Chapter 10

An Array Instance Variable

Goal

- Implement a type that uses an array instance variable.

10.1 `StringBag` — A Simple Collection Class

As you continue your study of computing fundamentals, you will spend a fair amount of time using arrays and managing collections of data. The Java array is one of several data storage structures used inside classes with the main task of storing a collection. These are known as collection classes with some of the following characteristics:

- The main responsibility of a collection class is to store a collection of objects
- Objects are added and removed from a collection
- A collection class allows clients to access the individual elements
- A collection class may have search-and-sort operations for locating a particular item.
- Some collections allow duplicate elements; other collections do not

The Java array uses subscript notation to access individual elements. The collection class shown next exemplifies a higher-level approach to storing a collection of objects. It presents users with messages and hides

the array processing details inside the methods. The relatively simple collection class also provides a review of Java classes and methods. This time, however, the class will have an array instance variable. The methods will employ array-processing algorithms. More specifically, this collection will represent a bag. Bag is a mathematical term for an unordered collection of values that may have duplicates. It is also know as a multi-set. This bag will be store a collection of strings and will be named `StringBag`. A `StringBag` object will have the following characteristics:

- A `StringBag` object can store a collection of `String` objects
- `StringBag` elements need not be unique, duplicates are allowed
- The order of elements is not important
- Programmers can ask how many occurrences of a String are in the bag (may be 0)
- Elements can be removed from a `StringBag` object
- This `StringBag` class is useful for learning about collections, array processing, Java classes and Test-Driven Development.

A `StringBag` object can store any number of `String` objects. A `StringBag` object will understand the messages such as `add`, `remove` and `occurencesOf`. The design of `StringBag` is provided here as three commented method headings.

```
// Put stringToAdd into this StringBag (order not important)
public void add(String stringToAdd);

// Return how often element equals an element in this StringBag
public int occurencesOf(String element);

// Remove one  occurrence of stringToRemove if found and return true.
// Return false if stringToRemove is not found in this StringBag.
public boolean remove(String stringToRemove);
```

Using Test Driven Development, the tests come first. Which method should be tested first? It's difficult to implement only one and know it works. If we work on `add` alone, how do we know an element has actually been added. One solution is to develop `occurencesOf` at the same time and verify both are working together. A test method could add several elements and verify they are there with `occurencesOf`. We should also verify contains returns false for elements in the bag. So `add(String)` and `occurencesOf(String)` will be developed first. We'll begin with a unit test with one test method that adds one element. `occurencesOf` should return 0 before add and 1 after.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringBagTest {

  @Test
  public void testAddAndOccurencesOfForOnlyOneElement () {
    StringBag friends = new StringBag();
    friends.add("Sage");
    assertEquals(1, friends.occurencesOf("Sage"));
  }
}
```

Of course, this unit test will not compile. The class doesn't even exist; nor do the `add` and `occurencesOf` methods; nor does the constructor. The following start at a `StringBag` type at least allows the unit test to compile. The assertions will not pass, at least not yet. All methods are written as stubs—a temporary substitute for yet-to-be-developed code.

```
// A class for storing a multi-set (bag) of String elements.
public class StringBag {
```

```
  // Construct an empty StringBag object (no elements stored yet)
  public StringBag() {
    // TODO Complete this method
  }

  // Add an element to this StringBag
  public void add(String stringToAdd) {
     // TODO Complete this method
  }

  // Return how often element equals an element in this StringBag
  public int occurencesOf(String element) {
     // TODO Complete this method
     return 0;
  }
}
```

The **`StringBag`** Constructor

The private instance variables of the `StringBag` class include an array named `data` for storing a collection of `String` objects. Each `StringBag` object also has an integer named `n` to maintain the number of meaningful elements that are in the `StringBag`. The `add` and `occurencesOf` methods will need both instance variables to accomplish their responsibilities. The constructor establishes an empty `StringBag` object by setting `n` to zero. The array capacity is set to the arbitrary initial capacity of 10. We don't know how big the collection will grow to when used later (and we will have to deal with that later).

