Chapter 3

Vectors

Concepts:
- ▷ Vectors
- ▷ Extending arrays
- ▷ Matrices

Climb high, climb far,
your goal the sky, your aim the star.
—Inscription on a college staircase

THE BEHAVIOR OF A PROGRAM usually depends on its input. Suppose, for example, that we wish to write a program that reads in n `String` values. One approach would keep track of the n values with n `String` variables:

StringReader

```
public static void main(String args[])
{
    // read in n = 4 strings
    Scanner s = new Scanner(System.in);
    String v1, v2, v3, v4;
    v1 = s.next();  // read a space-delimited word
    v2 = s.next();
    v3 = s.next();
    v4 = s.next();
}
```

This approach is problematic for the programmer of a *scalable* application—an application that works with large sets of data as well as small. As soon as n changes from its current value of 4, it has to be rewritten. Scalable applications are not uncommon, and so we contemplate how they might be supported.

One approach is to use *arrays*. An array of n values acts, essentially, as a collection of similarly typed variables whose names can be computed at run time. A program reading n values is shown here:

```
public static void main(String args[])
{
    // read in n = 4 strings
    Scanner s = new Scanner(System.in);
    String data[];
    int n = 4;
    // allocate array of n String references:
    data = new String[n];
    for (int i = 0; i < n; i++)
```

```
        {
            data[i] = s.next();
        }
    }
```

Here, n is a constant whose value is determined at compile time. As the program starts up, a new array of n integers is constructed and referenced through the variable named `data`.

All is fine, unless you want to read a different number of values. Then n has to be changed, and the program must be recompiled and rerun. Another solution is to pick an *upper bound* on the length of the array and only use the portion of the array that is necessary. Here's a modified procedure that uses up to one million array elements:

```
    public static void main(String args[])
    {
        // read in up to 1 million Strings
        Scanner s = new Scanner(System.in);
        String data[];
        int n = 0;
        data = new String[1000000];
        // read in strings until we hit end of file
        while (s.hasNext())
        {
            data[n] = s.next();
            n++;
        }
    }
```

Unfortunately, if you are running your program on a small machine and have small amounts of data, you are in trouble (see Problem 3.9). Because the array is so large, it will not fit on your machine—even if you want to read small amounts of data. You have to recompile the program with a smaller upper bound and try again. All this seems rather silly, considering how simple the problem appears to be.

We might, of course, require the user to specify the maximum size of the array before the data are read, at *run time*. Once the size is specified, an appropriately sized array can be allocated. While this may appear easier to program, the burden has shifted to the *user* of the program: the user has to commit to a specific upper bound—beforehand:

```
    public static void main(String args[])
    {
        // read in as many Strings as demanded by input
        Scanner s = new Scanner(System.in);
        String data[];
        int n;
        // read in the number of strings to be read
        n = s.nextInt();
        // allocate references for n strings
        data = new String[n];
        // read in the n strings
        for (int i = 0; i < n; i++)
        {
            data[i] = s.next();
        }
    }
```

A nice solution is to build a *vector*—an array whose size may easily be changed. Here is our `String` reading program retooled one last time, using `Vector`s:

```
    public static void main(String args[])
    {
        // read in an arbitrary number of strings
        Scanner s = new Scanner(System.in);
        Vector data;
        // allocate vector for storage
        data = new Vector();
        // read strings, adding them to end of vector, until eof
        while (s.hasNext())
        {
            String st = s.next();
            data.add(st);
        }
    }
```

The `Vector` starts empty and expands (using the `add` method) with every `String` read from the input. Notice that the program doesn't explicitly keep track of the number of values stored in `data`, but that the number may be determined by a call to the `size` method.

3.1 The Interface

The semantics of a `Vector` are similar to the semantics of an array. Both can store multiple values that may be accessed in any order. We call this property *random access*. Unlike the array, however, the `Vector` starts empty and is extended to hold object references. In addition, values may be removed from the `Vector` causing it to shrink. To accomplish these same size-changing operations in an array, the array would have to be reallocated.

