Chapter 2: Bag Implementations That Use Arrays

1. Why are the methods `getIndexOf` and `removeEntry` in the class `ArrayBag` private instead of public?

   The methods are implementation details that should be hidden from the client. They are not ADT bag operations and are not declared in `BagInterface`. Thus, they should not be public methods.

2. Implement a method replace for the ADT bag that replaces and returns any object currently in a bag with a given object.

   Add the following declaration to `BagInterface`:

```
/** Replaces an entry in this bag with a given object.
    @param replacement  the given object
    @return the original entry in the bag that was replaced */
public T replace(T replacement);
```

   Add the following method to `ArrayBag`:

```
public T replace(T replacement)
{
   T replacedEntry = bag[numberOfEntries - 1];
   bag[numberOfEntries - 1] = replacement;
   return replacedEntry;
} // end replace
```

3. Revise the definition of the method `remove`, as given in Segment 2.24, so that it removes a random entry from a bag. Would this change affect any other method within the class `ArrayBag`?

   Begin the file containing `ArrayBag` with the following statement:

```
import java.util.Random;
```

   Add the following data field to `ArrayBag`:

```
private Random generator;
```

   Add the following statement to the initializing constructor of `ArrayBag`:

```
generator = new Random();
```

   The definition of the method `remove` follows:

```
public T remove()
{
   T result = removeEntry(generator.nextInt(numberOfEntries));
   return result;
} // end remove
```

4. Define a method removeEvery for the class ArrayBag that removes all occurrences of a given entry from a bag.

   The following method is easy to write, but it is inefficient since it repeatedly begins the search from the beginning of the array `bag`:

```
/** Removes every occurrence of a given entry from this bag.
    @param anEntry  the entry to be removed */
public void removeEvery(T anEntry)
{
   int index = getIndexOf(anEntry);
   while (index > -1)
   {
      T result = removeEntry(index);
      index = getIndexOf(anEntry);
   } // end while
} // end removeEvery
```

The following method continues the search from the last found entry, so it is more efficient. But it is easy to make a mistake while coding:

```
public void removeEvery2(T anEntry)
{
   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anEntry.equals(bag[index]))
      {
         removeEntry(index);
         // since entries in array bag are shifted, want index to remain the same;
         // and since the for statement will increment index, need to decrement it here
         index--;
      } // end if
   } // end for
} // end removeEvery2
```

5. An instance of the class `ArrayBag` has a fixed size, whereas an instance of `ResizableArrayBag` does not. Give some examples of situations where a bag would be appropriate if its size is: a. Fixed; b. Resizable.

   a. Simulating any application involving an actual bag, such a grocery bag.
   b. Maintaining any collection that can grow in size or whose eventual size is unknown.

6. Suppose that you wanted to define a class `PileOfBooks` that implements the interface described in Project 2 of the previous chapter. Would a bag be a reasonable collection to represent the pile of books? Explain.

   No. The books in a pile have an order. A bag does not order its entries.

7. Consider an instance `myBag` of the class `ResizableArrayBag`, as discussed in Segments 2.36 to 2.40. Suppose that the initial capacity of `myBag` is 10. What is the length of the array `bag` after

   a. Adding 145 entries to `myBag`?

   b. Adding an additional 20 entries to `myBag`?

   a. 160. During the 11th addition, the bag doubles in size to 20. At the 21st addition, the bag's size increases to 40. At the 41st addition, it doubles in size again to 80. At the 81st addition, the size becomes 160 and stays that size during the addition of the 145th entry.
   b. 320. The array can accommodate 160 entries. Since it contains 145 entries, it can accommodate 15 more before having to double in size again.

8. Define a method at the client level that accepts as its argument an instance of the class `ArrayBag` and returns an instance of the class `ResizableArrayBag` that contains the same entries as the argument bag.

```
public static ResizableArrayBag<String> convertToResizable(BagInterface<String> aBag)
{
   ResizableArrayBag<String> newBag = new ResizableArrayBag<String>();

   Object[] bagArray = aBag.toArray();
   for (int index = 0; index < bagArray.length; index++)
      newBag.add((String)bagArray[index]);

   return newBag;
} // end convertToResizable
```

9. Suppose that a bag contains `Comparable` objects. Implement the following methods for the class `ArrayBag`:
   - The method `getMin` that returns the smallest object in a bag
   - The method `getMin` that returns the smallest object in a bag
   - The method `getMax` that returns the largest object in a bag
   - The method `removeMin` that removes and returns the smallest object in a bag
   - The method `removeMax` that removes and returns the largest object in a bag

   Students might have trouble with this exercise, depending on their knowledge of Java. The necessary details aren't covered until Chapters 8 and 9 when we discuss sorting. You might want to ask for a pseudocode solution instead of a Java method.

