As with the bag, the purpose of these methods is to allow a programmer to explicitly set the capacity of the collection. If a programmer does not explicitly set the capacity, then the class will still work correctly, but some operations will be less efficient because the capacity might be repeatedly increased.

The Sequence ADT—Documention

The complete specification for this first version of our sequence class is shown in Figure 3.10. This specification is also available from the DoubleArraySeq link at the following web address:

http://www.cs.colorado.edu/~main/docs/

When you read the specification, you'll see that the package name is edu.colorado.collections. So you should create a subdirectory called edu/colorado/collections for your implementation.

The specification also indicates some limitations—the same limitations that we saw for the bag class. For example, an OutOfMemoryError can occur in any method that increases the capacity. Several of the methods throw an IllegalStateException to indicate that they have been illegally activated (with no current element). Also, an attempt to move the capacity beyond the maximum integer causes the class to fail by an arithmetic overflow.

After you've looked through the specifications, we'll suggest a design that uses three private instance variables.

The Sequence ADT—Design

Our suggested design for the sequence ADT has three private instance variables. The first variable, data, is an array that stores the elements of the sequence. Just like the bag, data is a partially filled array, and a second instance variable, called manyItems, keeps track of how much of the data array is currently being used. Therefore, the used part of the array extends from data[0] to data[manyItems-1]. The third instance variable, currentlndex, gives the index of the current element in the array (if there is one). Sometimes a sequence has no current element, in which case currcntIndex will be set to the same number as manyItems (since this is larger than any valid index). The complete invariant of our ADT is stated as three rules:

1. The number of elements in the sequence is stored in the instance variable manyItems.

2. For an empty sequence (with no elements), we do not care what is stored in any of data; for a nonempty sequence, the elements of the sequence are stored from the front to the end in data[0] to data[manyItems-1], and we don't care what is stored in the rest of data.

3. If there is a current element, then it lies in data[currentIndex]; if there is no current element, then currentlndex equals manyItems.

Specification

Constructor for the DoubleArraySeq

public DoubleArraySeq()
Initialize an empty sequence with an initial capacity of 10. Note that the addAfter and addBefore methods work efficiently (without needing more memory) until this capacity is reached.

Postcondition:
This sequence is empty and has an initial capacity of 10.

Throws: OutOfMemoryError
Indicates insufficient memory for new double[10].

Second Constructor for the DoubleArraySeq

public DoubleArraySeq(int initialCapacity)
Initialize an empty sequence with a specified initial capacity. Note that the addAfter and addBefore methods work efficiently (without needing more memory) until this capacity is reached.

Parameter:
initialCapacity—the initial capacity of this sequence

Precondition:
initialCapacity is non-negative.

Postcondition:
This sequence is empty and has the given initial capacity.

Throws: IllegalArgumentException
Indicates that initialCapacity is negative.

Throws: OutOfMemoryError
Indicates insufficient memory for new double[initialCapacity].

(continued)
(FIGURE 3.10 continued)

♦ addAfter and addBefore

public void addAfter(double element)
public void addBefore(double element)

Adds a new element to this sequence, either before or after the current element. If this new element would take this sequence beyond its current capacity, then the capacity is increased before adding the new element.

Parameter:
element – the new element that is being added

Postcondition:
A new copy of the element has been added to this sequence. If there was a current element, then addAfter places the new element after the current element, and addBefore places the new element before the current element. If there was no current element, then addAfter places the new element at the end of this sequence, and addBefore places the new element at the front of this sequence. In all cases, the new element becomes the new current element of this sequence.

Throws: OutOfMemoryError
Indicates insufficient memory to increase the size of this sequence.

Note:
An attempt to increase the capacity beyond Integer.MAX_VALUE will cause this sequence to fail with an arithmetic overflow.

♦ addAll

public void addAll(DoubleArraySeq addend)
Place the contents of another sequence at the end of this sequence.

Parameter:
addend – a sequence whose contents will be placed at the end of this sequence

Precondition:
The parameter, addend, is not null.

Postcondition:
The elements from addend have been placed at the end of this sequence. The current element of this sequence remains where it was, and the addend is also unchanged.

Throws: NullPointerException
Indicates that addend is null.

Throws: OutOfMemoryError
Indicates insufficient memory to increase the capacity of this sequence.

Note:
An attempt to increase the capacity beyond Integer.MAX_VALUE will cause this sequence to fail with an arithmetic overflow.

(continued)
(FIGURE 3.10 continued)

♦ advance
  public void advance( )
  Move forward so that the current element is now the next element in this sequence.
  
