

## Vectors

The size of an array is set when the array is declared, and the array cannot grow or shrink thereafter. This presents a problem when data are read from an external source and the number of data items is not known in advance. We used a work-around earlier by declaring a "larger than necessary" array and then maintaining a variable to hold the "actual" number of elements initialized. Obviously, this is wasteful. It is also possible that someday more data items are read than anticipated. In this case the program simply won't work unless the source code is updated and the program is recompiled. These problems are averted using a *dynamic array*.

Java's `Vector` class implements a dynamic array of objects. Like an array, it contains elements accessible using a simple integer index. However, the size of a `Vector` object can grow or shrink to accommodate adding and removing items on an as-needed basis. Of course, there is behind-the-scenes storage management taking place, but this is transparent to us.

By way of introduction, we'll digress briefly with an example to convince you of the need for dynamic arrays. Let's revisit two programs shown earlier: `RandomGen` and `NumberStats`. The `RandomGen` program outputs random numbers to the standard output stream, and the `NumberStats` program analyses data read from the standard input stream. That's great because we can connect the two programs using a pipe. (Pipes were presented earlier.) A sample dialogue follows:

```
PROMPT>java RandomGen 100 400.0 600.0 | java NumberStats
N = 100
Minimum = 402.17233767914325
Maximum = 599.6718556474029
Mean = 506.783572075286
Standard deviation = 55.920094678999526
```

In the first part of the command line, the `RandomGen` program generates 100 random floating-point numbers ranging from 400 to 600. Normally these values are sent to the host system's CRT display; however, the pipe symbol ( `|` ) effectively "catches" the output and sends it as input to the `NumberStats` program (via the `java` interpreter). The output is not surprising given our understanding of these programs. However, if we are interested in gathering statistics on, say, 150 numbers, then a problem is lurking. Here is another sample dialogue:

```
PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
at NumberStats.main(Compiled Code)
```

The problem lies in the `NumberStats` program. If you recall, the `numbers` array was declared with a default size of 100 (see line 13 `NumberStats.java`). As soon as the 101<sup>st</sup> element arrived via the pipe, the program crashed with an "array index out of bounds" exception. The most obvious fix is to edit the source code and recompile with yet another "larger than necessary" array size. But this is a nuisance.<sup>1</sup>

---

<sup>1</sup> It is possible to create pseudo-dynamic arrays as follows. When the capacity of the original array is reached, a new, larger array is declared and the elements from the old array are copied into the new array. The old array is discarded. If the new array also reaches its capacity, the process is repeated, and so on.

## Dynamic Arrays of Primitive Data Types

An alternative approach, which we will explore now, is to re-work the `NumberStats` program using the `Vector` class, instead of arrays. A key difference between `Vector` objects and arrays is that the former only hold objects. To store primitive data types, like integers, characters, or floating-point numbers, the values must be encoded as wrapper-class objects. (Java's wrapper classes were presented earlier.)

An object of the `Vector` class is instantiated like an object of any other class — using a constructor method. Once instantiated, elements are added and retrieved using instance methods. The `Vector` class methods are summarized in Table 1.

Table 1. Methods of the `Vector` Class

Method	Description
Constructor	
<code>Vector()</code>	constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero; returns a reference to the new object
Instance Methods	
<code>addElement(Object obj)</code>	adds the specified element to the end of this vector, increasing its size by one; returns a <code>void</code>
<code>add(int index, Object o)</code>	Adds the specified element at the specified position to the this vector; returns a <code>void</code>
<code>add(Object o)</code>	Adds the specified element to the end of this vector; returns a <code>boolean</code>
<code>elementAt(int index)</code>	returns an <code>Object</code> representing the element at the specified index (same as <code>get()</code> )
<code>get(int index)</code>	returns an <code>Object</code> representing the element at the specified index. (same as <code>elementAt()</code> )
<code>removeElementAt(int index)</code>	deletes the element at the specified index; returns a <code>void</code>
<code>size()</code>	returns an <code>int</code> equal to the number of elements in this vector

Let's assume we need a dynamic array (a `Vector`) of integers. The follow statement instantiates a `Vector` object named `samples`:

```
Vector samples = new Vector();
```

To add an element to the vector, we use the `addElement()` method. However, as shown in Table 1, the argument must be an `Object`. Since all Java classes are subclasses of the `Object` class, we can use any object reference as an argument. We cannot, however, use a primitive data type. If we wish to add an `int` to `samples`, it first must be "wrapped" into an `Integer` object. If, for example, the value of the integer is 5, it is instantiated as an `Integer` object using

```
Integer x = new Integer(5);
```

and then added to `samples` using the `addElement()` instance method:

```
samples.addElement(x);
```

Of course, we have no use for the variable `x`, so we can combine the two steps above:

```
samples.addElement(new Integer(5));
```

Note that the `Integer` constructor also accepts a string representation of an integer as an argument. For example

```
samples.addElement(new Integer("5"));
```

Additional elements are added to `samples` in a similar manner. The best part is this: We need not concern ourselves with the capacity of `samples`. Extra space is allocated on an as-needed basis.