Change the header of `BagInterface` to

```
public interface BagInterface<T extends Comparable<? super T>>
```

Change the header of `ArrayBag` to

```
public class ArrayBag<T extends Comparable<? super T>> implements BagInterface<T>
```

Allocate the array `tempBag` in the constructor of `ArrayBag` as follows:

```
T[] tempBag = (T[])new Comparable<?>[capacity];
```

Allocate the array `result` in the method `toArray` as follows:

```
T[] result = (T[])new Comparable<?>[numberOfEntries];
```

The required methods follow:

```
/** Gets the smallest value in this bag.
    @returns a reference to the smallest object, or null if the bag is empty */
public T getMin()
{
   return bag[getIndexOfMin()];
} // end getMin

// Returns the index of the smallest value in this bag,
// or -1 if the bag is empty.
private int getIndexOfMin()
{
   int indexOfSmallest = -1;
   if (numberOfEntries > 0)
   {
      indexOfSmallest = 0;
      for (int index = 1; index < numberOfEntries; index++)
      {
         if (bag[index].compareTo(bag[indexOfSmallest]) < 0)
            indexOfSmallest = index;
      } // end for
   } // end if

   return indexOfSmallest;
} // end getIndexOfMin

/** Gets the largest value in this bag.
    @returns a reference to the largest object, or null if the bag is empty */
public T getMax()
{
   return bag[getIndexOfMax()];
} // end getMax

// Returns the index of the largest value in this bag,
// or -1 if the bag is empty.
private int getIndexOfMax()
{
   int indexOfLargest = -1;
   if (numberOfEntries > 0)
   {
      indexOfLargest = 0;
      for (int index = 1; index < numberOfEntries; index++)
      {
         if (bag[index].compareTo(bag[indexOfLargest]) > 0)
            indexOfLargest = index;
      } // end for
   } // end if

   return indexOfLargest;
} // end getIndexOfMax
```

```
/** Removes the smallest value in this bag.
    @returns a reference to the removed (smallest) object,
             or null if the bag is empty */
public T removeMin()
{
   T smallest = null;
   int indexOfMin = getIndexOfMin();
   if (indexOfMin > -1)
   {
      smallest = bag[indexOfMin];
      removeEntry(indexOfMin);
   } // end if

   return smallest;
} // end removeMin

/** Removes the largest value in this bag.
    @returns a reference to the removed (largest) object,
             or null if the bag is empty */
public T removeMax()
{
   T largest = null;
   int indexOfMax = getIndexOfMax();
   if (indexOfMax > -1)
   {
      largest = bag[indexOfMax];
      removeEntry(indexOfMax);
   } // end if

   return largest;
} // end removeMax
```

10. Suppose that a bag contains `Comparable` objects. Define a method for the class `ArrayBag` that returns a new bag of items that are less than some given item. The header of the method could be as follows:

    ```
    public BagInterface<T> getAllLessThan(Comparable<T> anObject)
    ```

    Make sure that your method does not affect the state of the original bag.

    See the note in the solution to Exercise 9 about student background.

```
/** Creates a new bag of objects that are in this bag and are less than a given object.
    @param  anObject  a given object
    @return a new bag of objects that are in this bag and are less than anObject */
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
{
   BagInterface<T> result = new ArrayBag<T>();

   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anObject.compareTo(bag[index]) > 0)
         result.add(bag[index]);
   } // end for

   return result;
} // end getAllLessThan
```

11. Define an `equals` method for the class `ArrayBag` that returns true when the contents of two bags are the same. Note that two equal bags contain the same number of entries, and each entry occurs in each bag the same number of times.

```
public boolean equals(Object other)
{
   boolean result = false;
   if (other instanceof ArrayBag)
   {
      // the cast is safe here
      @SuppressWarnings("unchecked")
      ArrayBag<T> otherBag = (ArrayBag<T>)other;
      int otherBagLength = otherBag.getCurrentSize();
      if (numberOfEntries == otherBagLength) // bags must contain the same number of objects
      {
         result = true;                      // assume equal
         for (int index = 0; (index < numberOfEntries) && result; index++)
         {
            T thisBagEntry = bag[index];
            T otherBagEntry = otherBag.bag[index];
            if (!thisBagEntry.equals(otherBagEntry))
               result = false;               // bags have unequal entries
         } // end for
      } // end if
   // else bags have unequal number of entries
   } // end if

   return result;
} // end equals
```

