CSC 115 - Lab#2 (review)

During Lab number 2, you will have done the following:

Created a Java class that has a non-simple data field

We used the example of the `Numbers` class that was specified in the specifications. During the design process, you will have discussed what information is necessary to keep track of a list of integers and what data types to use. Since most of us have had experience with the array data type in both C and Java, this seems like a good choice for the main *attribute* or data field. Some discussion points you likely discussed:

- What data type can we use to store a collection of integers?
- Given the specifications that require insertion and deletion, how will we *maintain* the datatype(s) we choose for the main data fields?

Note that there is no single *right* way to design a new class and to handle the private data fields. However, there are two recommendations we like to consider when making these design decisions:

- Does our design remain stable during any of the actions that are performed on the data fields?
- Does the effort to maintain stability seem to be the *simplest* solution?

Note that the answers to these questions are not always clear at the beginning, but during the programming phase, we may discover that we need to make some design changes that are more stable and/or more simple.

It was a general consensus that we:

- Use an integer array data type as our main data field.
    - ⋆ We decided that we would start with an empty array of some initial size, and then enlarge it as needed if becomes full (after many `add` calls).
    - ⋆ We decided to insert elements into the array from left to right, so that all the added integers were indexed from 0 to the number of elements -1.
    - ⋆ Noting that if we maintained a size (or count, or depth) variable that is always set to the current number of elements, that value also acts as the index number to the next *empty* spot in the array.
    - ⋆ Initially, we delayed the discussion about how to handle `removed` items.

- Besides the `int[]` data field to store the number of elements, we talked about how big to make the initial empty array.
  - ⋆ We can simply choose a *literal* number, like 10 or 6000, depending on how we envision the use of this class.
  - ⋆ **programmer tip:** It is good practice to stay away from using literal values in programming, opting instead for a *constant*. The constant is declared near the top of the source code; within the body of the code, it is used in place of the literal value. When it needs to be changed, there is only one place that is updated.
  - ⋆ Constant values in Java are defined by the key word `final`[*] which means that they are not *variable*, and also by the word `static`,[†]which means that there is only one memory location stored for that value.
    - → In an object-oriented language, we can expect that multiple users may create several `Numbers` lists, each of them having a unique array to store the numbers
    - → However, since the starting size of each array is the same, we really only need to store a single integer for that, hence the word `static`.

    Since the constant is not really a data field, we often separate it from the data fields by a single line.

At this point, we will have created the shell of our program, with the data fields we anticipate using.

```
public class Numbers {

  private static final int INIT_SIZE = 3;

  private int[] numbers;
  private int count;

  public Numbers() {
  }

  public void add(int num) {
  }

  public void remove(int num) {
```

[*]Making a 'variable' 'final' is an oxymoron, However, on a computer, a variable is actually the name of a fixed storage location. The `final` key word makes that location non-updateable. There is a level of efficiency in the handling of a storage location that cannot be changed.

[†]For every object created, separate storage is set aside to contain each of the data fields *for that object*. A *static* variable has only one storage location, no matter how many objects are created. However, any object that is created may access that value.

```
    }

    public String toString() {
      return null; // placed to allow compilation.
    }

    public static void main(String[] args) {
    }
  }
```

Note that since the specifications are provided, the comments above each method header can be added by copying them directly. They are not provided in the above code for space reasons, but they should always be part of the source code. Placing them at the beginning makes it much easier to program, since we only need to follow the directions specified in the comments.

Added the methods that act on the data fields

A little bit of theory that may help

In the example, the method headers were strictly set out, and we filled in the programming details for each of them. However, we have some design considerations with respect to the actual data fields; we want them to remain stable and predictable at all times. When we begin *implementing*, (writing the code for) the individual methods, we must make sure the data fields are still stable, in other words, we have to clean up after the method.

Using this analogy for a little longer: it is easier to *put everything back in place* if we actually have a description of what "in place" looks like. One way to guarantee this is to have a clear and concise *invariant statement* of the data fields. Rather than give a clinical description of an invariant statement, we use an example for the `numbers` and `count` data fields:

Suppose we decide (during the design process) that:

- We will use an array to store the numbers.
- We will add items to the array from left to right.
- When taking items out of the array, we could leave an empty spot, but since the number list is not required to be sorted, we fill the emptied spot with the last item in the array.