To retrieve an element from `samples`, we use the `elementAt()` instance method. The argument is a simple integer index specifying the location of the element we wish to retrieve. Once again, we have a minor problem because `samples` is a `Vector` object containing a dynamic array of `Object` objects. We cannot retrieve an element and assign it to an `int` variable:

```
int y = samples.elementAt(i); // Wrong!
```

The `elementAt()` method returns an `Object` (actually, a reference to an `Object`). We can assign it to an `Integer` wrapper class object, but only if the `Object` reference is cast to an `Integer` reference:<sup>2</sup>

```
Integer temp = (Integer)samples.elementAt(i);
```

That's half the story. What we really want is to retrieve an element and assign it to an `int`. So, we follow the statement above with

```
int y = temp.intValue();
```

We can avoid instantiating the `Integer` object `temp` by combining the two preceding statements:

```
int x = ((Integer)samples.elementAt(i)).intValue();
```

The extra parentheses are needed to ensure the operations occur in the correct order.

At any time, we can retrieve the number of elements stored in the vector using the `size()` method:

```
for (int i = 0; i < samples.size(); i++)  
    ...
```

So, unlike our experience with arrays, we do not need a separate variable representing the number of initialized entries.

---

<sup>2</sup> We did not cast the `Integer` reference to a `Object` reference when using `addElement()` because of the inheritance relationship between the `Integer` (subclass) and `Object` (superclass) classes: an `Integer` object is an `Object` object (always!). However, we must cast the `Object` reference to an `Integer` reference when going the other way using `elementAt()` because an `Object` object is not necessarily a `Integer` object.

Let's put these pieces together in a new version of the `NumberStats` program that uses dynamic arrays. The program `NumbersStats2` inputs any number of floating point numbers from the standard input and outputs some simple statistics on these numbers (see Figure 1).

```

1  import java.io.*;
2  import java.util.*;
3
4  public class NumberStats2
5  {
6      public static void main(String[] args) throws IOException
7      {
8          BufferedReader stdin =
9              new BufferedReader(new InputStreamReader(System.in), 1);
10
11         // declare 'v' as a dynamic array (a Vector object)
12         Vector v = new Vector();
13
14         // input data and put into dynamic array
15         String line;
16         while ((line = stdin.readLine()) != null)
17         {
18             // prepare to tokenize line
19             StringTokenizer st = new StringTokenizer(line, " ,\t");
20
21             // process tokens in line
22             while (st.hasMoreTokens())
23             {
24                 String s = st.nextToken(); // get a token
25                 Double d = new Double(s); // convert and wrap as Double
26                 v.addElement(d);          // add to dynamic array
27             }
28         }
29
30         // declare a double array with exactly the size needed
31         double[] numbers = new double[v.size()];
32
33         // copy and convert elements of Vector object into double array
34         for (int i = 0; i < v.size(); i++)
35             numbers[i] = ((Double)v.elementAt(i)).doubleValue();
36
37         // output some statistics on the data
38         System.out.println("N = " + numbers.length);
39         System.out.println("Minimum = " + min(numbers));
40         System.out.println("Maximum = " + max(numbers));
41         System.out.println("Mean = " + mean(numbers));
42         System.out.println("Standard deviation = " + sd(numbers));
43     }
44
45     // find the minimum value in an array
46     public static double min(double[] n)
47     {
48         double min = n[0];
49         for (int j = 1; j < n.length; j++)
50             if (n[j] < min)
51                 min = n[j];
52         return min;
53     }
54
55     // find the maximum value in a array
56     public static double max(double[] n)
57     {
58         double max = n[0];
59         for (int j = 1; j < n.length; j++)
60             if (n[j] > max)
61                 max = n[j];
62         return max;

```

```

63     }
64
65     // calculate the mean of the values in an array
66     public static double mean(double n[])
67     {
68         double mean = 0.0;
69         for (int j = 0; j < n.length; j++)
70             mean += n[j];
71         return mean / n.length;
72     }
73
74     // calculate the standard deviation of values in an array
75     public static double sd(double[] n)
76     {
77         double m = mean(n);
78         double t = 0.0;
79         for (int j = 0; j < n.length; j++)
80             t += (m - n[j]) * (m - n[j]);
81         return Math.sqrt(t / (n.length - 1.0));
82     }
83 }

```

Figure 1. NumberStats2.java

Now we can do what our original version of NumberStats could not:

```

PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats2
N = 150
Minimum = 400.4034372537817
Maximum = 598.4001802690623
Mean = 498.7361024890425
Standard deviation = 55.27611100869566

```

A `Vector` object named `v` is declared in line 12. Values are read from the standard input, as before, except now they are placed in the dynamic array `v` instead of in an over-sized `double` array. This involves converting the inputted `String` to a `Double` object (line 25) and then inserting it into the dynamic array using the `addElement()` method of the `Vector` class (line 26). Once all the elements are read, we can declare a `double` array with precisely the space need. This is done in line 31 using the `size()` method of the `Vector` class to specified the size of a `double` array named `numbers`.

