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.

```
/** Replaces an unspecified 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)
{
   T replacedEntry = bag[numberOfEntries - 1];
   bag[numberOfEntries - 1] = replacement;
   return replacedEntry;
} // end replace
```

3. Revise the definition of the method `clear`, as given in Segment 2.23, so that it is more efficient and calls only the method `checkInitialization`.

```
public void clear()
{
   checkInitialization();
   for (int index = 0; index < numberOfEntries; index++)
     bag[index] = null;
   numberOfEntries = 0;
} // end clear
```

4. 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
```

5. 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); // removeEntry is a private method in ArrayBag
      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, index can remain the same;
         // but the for statement will increment index, so need to decrement it here:
         index--;
      } // end if
   } // end for
} // end removeEvery2
```

6. 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.

7. 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.

8. 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.

9. 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.

For simplicity, we assume that the original bag contains strings. To drop this assumption, we would need to write the method as a generic method, which is described in Java Interlude 3.

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

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

   return newBag;
} // end convertToResizable
```

10. Suppose that a bag contains `Comparable` objects such as strings. A `Comparable` object belongs to a class that implements the standard interface `Comparable<T>`, and so has the method `compareTo`. Implement the following methods for the class `ArrayBag`:
    - 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 Java Interlude 3. 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<?>[desiredCapacity];
```

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()
{
   if (isEmpty())
      return null;
   else
      return bag[getIndexOfMin()];
} // end getMin

// Returns the index of the smallest in this bag.
// Precondition: The bag is not empty.
private int getIndexOfMin()
{
   int indexOfSmallest = 0;
   for (int index = 1; index < numberOfEntries; index++)
   {
      if (bag[index].compareTo(bag[indexOfSmallest]) < 0)
         indexOfSmallest = index;
   } // end for
```

```
   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()
{
   if (isEmpty())
      return null;
   else
      return bag[getIndexOfMax()];
} // end getMax

// Returns the index of the largest value in this bag.
// Precondition: The bag is not empty.
private int getIndexOfMax()
{
   int indexOfLargest = 0;
   for (int index = 1; index < numberOfEntries; index++)
   {
      if (bag[index].compareTo(bag[indexOfLargest]) > 0)
         indexOfLargest = index;
   } // end for
   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()
{
   if (isEmpty())
      return null;
   else
   {
      int indexOfMin = getIndexOfMin();
      T smallest = bag[indexOfMin];
      removeEntry(indexOfMin);
      return smallest;
   } // end if
} // 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()
{
   if (isEmpty())
      return null;
   else
   {
      int indexOfMax = getIndexOfMax();
      T largest = bag[indexOfMax];
      removeEntry(indexOfMax);
      return largest;
   } // end if
} // end removeMax
```

11. Suppose that a bag contains `Comparable` objects, as described in the previous exercise. 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 10 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<>();

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

   return result;
} // end getAllLessThan
```

12. 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. The order of the entries in each array is irrelevant.

```
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
```

13. 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
```

14. 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 `doubleCapacity` 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.

15. 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<>();
   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
```

16. 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<>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;
   BagInterface<T> copyOfAnotherBag = new ResizableArrayBag<>()

   int index;

   // Copy the second bag
   for (index = 0; index < otherBag.numberOfEntries; index++)
      copyOfAnotherBag.add(otherBag.bag[index]);
   // 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
```

17. 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<>();
   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
```

Lab 12 List Client

Goal

In this lab you will complete two applications that use the Abstract Data Type (ADT) list.

Resources

- Chapter 12: Lists
- *docs.oracle.com/javase/8/docs/api/* —API documentation for the Java `List` interface

In `javadoc` directory

- *ListInterface.html*—Interface documentation for the interface `ListInterface`

Java Files

- *AList.java*
- *CountingGame.java*
- *ListInterface.java*
- *Primes.java*

Introduction

The ADT list is one of the basic tools for use in developing software applications. It is an ordered collection of objects that can be accessed based on their position. Before continuing the lab you should review the material in Chapter 12. In particular, review the documentation of the interface *ListInterface.java*. While not all of the methods will be used in our applications, most of them will.