The following statements are pretty obvious truths that we can derive from the above design decisions.

- The array will contain the elements from left to right.
- The count variable will contain the correct value for the number of elements in the list.
- The count variable will also be the index of the next available cell in the array.

These statements together make up the invariant, which we can apply when we implement (write the code for) each method, because:

(1) Before the first statement of the method body, we know the invariant to be true.
(2) Before the final closing brace of the method, we tidy up so that the invariant is still true.

Some of you may have noticed that we have no proof that statement 1 above is true, and you are right. However, if we set that up in the constructor method, then we can logically prove that our algorithm is correct by mathematical induction:

(1) The constructor sets up the data fields so the invariant is true. (the base case).
(2) Suppose that at some invocation of a method, the invariant is true (the induction hypothesis).
(3) If, at the end of the method, the invariant is still true, then the invariant holds for *all* invocations of the method.

Back to the practical part

The order in which we programmed the methods varied depending on the lab, but a reasonable order follows:

Constructor

As per the invariant, we need an array that contains no elements. and a `count` variable that is set to 0. By default, `count` is set to 0, so we don't have to do anything. However, `numbers` is `null` by default. Since `count` must also be the index of the first available empty spot, we actually need an array that has a cell at index 0. So, the constructor must initialize the array to be ready to accept the integer elements.

```
/**
 * Creates a collection of integers that is initially empty.
 */
  public Numbers() {
    numbers = new int[INIT_SIZE];
  }
```

will do that for us.

toString

The `toString` method should always be implemented first. Once it is complete, we can check each method incrementally to make sure that everything is working as it should. Given the invariant, we only have to print out the elements from index 0 to index `count`-1.

```
/**
 * Formats a string representation of the collection, which can be
 * in any order.
 * The numbers are separated by commas, and enclosed in curly braces.
 * An empty collection is represented by curly braces with nothing
   between them.
 * For instance, if 1,6,5,3,-7,3 are in the collection, then the string
 * <pre> {1,6,5,3,-7,3}</pre> in <i>any</i> order is returned.
 * If the collection is empty, then <code>{}</code> is returned.
 * @return The collection of numbers, in the format described above
 */
public String toString() {
  StringBuilder s = new StringBuilder(count*5);
  s.append("{");
  for (int i=0; i<count-1; i++) {
    s.append(numbers[i]+",");
  }
  if (count > 0) {
    s.append(numbers[count-1]);
  }
  s.append("}");
  return s.toString();
}
```

Note the above example uses the *javadoc* commenting style. This is the exact commenting that produced the specification document. There are some html tags used in this style, so it is helpful to know a bit about html. This is provided for your interest only. You are not required to use this particular style, but if you like to learn by example, you are welcome to try this.

Once this method is complete, we can test the constructor and the `toString` method to make sure that the code is good up to this point. In the main method (which we create to test during programming), we add the following to see if the output is as expected. If it is not, then we are not finished and should not proceed until we fix the errors.

```
/**
```

```
     * Unit tester used by the programmer during development.
     * @param args Not used.
     */
    public static void main(String[] args) {
      Numbers numbers = new Numbers();
      System.out.println(numbers);
    }
```

add

Most of us agreed that the `add` method was a good choice to implement next. One reason is that we cannot remove items until we can add a few. Also, we may not have decided how to handle the array during a remove operation.

Another reason for choosing the `add` method is because it is pretty easy. Basically, we just insert the element in the parameter into the array at position `count` and then increment `count`. **However**, we must think of everything, and that includes the decision we made to deal with the array if there is no more room. Given our attention to the invariant statement, the `count` is the index of the next empty cell. When the array if full, count will be the same as the array `length`, which would be the next cell if we were allowed to move passed the end of the array.

We decided to double the array capacity when it becomes full; we do this by creating a new array that is double in size, copying all the elements to this new array and then making that the new data field. We could accomplish this inside the `add` method, but since it is a bit of segue, we should consider creating a specific method instead. In our decision of whether to make this new method `public` or `private`, we consider:

(1) Will the method give a malicious user direct access to a private data field?
(2) Can we think of any possibility that a user may find this method useful?