```
public class StringBag {
```

```
   private String[] data; // Stores the collection
   private int n;         // Current number of elements

   // Construct an empty StringBag object
   public StringBag() {
      n = 0;
      data = new String[10]; // Initial capacity is 10
   }
```

`public void add(String stringToAdd)`

Both `n` and `data` must be available to the `add` method. This is not a problem, since any `StringBag` method has access to the private instance variables of `StringBag`. To add an element to the `StringBag`, the argument reference passed to the `stringToAdd` parameter can be placed at the "end" of the array, or more specifically, at the first available array location. This two-step algorithm summarizes how a new `String` is added to the first available array position:

Algorithm: **Adding an element**

`data[n]` = *the-argument-passed-to-`StringBag.add`*
increment `n` by `+1`

The argument passed to `StringBag`'s `add` method is stored into the proper array location using `n` as the index. Then `n` gets incremented by 1 to reflect the new addition. Incrementing `n` by 1 maintains the number of elements in the `StringBag`.

Incrementing `n` also conveniently sets up a situation where the next added element is inserted into the proper array location. The array location at `data[n]` is the next place to store the next element can be placed. This is demonstrated in the following view of the state of the `StringBag` before and after the string `"and a fourth"` after this code executes

```
StringBag bag = new StringBag();
bag.add("A string");
bag.add("Another string");
bag.add("and still another");
```

Before

Instance Variables	State of `bagOfStrings`
data[0]	"A string"
data[1]	"Another string"
data[2]	"and still another"
data[3]	null // next available
data[4]	null
...	...
data[9]	null
n	3

After

Instance Variable	State of `bagOfStrings`
data[0]	"A string"
data[1]	"Another string"
data[2]	"and still another"
data[3]	**"and a fourth"**
data[4]	null // next available
...	...
data[9]	null
n	4

Here is the `add` method that places new elements at the first available location. It is important to keep the elements together. Don't allow null between elements. This method ensures nulls are not in the mix.

```
// Add an element to this StringBag
public void add(String stringToAdd) {
  // Store the reference into the array
  data[n] = stringToAdd;
  // Make sure n is always increased by one
  n++;
}
```

The unit test is run, but the single test method does not pass; `occurencesOf` still does nothing.

`public int occurencesOf(String element)`

Since there is no specified ordering for Bags in general or `StringBag` in particular, the element passed as an argument may be located at any index. Also, a value that equals the argument may occur more than once. Thus each element in indexes 0..n-1 must be compared. It makes the most sense to use the `equals` method, assuming `equals` has been overridden to compare the state of two objects rather than the reference values. And with `String`, equals does compare state.

By setting `result` to 0 below, the `occurencesOf` method first states there are no elements equal to `element`.

```
// Return how often element equals an element in this StringBag
public int occurencesOf(String element) {
  int result = 0;
  for (int subscript = 0; subscript < n; subscript++) {
    if (element.equals(data[subscript]))
      result++;
  }
  return result;
}
```

The for loop then iterates over every meaningful element in the array. Each time `element` equals any array element, `result` increments by 1. Our first assertion passes.

```
@Test
public void testAddAndOccurencesOfForOnlyOneElement() {
  StringBag friends = new StringBag();
  friends.add("Sage");
  assertEquals(1, friends.occurencesOf("Sage"));
}
```

Other Test Methods

Another test method verifies that duplicate elements are can exist and are found.

```
@Test
public void testOccurencesOf() {
  StringBag names = new StringBag();
  names.add("Tyler");
  names.add("Devon");
  names.add("Tyler");
  names.add("Tyler");
  assertEquals(1, names.occurencesOf("Devon"));
  assertEquals(3, names.occurencesOf("Tyler"));
}
```

Another test method verifies 0 is returned when the String argument is not in the bag.

