Chapter 3 The Vector

A *vector* is an ordered set of elements in which each element is associated with, and is accessible by, a nonnegative integer called its *index*. Any collection whose elements may be accessed by their positions or indices is said to be *random access*. A music CD, for example, is a random-access collection of tracks. Likewise, a DVD is a random-access collection of video clips.

3.1 Our Current Model

The UML class diagram that follows shows our first step toward the final model presented in Chapter 2.

Fig. 3.1. Our current software model

3.2 The Vector ADT

The vector ADT is shown below.

```
vector: a random-access collection of elements

operations:

         append(element) - Add a new element to the end of the
                           collection.
                 clear() - Make the collection empty.
       contains(element) - Does the collection contain the given
                           element?
        elementAt(index) - Access the element at the given index.
        indexOf(element) - What is the index of the given element?
insertAt(index, element) - Insert a new element at the given
                           index.
               isEmpty() - Is the collection empty?
         removeAt(index) - Remove the element at the given index.
         remove(element) - Remove the given element from the
                           collection.
 replace(index, element) - Replace the element at the given index
                           with a new element.
                  size() - How many elements are in the
                           collection?
```

It may have already occurred to you that a vector is very much like an array. A vector is so much like an array that we may think of a vector as an "intelligent" array, for we can do anything with a vector that we can do with an array, yet a vector offers something more in the way of convenience. Clearing an array, for example, requires a loop, whereas clearing a vector requires only an invocation of the `clear` operation, as the code below illustrates.

```
Integer[] array = new Integer[100];
int size = 0;                         // Keep track of the number
                                      // of elements in use.
.
. // Code that populates part of the array goes here.
.

Vector vec = new Vector();
.
. // Code that inserts some elements into the vector goes here.
.

// Clear the array.

for (int j = 0; j < size; j++)
    array[j] = null;
size = 0;

// Clear the vector.

vec.clear();
```