With these characteristics in mind, let us consider a portion of the "interface"[1] for this structure:

`Vector`

[1] Stricktly speaking, constructors cannot be specified in formal Java`interfaces`. Nonetheless, adopt a convention of identifying constructors as part of the public view of structures (often called the *Application Program Interface* or *API*).

```
public class Vector extends AbstractList implements Cloneable
{
    public Vector()
    // post: constructs a vector with capacity for 10 elements

    public Vector(int initialCapacity)
    // pre: initialCapacity >= 0
    // post: constructs an empty vector with initialCapacity capacity

    public void add(Object obj)
    // post: adds new element to end of possibly extended vector

    public Object remove(Object element)
    // post: element equal to parameter is removed and returned

    public Object get(int index)
    // pre: 0 <= index && index < size()
    // post: returns the element stored in location index

    public void add(int index, Object obj)
    // pre: 0 <= index <= size()
    // post: inserts new value in vector with desired index,
    //   moving elements from index to size()-1 to right

    public boolean isEmpty()
    // post: returns true iff there are no elements in the vector

    public Object remove(int where)
    // pre: 0 <= where && where < size()
    // post: indicated element is removed, size decreases by 1

    public Object set(int index, Object obj)
    // pre: 0 <= index && index < size()
    // post: element value is changed to obj; old value is returned

    public int size()
    // post: returns the size of the vector
}
```

First, the constructors allow construction of a `Vector` with an optional initial *capacity*. The capacity is the initial number of `Vector` locations that are reserved for expansion. The `Vector` starts empty and may be freely expanded to its capacity. At that point the `Vector`'s memory is reallocated to handle further expansion. While the particulars of memory allocation and reallocation are hidden from the user, there is obvious benefit to specifying an appropriate initial capacity.

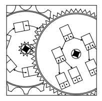
The one-parameter `add` method adds a value to the end of the `Vector`, expanding it. To insert a new value in the middle of the `Vector`, we use the two-parameter `add` method, which includes a location for insertion. To access an existing element, one calls `get`. If `remove` is called with an `Object`, it removes at most one element, selected by value. Another `remove` method shrinks the logical size of the `Vector` by removing an element from an indicated location. The `set` method is used to change a value in the `Vector`. Finally, two methods provide feedback about the current logical size of the `Vector`: `size` and `isEmpty`. The `size` method returns the number of values stored within the `Vector`. As elements are added to the `Vector`, the size increases from zero up to the capacity of the `Vector`. When the size is zero, then `isEmpty` returns `true`. The result is a data structure that provides constant-time access to data within the structure, without concern for determining explicit bounds on the structure's size.

There are several ways that a `Vector` is different than its array counterpart. First, while both the array and `Vector` maintain a number of references to objects, the `Vector` typically grows with use and stores a non-`null` reference in each entry. An array is a static structure whose entries may be initialized and used in any order and are often `null`. Second, the `Vector` has an *end* where elements can be appended, while the array does not directly support the concept of appending values. There are times, of course, when the append operation might not be a feature desired in the structure; either the `Vector` or array would be a suitable choice.

The interface for `Vectors` in the `structure` package was driven, almost exclusively, by the interface for Java's proprietary `java.util.Vector` class. Thus, while we do not have access to the code for that class, any program written to use Java's `Vector` class can be made to use the `Vector` class described here; their interfaces are consistent.

3.2 Example: The Word List Revisited

We now reconsider an implementation of the word list part of our Hangman program of Section 1.6 implemented directly using `Vectors`:



WordList

```
Vector list;
String targetWord;
java.util.Random generator = new java.util.Random();
list = new Vector(10);
list.add("clarify");
list.add("entered");
list.add("clerk");
while (list.size() != 0)
{
    {   // select a word from the list
        int index = Math.abs(generator.nextInt())%list.size();
        targetWord = (String)list.get(index);
    }
    // ... play the game using targetWord ...
    list.remove(targetWord);
}
```

Here, the operations of the Vector are seen to be very similar to the operations of the WordList program fragment shown on page 19. The Vector class, however, does not have a selectAny method. Instead, the bracketed code accomplishes that task. Since only Strings are placed within the Vector, the assignment of targetWord involves a cast from Object (the type of value returned from the get method of Vector) to String. This cast is necessary for Java to be reassured that you're expecting an element of type String to be returned. If the cast were not provided, Java would complain that the types involved in the assignment were incompatible.