```
@Test
public void testOccurencesOfWhenItShyouldReturnZeros() {
  StringBag names = new StringBag();
  assertEquals(0, names.occurencesOf("Devon"));
  assertEquals(0, names.occurencesOf("Tyler"));
  names.add("Sage");
  names.add("Hayden");
  assertEquals(0, names.occurencesOf("Devon"));
  assertEquals(0, names.occurencesOf("Tyler"));
}
```

Another test method documents that this collection is case sensitive.

```
@Test
public void testOccurencesOfForCaseSensitivity() {
  StringBag names = new StringBag();
  names.add("UPPER");
  names.add("Lower");

  // Not in the bag (case sensitive)
  assertEquals(0, names.occurencesOf("upper"));
  assertEquals(0, names.occurencesOf("lower"));
```

```
    // In the bag
    assertEquals(1, names.occurencesOf("UPPER"));
    assertEquals(1, names.occurencesOf("Lower"));
  }
```

Yet another test method tries to add 500 strings only to find something goes wrong.

```
  @Test
  public void testAdding500Elements() {
    StringBag bag = new StringBag();
    for (int count = 1; count <= 500; count++) {
      bag.add("Str#" + count);
    }
    assertEquals(1, bag.occurencesOf("Str#1"));
    assertEquals(1, bag.occurencesOf("Str#2"));
    assertEquals(1, bag.occurencesOf("Str#499"));
    assertEquals(1, bag.occurencesOf("Str#500"));
  }
```

```
java.lang.ArrayIndexOutOfBoundsException: 10
at StringBag.add(StringBag.java:34)
at StringBagTest.testAdding500Elements(StringBagTest.java:39)
```

After 10 adds, n == 10. The attempt to store the 11th element in the StringbBag results in an `ArrayIndexOutOfBounds` exception with the attempt to assign an element to `data[10]`.

Before any new `String` is added, a check should be made to ensure that there is the capacity to add another element. If the array is filled to capacity (`n == data.length`) there is not enough room to add the new element. In this case, we need to increase the array capacity.

The code to increase the capacity of the array could be included in the `add` method. However this task is complex enough that it will be placed into a "helper" method named `growArray`. The `add` method changes with a guarded action: grow the array only when necessary.

```
  public void add(String stringToAdd) {
    // Make sure the array can store a new element
    if (n == data.length) {
      growArray();
    }

    // Store the reference into the array
    data[n] = stringToAdd;
    // Make sure my_size is always increased by one
    n++;
  }
```

The `growArray` method will help this `add` method perform its task with less code. The `add` method delegates a well-defined responsibility of growing the array to another method. This makes for more readable and maintainable code.

`private void growArray()`

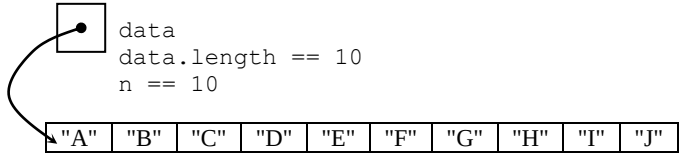
Because `growArray` is inside class `StringBag`, any `StringBag` object can send a `growArray` message to itself. The message was sent from this object in `add`. And because `data` is an instance variable, any `StringBag` object can change `data` to reference a new array with more capacity. This is done with the following algorithm:

- Make a temporary array that is bigger (by 10) than the instance variable.
- Copy the original contents (`data[0]` through `data[n - 1]`) into this temporary array.
- Assign the reference to the temporary array to the array instance variable

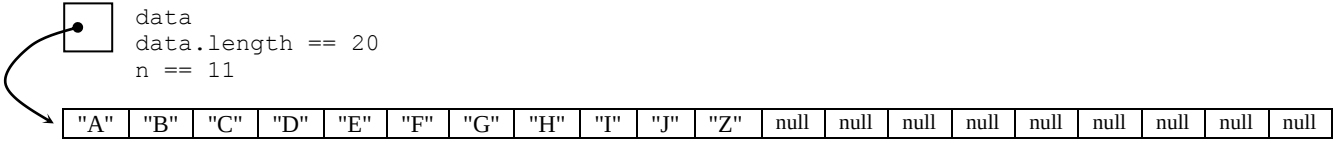
```
// Change data to have the same elements in indexes 0..n - 1
// and have the same number of new array locations to store new elements.
private void growArray() {
  String[] temp = new String[n + 10];
```