If the answer to the first question is 'yes', then the method must be `private`. If the answer to the first question is 'no' and the answer to the second question is 'yes', then consider making the method `public`.

The method that enlarges the array could look like:

```
    /**
     * Doubles the size of the container array.
     */
    private void enlargeArray() {
      int[] tmp = new int[count+count];
      for (int i=0; i<count; i++) {
        tmp[i] = numbers[i];
      }
      numbers = tmp;
    }
```

Note that the commenting style is javadoc, but this method did not show up in the specifications, because it is `private`.

The `add` method can be written before or after the method to enlarge the array.

```
  /**
   * Adds an integer to the collection.
   * @param num The new number.
   */
  public void add(int num) {
    if (count == numbers.length) {
      enlargeArray();
    }
    numbers[count++] = num;
  }
```

Before proceeding further, we need to update `main` to include a call to the `add` method and then print out the list to make sure everything works.

remove

During the design and implementation of this method, you will have discussed how to *best* handle the removal of an integer from the array, in the simplest way. Options discussed may include:

replace the removed item with an indicator that the array cell is empty: This can work, but we need to determine what integer value would indicate that condition. Either one of the Java constants `Integer.MAX_VALUE` and `Integer.MIN_VALUE` could be a choice. Note that once we allow for spaces in the array, we need to re-think the invariant statement and re-do both the `toString` and `add` methods. Also, although `count` still is correct with respect to the number of elements, it will not necessarily point to the next available empty cell. All this considered, it is possible to handle the array in this manner, but we all agreed that if we could think of another way that required less maintenance, we would prefer that.

remove the item, then shift all the items after that item, one cell position to the left: This idea is more appealing than the previous one, because it does not require us to modify the `toString` and `add` methods.

remove the item, then stick the last item in the list into its spot: This idea is even more appealing, and we can use it because the specifications do not require any *ordering of the elements* of this list. This was generally the most favoured, since it does not break any requirement rules, and it is the easiest to implement. It also requires less work for the computer (which is something to think about as we become more skilled at programming).

Sometimes during implementation, an idea of *how* to implement something may not come to us initially, so we may write something like:

```java
/**
 * Removes a single number from the collection.
 * If there are duplicates, then there will be one fewer number of that
   value in the collection.
 * @param num The value of the number to be removed.
 */
public void remove(int num) {
  // find the index of the number we're looking for.
  // int index = ???
  if (index != -1) {
    // remove it by putting the last number into its index slot.
    numbers[index] = numbers[count-1];
    count--;
  }
}
```

The process of finding the index of the first occurrence of the number could be implemented as a separate method. We ask ourselves the same questions we asked when deciding about the `enlargeArray` method, and decide that it may be a good idea to have a method that finds the index location of a given number. However, because (depending on the lab), we decide that the user doesn't need to know about indices of items. The reason we choose the "first occurrence" is because the ordering doesn't matter, and we only have to remove one occurrence, as per the specifications.

Here is what our new method could look like:

```java
/*
 * Returns the first index location for the given integer in the
   collection.
```

```
     * If the number is not in the collection, -1 is returned.
     */
    private int findNum(int num) {
      for (int i=0; i<count; i++) {
        if (numbers[i] == num) {
          return i;
        }
      }
      return -1;
    }
```

Note that the commenting style is not javadoc in the above example; in fact, it is not as specific. This is because it is private and will only ever be used inside this class definition. The only person who will use it is the programmer who has access to the source code.

Note also that we can use -1 as an indicator that the number is not in the list, which is a reasonable possibility. Now, the `remove` method has what it needs:

```
    /**
     * Removes a single number from the collection.
     * If there are duplicates, then there will be one fewer number of that
        value in the collection.
     * @param num The value of the number to be removed.
     */
    public void remove(int num) {
      int index = findNum(num);
      int theNum;
      if (index != -1) {
        theNum = numbers[index];
        numbers[index] = numbers[--count];
      }
    }
```

Note the use of the prefix $--$ before count. In the statement, `count` is decremented first, then applied to the statement, which is exactly what we did in our incomplete version of `remove`

Enjoyed yourselves!

Document created by Bette Bultena