Now that we have an implementation of the Hangman code in terms of both the WordList and Vector structures, we can deduce an implementation of the WordList structure in terms of the Vector class. In this implementation, the WordList contains a Vector that is used to hold the various words, as well as the random number generator (the variable generator in the code shown above). To demonstrate the implementation, we look at the implementation of the WordList's constructor and selectAny method:

```
protected Vector theList;
protected java.util.Random generator;

public WordList(int n)
{
    theList = new Vector(n);
    generator = new java.util.Random();
}

public String selectAny()
{
    int i = Math.abs(generator.nextInt())%theList.size();
    return (String)theList.get(i);
}
```

Clearly, the use of a Vector within the WordList is an improvement over the direct use of an array, just as the use of WordList is an improvement over the complications of directly using a Vector in the Hangman program.

3.3 Example: Word Frequency

Suppose one day you read a book, and within the first few pages you read "behemoth" twice. A mighty unusual writing style! Word frequencies within documents can yield interesting information.[2] Here is a little application for computing the frequency of words appearing on the input:

WordFreq

```
public static void main(String args[])
```

[2] Recently, using informal "literary forensics," Don Foster has identified the author of the anonymously penned book *Primary Colors* and is responsible for new attributions of poetry to Shakespeare. Foster also identified Major Henry Livingston Jr. as the true author of "The Night Before Christmas."

```
{
    Vector vocab = new Vector(1000);
    Scanner s = new Scanner(System.in);
    int i;

    // for each word on input
    while (s.hasNext())
    {
        Association wordInfo; // word-frequency association
        String vocabWord;     // word in the list

        // read in and tally instance of a word
        String word = s.next();
        for (i = 0; i < vocab.size(); i++)
        {
            // get the association
            wordInfo = (Association)vocab.get(i);
            // get the word from the association
            vocabWord = (String)wordInfo.getKey();
            if (vocabWord.equals(word))
            {   // match: increment integer in association
                Integer f = (Integer)wordInfo.getValue();
                wordInfo.setValue(new Integer(f.intValue() + 1));
                break;
            }
        }
        // mismatch: add new word, frequency 1.
        if (i == vocab.size())
        {
            vocab.add(new Association(word,new Integer(1)));
        }
    }
    // print out the accumulated word frequencies
    for (i = 0; i < vocab.size(); i++)
    {
        Association wordInfo = (Association)vocab.get(i);
        System.out.println(
           wordInfo.getKey()+" occurs "+
           wordInfo.getValue()+" times.");
    }
}
```

First, for each word found on the input, we maintain an `Association` between the word (a `String`) and its frequency (an `Integer`). Each element of the `Vector` is such an `Association`. Now, the outer loop at the top reads in each word. The inner loop scans through the `Vector` searching for matching words that might have been read in. Matching words have their values updated. New words cause the construction of a new `Association`. The second loop scans through the `Vector`, printing out each of the `Associations`.

Each of these applications demonstrates the most common use of `Vector`s—keeping track of data when the number of entries is not known far in advance. When considering the `List` data structure we will consider the efficiency of these algorithms and, if necessary, seek improvements.

3.4 The Implementation

Clearly, the `Vector` must be able to store a large number of similar items. We choose, then, to have the implementation of the `Vector` maintain an array of `Object`s, along with an integer that describes its current *size* or *extent*. When the size is about to exceed the *capacity* (the length of the underlying array), the `Vector`'s capacity is increased to hold the growing number of elements.