  **Precondition:**
  isCurrent( ) returns true.
  
  **Postcondition:**
  If the current element was already the end element of this sequence (with nothing after it),
  then there is no longer any current element. Otherwise, the new element is the element
  immediately after the original current element.
  
  **Throws:** IllegalStateException
  Indicates that there is no current element, so advance may not be called.

♦ clone
  public DoubleArraySeq clone( )
  Generate a copy of this sequence.
  
  **Returns:**
  The return value is a copy of this sequence. Subsequent changes to the copy will not affect
  the original, nor vice versa. The return value must be typecast to a DoubleArraySeq before
  it is used.
  
  **Throws:** OutOfMemoryError
  Indicates insufficient memory for creating the clone.

♦ concatenation
  public static DoubleArraySeq concatenation
  (DoubleArraySeq s1, DoubleArraySeq s2)
  Create a new sequence that contains all the elements from one sequence followed by another.
  
  **Parameters:**
  s1 – the first of two sequences
  s2 – the second of two sequences
  
  **Precondition:**
  Neither s1 nor s2 is null.
  
  **Returns:**
  a new sequence that has the elements of s1 followed by the elements of s2 (with no current element)
  
  **Throws:** NullPointerException
  Indicates that one of the arguments is null.
  
  **Throws:** OutOfMemoryError
  Indicates insufficient memory for the new sequence.
  
  **Note:**
  An attempt to increase the capacity beyond Integer.MAX_VALUE will cause this sequence to
  fail with an arithmetic overflow.
(FIGURE 3.10 continued)

♦ `ensureCapacity`
  public void ensureCapacity(int minimumCapacity)
  Change the current capacity of this sequence.
  **Parameter:**
  `minimumCapacity` – the new capacity for this sequence
  **Postcondition:**
  This sequence’s capacity has been changed to at least `minimumCapacity`.
  **Throws:** `OutOfMemoryError`
  Indicates insufficient memory for new `double[minimumCapacity]`.

♦ `getCapacity`
  public int getCapacity()
  Accessor method to determine the current capacity of this sequence. The `addBefore` and `addAfter` methods work efficiently (without needing more memory) until this capacity is reached.
  **Returns:**
  the current capacity of this sequence

♦ `getCurrent`
  public double getCurrent()
  Accessor method to determine the current element of this sequence.
  **Precondition:**
  `isCurrent()` returns true.
  **Returns:**
  the current element of this sequence
  **Throws:** `IllegalStateException`
  Indicates that there is no current element.

♦ `isCurrent`
  public boolean isCurrent()
  Accessor method to determine whether this sequence has a specified current element that can be retrieved with the `getCurrent` method.
  **Returns:**
  `true` (there is a current element) or `false` (there is no current element at the moment)

♦ `removeCurrent`
  public void removeCurrent()
  Remove the current element from this sequence.
  **Precondition:**
  `isCurrent()` returns true.
  **Postcondition:**
  The current element has been removed from this sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.
  **Throws:** `IllegalStateException`
  Indicates that there is no current element, so `removeCurrent` may not be called. (continued)
(FIGURE 3.10 continued)

- **size**
  
  ```java
  public int size() {
  Accessor method to determine the number of elements in this sequence.
  }
  
  **Returns:**
  
  the number of elements in this sequence

- **start**
  
  ```java
  public void start() {
  Set the current element at the front of this sequence.
  }
  
  **Postcondition:**
  
  The front element of this sequence is now the current element (but if this sequence has no
  elements at all, then there is no current element).

- **trimToSize**
  
  ```java
  public void trimToSize() {
  Reduce the current capacity of this sequence to its actual size (i.e., the number of elements it
  contains).
  }
  
  **Postcondition:**
  
  This sequence's capacity has been changed to its current size.
  
  **Throws:** OutOfMemoryError
  Indicates insufficient memory for altering the capacity.

As an example, suppose a sequence contains four numbers, with the current element at `data[2]`. The instance variables of the object might appear as shown here:

```
data
  3 1.4 6 9 ...
[0] [1] [2] [3] [4] [5]
```

- **currentIndex**
- **manyItems**

In this example, the current element is at `data[2]`, so the `getCurrent()` method would return the number 6. At this point, if we called `advance()`, then `currentIndex` would increase to 3, and `getCurrent()` would then return the 9.

Normally, a sequence has a current element, and the instance variable `currentIndex` contains the location of that current element. But if there is no current element, then `currentIndex` contains the same value as `manyItems`. In the preceding example, if `currentIndex` was 4, then that would indicate that there is no current element. Notice that this value (4) is beyond the used part of the array (which stretches from `data[0]` to `data[3]`).