12. The class `ResizableArrayBag` has an array that can grow in size as objects are added to the bag. Revise the class so that its array also can shrink in size as objects are removed from the bag. Accomplishing this task will require two new private methods, as follows:

- The first new method checks whether we should reduce the size of the array:

  ```
  private boolean isTooBig()
  ```

  This method returns true if the number of entries in the bag is less than half the size of the array and the size of the array is greater than 20.
- The second new method creates a new array that is three quarters the size of the current array and then copies the objects in the bag to the new array:

  ```
  private void reduceArray()
  ```

Implement each of these two methods, and then use them in the definitions of the two `remove` methods.

```
private boolean isTooBig()
{
   return (numberOfEntries < bag.length / 2) && (bag.length > 20);
} // end isTooBig

private void reduceArray()
{
   T[] oldBag = bag;                                  // save reference to array
   int oldSize = oldBag.length;                       // save old max size of array

   @SuppressWarnings("unchecked")
   T[] tempBag = (T[])new Object[3 * oldSize / 4]; // reduce size of array; unchecked cast
   bag = tempBag;

   // copy entries from old array to new, smaller array
   for (int index = 0; index < numberOfEntries; index++)
      bag[index] = oldBag[index];
} // end reduceArray
```

```
public T remove()
{
   T result = removeEntry(numberOfEntries - 1);
   if (isTooBig())
      reduceArray();

   return result;
} // end remove

public boolean remove(T anEntry)
{
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);

   if (isTooBig())
      reduceArray();

   return anEntry.equals(result);
} // end remove
```

13. Consider the two private methods described in the previous exercise.
    a. The method `isTooBig` requires the size of the array to be greater than 20. What problem could occur if this requirement is dropped?
    b. The method `reduceArray` is not analogous to the method `ensureCapacity` in that it does not reduce the size of the array by one half. What problem could occur if the size of the array is reduced by one half instead of three quarters?

    a. If the size of the array is less than 20, it will need to be resized after very few additions or removals. Since 20 is not very large, the amount of wasted space will be negligible.
    b. If the size of the array is reduced by half, a sequence of alternating removes and adds can cause a resize with each operation.

14. Define the method `union`, as described in Exercise 5 of the previous chapter, for the class `ResizableArrayBag`.

```
public BagInterface<T> union(BagInterface<T> anotherBag)
{
   BagInterface<T> unionBag = new ResizableArrayBag<T>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;

   int index;

   // add entries from this bag to the new bag
   for (index = 0; index < numberOfEntries; index++)
      unionBag.add(bag[index]);

   // add entries from the second bag to the new bag
   for (index = 0; index < otherBag.getCurrentSize(); index++)
      unionBag.add(otherBag.bag[index]);

   return unionBag;
} // end union
```

15. Define the method `intersection`, as described in Exercise 6 of the previous chapter, for the class `ResizableArrayBag`.

```
public BagInterface<T> intersection(BagInterface<T> anotherBag)
{
   // The count of an item in the intersection is
   // the smaller of the count in each bag

   BagInterface<T> intersectionBag = new ResizableArrayBag<T>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;
   BagInterface<T> copyOfAnotherBag = new ResizableArrayBag<T>()

   int index;

   // copy the second bag
   for (index = 0; index < otherBag.numberOfEntries; index++)
   {
      copyOfAnotherBag.add(otherBag.bag[index]);
   } // end for

   // add to intersectionBag each item in this bag that matches an item in anotherBag;
   // once matched, remove it from the second bag

   for (index = 0; index < getCurrentSize(); index++)
   {
      if (copyOfAnotherBag.contains(bag[index]))
      {
         intersectionBag.add(bag[index]);
         copyOfAnotherBag.remove(bag[index]);
      } // end if
   } // end for

   return intersectionBag;
} // end intersection
```

16. Define the method `difference`, as described in Exercise 7 of the previous chapter, for the class `ResizableArrayBag`.

```
public BagInterface<T> difference(BagInterface<T> anotherBag)
{
   // The count of an item in the difference is the difference of the counts
   // in the two bags.

   BagInterface<T> differenceBag = new ResizableArrayBag<T>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;

   int index;

   // copy this bag
   for (index = 0; index < numberOfEntries; index++)
   {
      differenceBag.add(bag[index]);
   } // end for

   // remove the ones that are in anotherBag
   for (index = 0; index < otherBag.getCurrentSize(); index++)
   {
      if (differenceBag.contains(otherBag.bag[index]))
      {
         differenceBag.remove(otherBag.bag[index]);
      } // end if
   } // end for

   return differenceBag;
} // end difference
```