The constructor is responsible for allocation of the space and initializing the local variables. The number of elements initially allocated for expansion can be specified by the user:

`Vector`

```
protected Object elementData[];     // the data
protected int elementCount;         // number of elements in vector

public Vector()
// post: constructs a vector with capacity for 10 elements
{
    this(10); // call one-parameter constructor
}

public Vector(int initialCapacity)
// pre: initialCapacity >= 0
// post: constructs an empty vector with initialCapacity capacity
{
    Assert.pre(initialCapacity >= 0, "Initial capacity should not be negative.");
    elementData = new Object[initialCapacity];
    elementCount = 0;
}
```

Unlike other languages, all arrays within Java must be explicitly allocated. At the time the array is allocated, the number of elements is specified. Thus, in the constructor, the `new` operator allocates the number of elements desired by the user. Since the size of an array can be gleaned from the array itself (by asking for `elementData.length`), the value does not need to be explicitly stored within the `Vector` object.[3]

To access and modify elements within a `Vector`, we use the following operations:

```
public Object get(int index)
```

[3] It could, of course, but explicitly storing it within the structure would mean that the implementor would have to ensure that the stored value was always consistent with the value accessible through the array's `length` variable.

```
// pre: 0 <= index && index < size()
// post: returns the element stored in location index
{
    return elementData[index];
}

public Object set(int index, Object obj)
// pre: 0 <= index && index < size()
// post: element value is changed to obj; old value is returned
{
    Object previous = elementData[index];
    elementData[index] = obj;
    return previous;
}
```

The arguments to both methods identify the location of the desired element. Because the index should be within the range of available values, the precondition states this fact.

For the accessor (`get`), the desired element is returned as the result. The `set` method allows the `Object` reference to be changed to a new value and returns the old value. These operations, effectively, translate operations on `Vector`s into operations on arrays.

Now consider the addition of an element to the `Vector`. One way this can be accomplished is through the use of the one-parameter `add` method. The task requires extending the size of the `Vector` and then storing the element at the location indexed by the current number of elements (this is the first free location within the `Vector`). Here is the Java method:

```
public void add(Object obj)
// post: adds new element to end of possibly extended vector
{
    ensureCapacity(elementCount+1);
    elementData[elementCount] = obj;
    elementCount++;
}
```

(We will discuss the method `ensureCapacity` later. Its purpose is simply to ensure that the data array actually has enough room to hold the indicated number of values.) Notice that, as with many modern languages, arrays are indexed starting at zero. There are many good reasons for doing this. There are probably just as many good reasons for not doing this, but the best defense is that this is what programmers are currently used to.

Principle 6 *Maintaining a consistent interface makes a structure useful.*

If one is interested in inserting an element in the middle of the `Vector`, it is necessary to use the two-parameter `add` method. The operation first creates an unused location at the desired point by shifting elements out of the way. Once the opening is created, the new element is inserted.