For serious number crunching, it is much more efficient to work with a `double` array than with a vector containing a dynamic array of `Double` objects. So, the elements of the `Vector` object are retrieved, converted, and copied into the `double` array `numbers` (lines 34-35). The expression

```
((Double)v.elementAt(i)).doubleValue()
```

performs the convoluted task of retrieving the element at position `i`, casting it to a `Double` object, and converting it to a `double`. The result is assigned to position `i` of the `double` array `numbers`.

At this point, it appears we are in basically the same position as in the original program, `NumberStats`. We have a `double` array of values, and we wish to calculate some simple statistics on the values. In fact, we are in a much better position. Because `numbers` is precisely the correct size, we do not need a separate variable for the "actual" number of initialized elements in the array: All elements are initialized! We can now work with the `length` field of the array

— a much more elegant approach. All the statistics methods are modified slightly with this improvement. It is no longer necessary to pass the "size" of the array to the statistics methods. We simply pass the array reference and the for loops are setup using the length field of the array.

### **Dynamic Arrays of Objects**

In the preceding example, the `Vector` class came to the rescue. We inputted an unspecified number of floating point values and created a `double` array with precisely the space needed. A dynamic array of `Vector` objects was used as an intermediate step, between inputting the values and creating the `double` array. However, the values had to be "wrapped" and "unwrapped" because the `Vector` class only works with objects of the `Object` class. This was inconvenient, but it was well worth the extra effort.

If the data of interest are objects (as opposed to primitive data types), then the `Vector` class is simple and natural to work with. This is illustrated in the following example working with `String` objects. The program `MedalWinners` uses a `Vector` object to hold the names of the medal winners in an Olympic event (see Figure 2).

```
1 import java.util.*;
2
3 public class MedalWinners
4 {
5     private static final String[] MEDAL = { "GOLD", "SILVER", "BRONZE" };
6
7     public static void main(String[] args)
8     {
9         // declare 'topThree' as a Vector object (a dynamic array)
10        Vector topThree = new Vector();
11
12        // add entries (top three finishers in men's 100 meter event)
13        topThree.addElement("Ben Johnson (CAN), 9.79 s");
14        topThree.addElement("Carl Lewis (USA), 9.92 s");
15        topThree.addElement("Linford Christie (GBR), 9.97 s");
16
17        // print results
18        System.out.println("1988 Seoul Olympics, Men's 100 M");
19        for (int i = 0; i < topThree.size(); i++)
20            System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));
21
22        // remove Ben Johnson (disqualified)
23        topThree.removeElementAt(0);
24
25        // add new bronze medal winner
26        topThree.addElement("Calvin Smith (USA), 9.99 s");
27
28        // print revised results
29        System.out.println();
30        System.out.println("1988 Seoul Olympics, Men's 100 M (revised)");
31        for (int i = 0; i < topThree.size(); i++)
32            System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));
33    }
34 }
```

Figure 2. `MedalWinners.java`

This program generates the following output:

```
1988 Seoul Olympics, Men's 100 M
GOLD:   Ben Johnson (CAN), 9.83 s
SILVER: Carl Lewis (USA), 9.92 s
BRONZE: Linford Christie (GBR), 9.97 s
```

```
1988 Seoul Olympics, Men's 100 M (revised)
GOLD:   Carl Lewis (USA), 9.92 s
SILVER: Linford Christie (GBR), 9.97 s
BRONZE: Calvin Smith (USA), 9.99 s
```

And the rest is history!

A `Vector` class object — a dynamic array — named `topThree` is declared in line 10. Three `String` objects are added to the array in lines 13-15. The `addElement()` method requires an `Object` as an argument, however an object of any class is a valid argument because all classes are subclasses of `Object`. As each element is added, the size of the dynamic array increases by one. The three elements are printed in lines 19-20, and you can see the effect in the first part of the program's output above.

In the print statement in line 18, the following expression appears:

```
topThree.elementAt(i)
```

The `elementAt()` method returns an `Object` reference; however, the `Object` is automatically converted to a string because of the concatenation operator. Be aware, however, that the following attempt to retrieve the gold-medal winner would generate a compile error:

```
String goldMedalWinner = topThree.elementAt(0); // Wrong!
```

The `Object` returned by the `elementAt()` method cannot be assigned to a `String` object unless it is cast:

```
String goldMedalWinner = (String)topThree.elementAt(0); // OK!
```

The statement above is good, clean Java code. Unfortunately, Mr. Johnson was not as clean. His use of metabolic steroids necessitated his removal from the Olympic record book. And so, too, we remove his entry from the `topThree` array in line 23 using the `removeElementAt()` method of the `Vector` class. The size of the array is automatically reduced by one. The previous Silver and Bronze medal winners are "bumped up" by one, and a new Bronze medal winner is added at the end of the array in line 26. The size of the array is then automatically increased by one. The final standings are outputted and appear in the last part of the program's output.