by the last call to next(). Once an element has been removed, remove() cannot be called again until another call to next() has been made.
- If remove() is invoked at an illegal or inappropriate time — i.e., before another call to next() — then an IllegalStateException should be thrown: We are violating the class contract by invoking the method when the object is not in the right state.
- The details of the iterator implementation will be hidden from the clients in a private **inner class** within FirstCollection called MyIterator.

Inner Class

- Inner classes increase encapsulation: An inner class is associated with an instance of its enclosing class and has access to other members of the enclosing class, even if they are declared private.
- It cannot define any static members itself.
- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.
- To instantiate an inner class, you must first instantiate the outer class.
  - OuterClass.InnerClass innerObject = outerObject.new InnerClass();

FirstCollection<E>: Iterators – remove()

```
@Override
public Iterator<E> iterator() {
   return new MyIterator();
}
```

- Placing MyIterator within FirstCollection gives it access to internal knowledge of a collection. In particular, this lets it "know" that it must run through a data array.
- Further, we can create multiple instances of this class, i.e., multiple iterators for the same collection, each with its own state.

FirstCollection<E>: MyIterators

The MyIterator class centers around a cursor variable, which marks the current position (state) of the iterator.

- cursor is initialized to 0.
- next() returns the item in position cursor of data and then increments cursor.
- hasNext() is true if cursor < size.
- remove() must remove the element just before the cursor, because that's the one that was returned by the previous call to next(). To ensure that remove() is not called before next(), FirstCollection maintains a state variable **canRemove**, which is only true if next() has been invoked.

FirstCollection<E>: Iterators – remove()

remove() proceeds like this:

- It shifts elements beyond cursor down by one and decrements size,
- It decrements cursor, so that the subsequent call to next() is handled correctly.
- It sets canRemove to false to disallow another deletion until next() is invoked again.

FirstCollection<E>: MyIterator

```
private class MyIterator implements Iterator<E> {
    // index of the next element to be returned by next()
    private int cursor = 0;
    private boolean canRemove = false;

    @Override
    public boolean hasNext() {return cursor < size; }

    @Override
    public E next() {
        if (cursor >= size)
            throw new NoSuchElementException();
        canRemove = true;
        return data[cursor++];
    }
```

FirstCollection<E>: MyIterator

```
    @Override
    public void remove() {
        if (!canRemove) {throw new IllegalStateException();}
        // delete element before cursor.
        //Note that must have cursor >= 1
        for (int i = cursor; i < size; ++i) {data[i - 1] = data[i];}

        // null out the vacated cell to avoid memory leak
        data[size - 1] = null;
        --size;
        --cursor;
        canRemove = false;
    }
}
```