```
    // Copy all existing elements into the new and larger array
    for (int index = 0; index < n; index++) {
      temp[index] = data[index];
    }
    // Store a reference to the new bigger array
    // as part of this object's state
    data = temp;
  }
```

When the array is filled to capacity (with the Strings "A" .. "J" added in this example), the instance variables `data` and `n` look like this:



During the message `add("Z");`, the `add` method would send the `growArray message` in order to increase the capacity by 10. The instance variables would change to this picture of memory:



Note: The `growArray` method is declare private because it is better design to *not* clutter the public part of a class with things that users of the class are not able to use or are not interested in using. It is good practice to hide details from users of your software.

`public boolean remove(String stringToRemove)`

If `stringToRemove` is found to equal one of the strings referenced by the array, `remove` effectively takes one of the occurrences of the String element. Consider the following test method that attempts to remove `"Not in the bag"`.

```
@Test
public void testRemoveOneThatIsThereAnotherThatIsNot() {
  StringBag bag = new StringBag();
  bag.add("A string");
  bag.add("Another string");
  bag.add("and still another");
  bag.add("and a fourth");
  assertFalse(bag.remove("Not in the bag"));
  assertTrue(bag.remove("Another string"));
}
```

Here are the values of the instance variables `data` and `n` and of the local objects `index` and `stringToRemove` while trying to remove `"Another string"`:

Instance Variable	**State of `bag`**
`data[0]`	`"A string"`
`data[1]`	`"Another string"`
`data[2]`	`"and still another"`
`data[3]`	`"and a fourth"`
`data[4]`	`null`
...	...
`data[9]`	`null`
`n`	4

The algorithm used to remove an element is in these steps (other algorithms also work).

- Find the index of an element to remove, or set to -1 if `stringToRemove` does not exist

- If the index != -1, move the element at the end of the array to this index
- Decrement n (n--)

The remove algorithm calls the private helper method `indexOf` that has the purpose of returning an index of the string to be removed. If the string does not equal an array element, the `indexOf` method (discussed later) returns -1. In this case of trying to remove the string `"Not in the bag"` the method simply returns false. The method terminated and the first assertion (above) passes.

```
// Remove an element that equals stringToRemove if found and return true.
// Return false if stringToRemove was not found in this StringBag.
public boolean remove(String stringToRemove) {

  // indexOf returns the index of an element that equals stringToRemove
  // or -1 if stringToRemove is not in this bag.
  int subscript = indexOf(stringToRemove);
  if (subscript == -1)
    return false;
  else {   // . . .
```

In the 2nd assertion `assertTrue(bag.remove("Another string"));` that attempts to remove an element that does exist, the array will be changed, `n` will be changed, and `indexOf` will return true. These variables that are local to `remove` indicate the string was found at index 1.

Local Variable	**State of `remove`'s Local Variable after a Sequential Search**
`stringToRemove`	`"Another string"`
`index`	`1`

Once found, the reference stored in `data[index]` must somehow be removed from the array, which is currently

`data[1]` or `"Another string"`. The simple way to do this is to move the last element into the spot where `stringToRemove` was found. It is okay to destroy the reference in `data[1]`. This is the object to be removed from the `StringBag`. Also, since there is no ordering requirement, it is also okay to move `data[n - 1]`, which is the last meaningful element in the array. When `n--` occurs, the 2nd reference to the string at data[n-1] is no longer considered to be in the collection. Although not necessary, this code assigns `null` to that 2nd unneeded reference.