The first application you will complete implements a child's selection game. In the standard version of this game, a group of children gather in a circle and begin a rhyme. (One such rhyme goes "Engine, engine number nine, going down Chicago line. If the train should jump the track, will you get your money back? My mother told me to pick the very best one and you are not it.") Each word in the rhyme is chanted in turn by one person in the circle. The last person is out of the game and the rhyme is restarted from the next person. Eventually, one person is left and he or she is the lucky individual that has been selected. This application will read the number of players in the game and the rhyme from the keyboard. The final output will be the number of the selected person. You will use two lists in this application. The first will be a list of numbers representing the players in the game. The second will be a list of words in the rhyme.

The second application is one that computes prime numbers using the Sieve of Eratosthenes.

Pre-Lab Visualization

Counting Game

Suppose we have six players named 1, 2, 3, 4, 5, and 6. Further, suppose the rhyme has three words A, B, C. There will be five rounds played with one player eliminated in each round. In the following table, fill in the players next to the part of the rhyme they say in each round. Cross out the players as they are eliminated. The first round has been completed for you.

	Round 1	Round 2	Round 3	Round 4	Round 5
A	1				
B	2				
C	3				

 Eliminated Players: 1 2 ~~3~~ 4 5 6

Suppose we have five players named 1, 2, 3, 4, and 5. And the rhyme has six words A, B, C, D, E, F. There will be four rounds played. In the following table, fill in the players next to the part of the rhyme they say in each round. Cross out the players as they are eliminated.

	Round 1	Round 2	Round 3	Round 4
A				
B				
C				
D				
E				
F				

 Eliminated Players: 1 2 3 4 5

Primes

Suppose you are interested in finding the primes between 2 and 15 inclusive. The list of candidates is:

Candidates: 2 3 4 5 6 7 8 9 10 11 12 13 14 15

The algorithm proceeds in rounds. In each round a single prime is discovered and numbers that have that prime as a factor are eliminated.

Round 1:

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Primes*:

 *Composites*:

Round 2:

To make the operation of the algorithm clearer, copy the contents of the lists as they appear at the end of the previous round.

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Candidates*:

 *Primes*:

 *Composites*:

Round 3:

Again, copy the contents of the lists as they appear at the end of the previous round.

Cross the first value off of the Candidates list and add it to the Primes list. Cross out any candidate that is divisible by the prime you have just discovered and add it to the composites list.

 *Candidates*:

 *Primes*:

 *Composites*:

You can complete the other rounds if you wish, but most of the interesting work has been completed.

Finding Composites

The heart of this algorithm is removing the composite values from the candidates list. Let's examine this process more closely. In the first round, after removing the 2, the list of candidates is

Candidates: 3 4 5 6 7 8 9 10 11 12 13 14 15

How many values are in the list?

The first value to be examined is the 3. What is its index?

The second value to be examined is the 4. What is its index?

Since 4 is divisible by 2, we need to remove it from the list of candidates. What is the new list of candidates after the 4 is removed?

 *Candidates*:

The third value to be examined is the 5. What is its index?

The fourth value to be examined is the 6. What is its index?

Since 6 is divisible by 2, we need to remove it from the list of candidates. What is the new list of candidates after the 6 is removed?

 *Candidates*:

The fifth value to be examined is the 7. What is its index?

Can you simply loop over the indices from 1 to 13 to examine all the candidates?

Develop an algorithm to examine all the values in the candidates list and remove them if they are divisible by the given prime.

Directed Lab Work

Counting Game

Pieces of the `CountingGame` class already exist and are in *`CountingGame.java`*. Take a look at that code now if you have not done so already. Also before you start, make sure you are familiar with the methods available to you in the `AList` class (check *`ListInterface.html`*).

Step 1. Compile the classes `CountingGame` and `AList`. Run the `main` method in `CountingGame`.

Checkpoint: If all has gone well, the program will run and accept input. It will then generate a null pointer exception. The goal now is to create the list of players.

Step 2. Create a new `Alist<Integer>` and assign it to `players`.

Step 3. Using a loop, add new objects of type `Integer` to the players list.

Checkpoint: Compile and run the program. Enter 3 for the number of players. The program should print out { <1> <2> <3> } for the players list. The next goal is to do one round of the game. It will be encapsulated in the method doRhyme().

Step 4. Complete the `doRhyme()` method. Use the following algorithm.