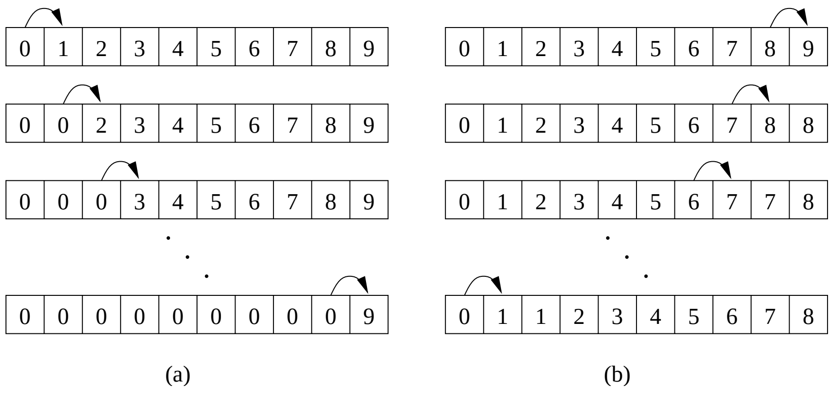


Figure 3.1 The incorrect (a) and correct (b) way of moving values in an array to make room for an inserted value.

```
public void add(int index, Object obj)
// pre: 0 <= index <= size()
// post: inserts new value in vector with desired index,
//   moving elements from index to size()-1 to right
{
    int i;
    ensureCapacity(elementCount+1);
    // must copy from right to left to avoid destroying data
    for (i = elementCount; i > index; i--) {
        elementData[i] = elementData[i-1];
    }
    // assertion: i == index and element[index] is available
    elementData[index] = obj;
    elementCount++;
}
```

Note that the loop that moves the elements higher in the array runs *backward*. To see why, it is only necessary to see what happens if the loop runs forward (see Figure 3.1a): the lowest element gets copied into higher and higher elements, ultimately copying over the entire `Vector` to the right of the insertion point. Figure 3.1b demonstrates the correct technique.

Removing an element from a specific location in the `Vector` is very similar, reversing the effect of `add`. Here, using an argument similar to the previous one, the loop moves in the forward direction:

```
public Object remove(int where)
// pre: 0 <= where && where < size()
// post: indicated element is removed, size decreases by 1
{
        Object result = get(where);
        elementCount--;
        while (where < elementCount) {
            elementData[where] = elementData[where+1];
            where++;
        }
        elementData[elementCount] = null; // free reference
        return result;
    }
```

We also allow the removal of a specific value from the `Vector`, by passing an example `Object` to `remove` (not shown). Within this code, the `equals` method of the value passed to the routine is used to compare it to values within the `Vector`. When (and if) a match is found, it is removed using the technique just described.

The methods having to do with size are relatively straightforward:

```
    public boolean isEmpty()
    // post: returns true iff there are no elements in the vector
    {
        return size() == 0;
    }

    public int size()
    // post: returns the size of the vector
    {
        return elementCount;
    }
```

The logical size of the `Vector` is the number of elements stored within the `Vector`, and it is empty when this size is zero.

3.5 Extensibility: A Feature

Sometimes, our initial estimate of the maximum number of values is too small. In this case, it is necessary to extend the capacity of the `Vector`, carefully maintaining the values already stored within the `Vector`. Fortunately, because we have packaged the implementation within an interface, it is only necessary to extend the functionality of the existing operations and provide some additional methods to describe the features.

A first approach might be to extend the `Vector` to include just as many elements as needed. Every time an element is added to the `Vector`, the number of elements is compared to the capacity of the array. If the capacity is used up, an array that is one element longer is allocated. This reallocation also requires copying of the existing data from one array to the other. Of course, for really long arrays, these copying operations would take a proportionally long time. Over time, as the array grows to n elements, the array data get copied many times. At the beginning, the array holds a single element, but it is expanded to hold two. The original element must be copied to the new space to complete the operation. When a third is added, the first two must be copied. The result is that

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

elements are copied as the array grows to size n. (Proving this last formula is the core of Problem 3.8.) This is expensive since, if in the beginning we had just allocated the `Vector` with a capacity of n elements, none of the data items would have to be copied during extension!

It turns out there is a happy medium: every time you extend the array, just double its capacity. Now, if we reconsider the number of times that an item gets copied during the extension process, the result is dramatically different. Suppose, for neatness only, that n is a power of 2, and that the `Vector` started with a capacity of 1. What do we know? When the `Vector` was extended from capacity 1 to capacity 2, one element was copied. When the array was extended from capacity 2 to capacity 4, two elements were copied. When the array was extended from capacity 4 to capacity 8, four elements were copied. This continues until the last extension, when the `Vector` had its capacity extended from $\frac{n}{2}$ to n: then $\frac{n}{2}$ elements had to be preserved. The total number of times elements were copied is

$$1 + 2 + 4 + \cdots + \frac{n}{2} = n - 1$$

Thus, extension by doubling allows unlimited growth of the `Vector` with an overhead that is proportional to the ultimate length of the array. Another way to think about it is that there is a constant overhead in supporting each element of a `Vector` extended in this way.