```
        // Move the last string in the array to where stringToRemove was found.
        data[subscript] = data[n - 1];
        // Mark old array element as no longer holding a reference (not required)
        data[n - 1] = null;
        // Decrease this StringBag's number of elements
        n--;
        // Let this method return true to where the message was sent
        return true;
      }
    } // End method remove
```

The state of `StringBag` now looks like this (three changes are highlighted):

Instance Variable	**State of `bagOfStrings`**	
`data[0]`	`"A string"`	
`data[1]`	`"And a fourth"`	Overwrite `"another string"`
`data[2]`	`"and still another"`	
`data[3]`	**`null`**	`data[3]` is no longer meaningful
`data[4]`	**`null`**	
...		
`data[9]`	**`null`**	
`n`	`3`	`n` is 3 now

Although the elements are not in the same order (this was not a requirement), the same elements exist after the requested removal. Because the last element has been relocated, `n` must decrement by 1. There are now only

three, not four, elements in this `StringBag` object.

The same code works even when removing the last element. The assignment is done. Decreasing `n` by one effectively eliminates the last element.

`private int indexOf(String element)`

The remove method used another method to find the index of an element to remove (or -1 if no element found). Although this code could have gone in remove, the well-defined responsibility of finding the index of an element in an array was placed in this private helper method to keep the remove algorithm a bit simpler. The `indexOf` method will sequentially search each array element beginning at index 0 until one of two things happen.

1. element equals an array element and that index of that element is returned to method `remove(String element)`
2. the loop terminates because there are no more element to examine. In this case, `indexOf` returns -1 to method `remove(String element)`

```
// Return the index of the first occurrence of stringToRemove.
private int indexOf(String element) {
  // Look at all elements until the string
  for (int index = 0; index < n; index++) {
    if (element.equals(data[index]))
      return index;
  }
  // Otherwise result is not changed from -1.
  return -1;
}
```

Again we see a helper method declared `private` because `indexOf` is currently considered a method that programmers are *not* meant to use. It was not in the specification. Here is the complete `StringBag` class.

```java
// A class for storing an unordered collection of Strings.
// This class was designed to provide practice and review in
// implementing methods and classes along with using arrays.
public class StringBag {

  private String[] data; // Stores the collection
  private int n; // Current number of elements

  // Construct an empty StringBag object
  public StringBag() {
    n = 0;
    data = new String[10]; // Initial capacity is 10
  }

  // Return the element at the specified index.
  // Precondition: index >= 0 && index < size()
  public String get(int index) {
    return data[index];
  }

  // Add a string to the StringBag in no particular place.
  // Always add StringToAdd (unless the computer runs out of memory)
  public void add(String stringToAdd) {
    // Make sure the array can store a new element
    if (n == data.length) {
      growArray();
    }

    // Store the reference into the array
    data[n] = stringToAdd;
    // Make sure my_size is always increased by one
    n++;
  }

  // Change data to have the same elements in indexes 0..n - 1 and have
  // the same number of new array locations to store new elements.
```

```
private void growArray() {
  String[] temp = new String[n + 10];

  // Copy all existing elements into the new and larger array
  for (int index = 0; index < n; index++) {
    temp[index] = data[index];
  }

  // Store a reference to the new bigger array as part of this
  // object's state
  data = temp;
}

// Return how often element equals an element in this StringBag
public int occurencesOf(String element) {
  int result = 0;
  for (int subscript = 0; subscript < n; subscript++) {
    if (element.equals(data[subscript]))
      result++;
  }
  return result;
}


// Remove an element that equals stringToRemove if found and return true.
// Return false if stringToRemove was not found in this StringBag.
public boolean remove(String stringToRemove) {
  int subscript = indexOf(stringToRemove);
  if (subscript == -1)
    return false;
  else {
    // Move the last string in the array to where stringToRemove was found.
    data[subscript] = data[n - 1];
    // Mark old array element as no longer holding a reference (not required)
    data[n - 1] = null;
    // Decrease this StringBag's number of elements
```

```
      n--;
      // Let this method return true to where the message was sent
      return true;
    }
  }

  // Return the index of the first occurrence of stringToRemove.
  // Otherwise return -1 if stringToRemove is not found.
  private int indexOf(String element) {
    // Look at all elements until the string
    for (int index = 0; index < n; index++) {
      if (element.equals(data[index]))
        return index;
    }

    // Otherwise result is not changed from -1.
    return -1;
  }

}  // End class StringBag
```

Other Test Methods

The `remove` method and its `indexOf` method are complex. Further testing is appropriate. This test verifies that all duplicates can be removed.