For each word in the rhyme
    Print the word in the rhyme and the player that says it.
Print the name of the player to be removed.
Remove that player from the list.
Return the index of the player that will start the next round.

Step 5. `Call doRhyme(players, rhyme, position)` in `main` after the call to `getRhyme()`.

Step 6. Print out the new player list.

Checkpoint: Compile and run the program. Enter 6 for the number of players. Enter A B C for the rhyme. It should print out something similar to

Player 1: A
Player 2: B
Player 3: C
Removing player 3
The players list is { <1> <2> <4> <5> <6> }

Enter 5 for the number of players. Enter A B C D E F for the rhyme. Compare your result with your answers in the pre-lab. Reconcile any differences. The final goal is to do multiple rounds.

Step 7. Wrap the lines of code from the previous two steps in a `while` loop that continues as long as there is more than one player left.

Final checkpoint: Compile and run the program. Enter 6 for the number of players. Enter A B C for the rhyme. The players should be removed in the order 3, 6, 4, 2, 5. The winner should be player 1.

Enter 5 for the number of players. Enter A B C D E F for the rhyme. Compare your result with your answers in the pre-lab exercises. Reconcile any differences.

Primes

The skeleton of the `Primes` class already exists and is in ***`Primes.java`***.

Step 1. Look at the skeleton in ***`Primes.java`***. Compile `Primes`. Run the `main` method in `Primes`.

Checkpoint: If all has gone well, the program will run and accept input. It will then end. The goal now is to create the list of candidates.

Step 2. In `main` declare and create the candidates list. Add in the values.

Step 3. Print out the candidates list.

Checkpoint: Compile and run the program. Enter 7 for the maximum value. You should see the list { <2> <3> <4> <5> <6> <7> }. The next goal is to do a single round finding a prime in the candidates list.

Step 4. In `main` declare and create the primes and composites lists.

Step 5. Remove the first value from the primes list and remember it in a variable.

Step 6. Print out the prime that was discovered.

Step 7. Add it to the primes list.

Step 8. Print out all three lists.

Checkpoint: Compile and run the program. Enter 7 for the maximum value. The value 2 should be removed from the candidates list and added to the primes. Now all values that are divisible by the prime should be removed from the candidates list and added to the composites list. Next, this procedure will be encapsulated in the method getComposites().

Step 9. Refer to the pre-lab exercises and complete the `getComposites()` method. To determine if one integer value is divisible by another, you can use the modulus operator (`%` in Java).

Step 10. Between the code from steps 7 and 8, call `getComposites()`.

Checkpoint: Compile and run the program. Enter 15 for the maximum value. Compare the results with the pre-lab exercises. Reconcile any differences.

Just as in the counting game, a loop will be used to do the rounds.

Step 11. Wrap the code from steps 5 through 8 in a `while` loop that continues as long as the candidates list is not empty.

Final checkpoint: Compile and run the program. Enter 15 for the maximum value. Compare the results with the pre-lab exercises. Reconcile any differences.

Run the program with 100 as the maximum value. Carefully verify your results.

Post-Lab Follow-Ups

1. Modify the counting game program to compute the winning player for different numbers of players from 2 up to an input value. Figure out which player wins for a rhyme of length three for games with 2 to 20 players and record the results in a table. Can you discover a relation between the size of the player list, the length of the rhyme, and the winning player?

2. After a certain point in our iteration, all the remaining values in the candidates list are prime. Modify the program to just copy values directly when that point has been reached.

3. Write a program that will get a list of words and then remove any duplicates.

4. Write a program that will get two lists of words and will create a third list that consists of any words in common between the two input lists.

5. Write a program that will read in a list of words and will create and display two lists. The first list will be the words in odd positions of the original list. The second list will be all the remaining words.

6. Write a program that will get a list of integer values each of which is greater than one and determines if all the posible pairs of values from the list are relatively prime. Two values are relatively prime if they share no prime factors. For example, the values 10 and 77 are relatively prime, but 10 and 55 are not since the share 5 as a factor. If we look at the list {6, 25, 77} each of the pairs (6, 25), (6, 77), and (25,77) are relatively prime and it passes our test. On the other hand the list {6, 25, 14} will fail the test. While the pairs (6, 25) and (25, 14) are relatively prime, the pair (6,14) is not.