3.3 A Vector Data Structure

Because a vector *is* essentially an array, it makes sense to implement the vector ADT as an array data structure. An array container for the vector ADT appears below. The methods are commented using a *precondition*/*postcondition* format. A method's precondition states what must be true if the method is to perform its intended task. A method's postcondition states what will be true, provided the precondition was met, after the method has returned.

```
public class Vector
{
    private static final int DEFAULT_CAPACITY = 100;
    private Object[] data;
    private int numItems;   // Keep track of the number of
                            // elements in use.
    public Vector()
    {
        // Acquire the array.

        data = new Object[DEFAULT_CAPACITY];
    }

    /*
     precondition: We can add an element to the end of the
                   collection only if the array is not full.
                   If this condition is not met, the method
                   returns false.
    postcondition: An element has been added at the end of the
                   collection, the collection's size is one
                   greater, and true has been returned.
    */

    public boolean append(Object element)
    {
        if (isFull())
            return false;
        data[numItems++] = element;
        return true;
    }
```

```
/*
This method must do more than reset 'numItems'. It must also
overwrite each of the array's non-null references with null so
that Java's garbage collector can reclaim the objects that are
no longer being used by the container. The garbage collector
is a program whose job it is to reclaim memory that is no
longer being used by your program. The garbage collector keeps
an eye on each of your program's objects. When an object
becomes unreachable, the garbage collector destroys it.

 precondition: N/A
postcondition: All of the array's non-null references have
               been overwritten with null and the collection's
               size is zero.
*/

public void clear()
{
    for (int j = 0; j < numItems; j++)
        data[j] = null;
    numItems = 0;
}

/*
 precondition: N/A
postcondition: If the target element is a member of the
               collection, true has been returned. Otherwise
               false has been returned.
*/

public boolean contains(Object element)
{
    return indexOf(element) != -1;
}

/*
 precondition: The given index must be between 0 and
               size() – 1. Otherwise the method returns
               null.
postcondition: The object with the given index has been
               returned.
*/

public Object elementAt(int index)
{
    if (index < 0 || index > numItems - 1)
        return null;
    return data[index];
}

/*
 precondition: N/A
postcondition: If the target element is a member of the
               collection, its index has been returned.
               Otherwise -1 has been returned.
*/

public int indexOf(Object element)
{
    for (int j = 0; j < numItems; j++)
        if (element.equals(data[j]))
            return j;
    return -1;
}
```

```
/*
 precondition: The given index must be between 0 and
               size() – 1, and the collection must not be
               full. If these conditions are not met, the
               method returns false.
postcondition: The given element has been inserted at the
               given index. The array elements at this index
               and beyond have been shifted away from index 0
               to make room for the new element. Also, the
               collection's size is one greater and true has
               been returned.
*/

public boolean insertAt(int index, Object element)
{
    if (index < 0 || index > numItems - 1 || isFull())
        return false;

    // The following loop makes room for the new element.

    for (int j = numItems - 1; j >= index; j--)
        data[j + 1] = data[j];
    data[index] = element;
    numItems++;
    return true;
}

/*
 precondition: N/A
postcondition: If the collection is empty, true has been
               returned. Otherwise false has been returned.
*/

public boolean isEmpty()
{
    return numItems == 0;
}

/*
This operation was not part of the vector ADT but is
appropriate for this implementation. See methods append()
and insertAt().

 precondition: N/A
postcondition: If the container's array is full, true has been
               returned. Otherwise false has been returned.
*/

public boolean isFull()
{
    return numItems == data.length;
}
```

```
/*
 precondition: The given index is between 0 and size() - 1. If
               this condition is not met, the method returns
               null.
postcondition: All objects at the given index and beyond have
               been shifted toward index 0, overwriting the
               reference of the element to be removed. Also,
               the collection's size is one less and the
               removed element has been returned.
*/

public Object removeAt(int index)
{
    if (index < 0 || index > numItems - 1)
        return null;
    Object temp = data[index];
    while (index < numItems - 1)
    {
        data[index] = data[index + 1];
        index++;
    }
    data[--numItems] = null;
    return temp;
}

/*
 precondition: The given element cannot be removed if it is
               not a member of the collection, in which case
               the method returns false.
postcondition: The given element has been removed from the
               collection, the collection's size is one less,
               and true has been returned.
*/

public boolean remove(Object element)
{
    int pos = indexOf(element);
    if (pos == -1)
        return false;
    removeAt(pos);
    return true;
}

/*
 precondition: The given index must be between 0 and
               size() - 1. If this condition is not met, the
               method returns false.
postcondition: The element at the given index has been
               replaced with the method's second argument and
               true has been returned.
*/

public boolean replace(int index, Object element)
{
    if (index < 0 || index > numItems - 1)
        return false;
    data[index] = element;
    return true;
}
```

```
        /*
         precondition: N/A
        postcondition: The number of elements in the collection has
                       been returned.
        */

        public int size()
        {
            return numItems;
        }
    }
```

3.4 Class `Object` and Type Independence

The `Vector` class is a *wrapper* for an array of type `Object`, just as class `Integer` is a wrapper for type `int`. Java's `Object` class has a special status: `Object` generalizes every other Java class, i.e., an instance of any Java class *is-a* `Object`. Thus an instance of `Vector` *may contain elements of any class type*! This feature of class `Vector` is called *type independence*. The type independence of class `Vector` makes `Vector` versatile; were it not for class `Object`'s special status, we would have no choice but to code a separate container for each type of element that we intended to store.

Exercises

1. Add attributes and operations to the model presented at the beginning of the chapter.
2. Class `Vector` has a dual relationship with class `Object`. Explain.
3. Suppose you have written an application that makes use of class `Vector`. Your supervisor informs you that `Vector`'s default capacity of 100 is insufficient given the application's intended use. What must you do to make your application meet the new requirement?
4. The code below will not compile with a Java 1.4 compiler but will compile with a Java 1.5 compiler. What feature of Java 1.5 makes this possible? What is the actual type of the `Vector`'s elements? Rewrite the code so that it will compile on Java 1.4.

```
Vector vec = new Vector();
Random rand = new Random();
for (int j = 0; j < 50; j++)
    vec.append(rand.nextInt());
```