```
  @Test
  public void testRemoveWhenDuplicatedO() {
    StringBag bag = new StringBag();
    bag.add("A");
    bag.add("B");
    bag.add("B");
    bag.add("B");
    bag.add("A");
```

```
    assertEquals(3, bag.occurencesOf("B"));
    assertTrue(bag.remove("B"));
    assertEquals(2, bag.occurencesOf("B"));

    assertTrue(bag.remove("B"));
    assertEquals(1, bag.occurencesOf("B"));

    assertTrue(bag.remove("B"));
    assertEquals(0, bag.occurencesOf("B"));

    // There should be no more Bs
    assertFalse(bag.remove("B"));
    assertEquals(0, bag.occurencesOf("lower"));
  }
```

Other tests should be made for these situations:

- when the bag is empty
- when there is one element, try removing an element that is not there
- when there is one element, try removing an element that *is* there
- remove all elements when size > 2

```
  @Test
  public void testRemoveWhenEmpty() {
    StringBag bag = new StringBag();
    assertEquals(0, bag.occurencesOf("B"));
    assertFalse(bag.remove("Not here"));
    assertEquals(0, bag.occurencesOf("B"));
  }


  @Test
  public void testRemoveNonExistentElementWhenSizeIsOne() {
```

```
    StringBag bag = new StringBag();
    bag.add("Only one element");
    assertEquals(1, bag.occurencesOf("Only one element"));
    assertFalse(bag.remove("Not here"));
    assertEquals(1, bag.occurencesOf("Only one element"));
  }

  @Test
  public void testRemoveElementWhenSizeIsOne() {
    StringBag bag = new StringBag();
    bag.add("Only one element");
    assertEquals(1, bag.occurencesOf("Only one element"));
    assertTrue(bag.remove("Only one element"));
    assertEquals(0, bag.occurencesOf("Only one element"));
  }
  @Test
  public void testRemoveAllElementsWhenSizeGreaterThanTwo() {
    StringBag bag = new StringBag();
    bag.add("A");
    bag.add("B");
    bag.add("C");
    assertTrue(bag.remove("A"));
    assertTrue(bag.remove("B"));
    assertTrue(bag.remove("C"));
    assertEquals(0, bag.occurencesOf("A"));
    assertEquals(0, bag.occurencesOf("B"));
    assertEquals(0, bag.occurencesOf("C"));
  }
```

Self-Check

10-1 What happens when an attempt is made to remove an element that is not in the bag.

10-2 Using the implementation of `remove` just given, what happens when an attempt is made to remove an element from an empty `StringBag` (`n == 0`)?

10-3 Must `remove` always maintain the `StringBag` elements in the same order as that in which they were originally added?

10-4 What happens when an attempt is made to remove an element that has two of the same values in the `StringBag`?

10-5 Write the output of the following code:

```
StringBag aBag = new StringBag();
aBag.add("First");
aBag.add("Second");
aBag.add("Third");
System.out.println(aBag.occurencesOf("first"));
System.out.println(aBag.occurencesOf("Second"));
System.out.println(aBag.remove("First"));
System.out.println(aBag.remove("Third"));
System.out.println(aBag.remove("Third"));
System.out.println(aBag.occurencesOf("first"));
System.out.println(aBag.occurencesOf("Second"));
```

Answers to Self-Checks

10-1 `remove` returns `false`, the `StringBag` object does not change.

10-2 Nothing noticeable to the user happens. The loop test (`index < my_size`) is false immediately, so index remains 0. Then the expression `if ( index == my_size )` is true and

false is returned.

10-3 No. The last element may be moved to the first vector position, or the second, or anywhere else. There are other collections used to store elements in order.

10-4 `StringBag remove` removes the first occurrence. All other occurrences of the same value remain in the bag.

10-5

```
0
1
true
true
false
0
1
```