The Java language specifies a `Vector` interface that allows the user to specify how the `Vector` is to be extended if its capacity is not sufficient for the current operation. When the `Vector` is constructed, a `capacityIncrement` is specified. This is simply the number of elements to be added to the underlying array when extension is required. A nonzero value for this increment leads to the n^2 behavior we saw before, but it may be useful if, for example, one does not have the luxury of being able to double the size of a large array. If the increment is zero, the doubling strategy is used.

Our design, then, demands another protected value to hold the increment; we call this `capacityIncrement`. This value is specified in a special constructor and is not changed during the life of the `Vector`:

```
protected int capacityIncrement;    // the rate of growth for vector

public Vector(int initialCapacity, int capacityIncr)
// pre: initialCapacity >= 0, capacityIncr >= 0
// post: constructs an empty vector with initialCapacity capacity
//    that extends capacity by capacityIncr, or doubles if 0
{
    Assert.pre(initialCapacity >= 0 && capacityIncr >= 0,
               "Neither capacity nor increment should be negative.");
```

```
        elementData = new Object[initialCapacity];
        elementCount = 0;
        capacityIncrement = capacityIncr;
    }
```

We are now prepared to investigate `ensureCapacity`, a method that, if necessary, resizes `Vector` to have a capacity of at least `minCapacity`:

```
    public void ensureCapacity(int minCapacity)
    // post: the capacity of this vector is at least minCapacity
    {
        if (elementData.length < minCapacity) {
            int newLength = elementData.length; // initial guess
            if (capacityIncrement == 0) {
                // increment of 0 suggests doubling (default)
                if (newLength == 0) newLength = 1;
                while (newLength < minCapacity) {
                    newLength *= 2;
                }
            } else {
                // increment != 0 suggests incremental increase
                while (newLength < minCapacity)
                {
                    newLength += capacityIncrement;
                }
            }
            // assertion: newLength > elementData.length.
            Object newElementData[] = new Object[newLength];
            int i;
            // copy old data to array
            for (i = 0; i < elementCount; i++) {
                newElementData[i] = elementData[i];
            }
            elementData = newElementData;
            // garbage collector will (eventually) pick up old elementData
        }
        // assertion: capacity is at least minCapacity
    }
```

This code deserves careful investigation. If the current length of the underlying array is already sufficient to provide `minCapacity` elements, then the method does nothing. On the other hand, if the `Vector` is too short, it must be extended. We use a loop here that determines the new capacity by doubling (if `capacityIncrement` is zero) or by directly incrementing if `capacityIncrement` is nonzero. In either case, by the time the loop is finished, the desired capacity is determined. At that point, an array of the appropriate size is allocated, the old values are copied over, and the old array is dereferenced in favor of the new.

3.6 Example: L-Systems

In the late 1960s biologists began to develop computational models for growth. One of the most successful models, *L-systems*, was developed by Aristid Lindenmayer. An L-system consists of a seed or *start string* of symbols derived from an alphabet, along with a number of rules for changing or *rewriting* the symbols, called *productions*. To simulate an interval of growth, strings are completely rewritten using the productions. When the rewriting begins with the start string, it is possible to iteratively simulate the growth of a simple organism. To demonstrate the complexity of this approach, we can use an alphabet of two characters—S (for stem) and L (for leaf). If the two productions

Before	After
S	L
L	SL

are used, we can generate the following strings over 6 time steps:

Time	String
0	S
1	L
2	SL
3	LSL
4	SLLSL
5	LSLSLLSL
6	SLLSLLSLSLLSL

Although there are some observations that might be made (there are never two consecutive Ss), any notion of a pattern in this string quickly breaks down. Still, many organisms display patterns that are motivated by the seemingly simple production system.

We can use Vectors to help us perform this rewriting process. By constructing two Character objects, L and S, we can store patterns in a Vector of references. The rewriting process involves constructing a new result Vector. Here is a program that would verify the growth pattern suggested in the table:

LSystem

```
public class LSystem
{   // constants that define the alphabet
    final static Character L = new Character('L');
    final static Character S = new Character('S');

    public static Vector rewrite(Vector s)
    // pre: s is a string of L and S values
    // post: returns a string rewritten by productions
    {
        Vector result = new Vector();
        for (int pos = 0; pos < s.size(); pos++)
        {
            // rewrite according to two different rules
            if (S == s.get(pos)) {
                result.add(L);
            } else if (L == s.get(pos)) {
                result.add(S); result.add(L);
            }
        }
        return result;
    }

    public static void main(String[] args)
    {
        Vector string = new Vector();
        string.add(S);

        // determine the number of strings
        Scanner s = new Scanner(System.in);
        int count = s.nextInt();

        // write out the start string
        System.out.println(string);
        for (int i = 1; i <= count; i++)
        {
            string = rewrite(string);   // rewrite the string
            System.out.println(string); // print it out
        }
    }
}
```

L-systems are an interesting example of a *grammar system*. The power of a grammar to generate complex structures—including languages and, biologically, plants—is of great interest to theoretical computer scientists.

3.7 Example: Vector-Based Sets

In Section 1.8 we discussed Java's interface for a `Set`. Mathematically, it is an unordered collection of unique values. The set abstraction is an important feature of many algorithms that appear in computer science, and so it is important that we actually consider a simple implementation before we go much further.

As we recall, the `Set` is an extension of the `Structure` interface. It demands that the programmer implement not only the basic `Structure` methods (`add`, `contains`, `remove`, etc.), but also the following methods of a `Set`. Here is the interface associated with a `Vector`-based implementation of a `Set`:

```
public class SetVector extends AbstractSet
{
    public SetVector()
    // post: constructs a new, empty set
```

SetVector

```
    public SetVector(Structure other)
    // post: constructs a new set with elements from other

    public void clear()
    // post: elements of set are removed

    public boolean isEmpty()
    // post: returns true iff set is empty

    public void add(Object e)
    // pre: e is non-null object
    // post: adds element e to set

    public Object remove(Object e)
    // pre: e is non-null object
    // post: e is removed from set, value returned

    public boolean contains(Object e)
    // pre: e is non-null
    // post: returns true iff e is in set

    public boolean containsAll(Structure other)
    // pre: other is non-null reference to set
    // post: returns true iff this set is subset of other

    public Object clone()
    // post: returns a copy of set

    public void addAll(Structure other)
    // pre: other is a non-null structure
    // post: add all elements of other to set, if needed

    public void retainAll(Structure other)
    // pre: other is non-null reference to set
    // post: returns set containing intersection of this and other

    public void removeAll(Structure other)
    // pre: other is non-null reference to set
    // post: returns set containing difference of this and other

    public Iterator iterator()
    // post: returns traversal to traverse the elements of set

    public int size()
    // post: returns number of elements in set
}
```

A `SetVector` might take the approach begun by the `WordList` implementation we have seen in Section 3.2: each element of the `Set` would be stored in a location in the `Vector`. Whenever a new value is to be added to the `Set`, it is only added if the `Set` does not already contain the value. When values are removed from the `Set`, the structure contracts. At all times, we are free to keep the order of the data in the `Vector` hidden from the user since the ordering of the values is not part of the abstraction.

We construct a `SetVector` using the following constructors, which initialize a protected `Vector`:

```
protected Vector data; // the underlying vector

public SetVector()
// post: constructs a new, empty set
{
    data = new Vector();
}

public SetVector(Structure other)
// post: constructs a new set with elements from other
{
    this();
    addAll(other);
}
```

The second constructor is a *copy constructor* that makes use of the union operator, `addAll`. Since the initial set is empty (the call to `this()` calls the first constructor), the `SetVector` essentially picks up all the values found in the other structure.

Most methods of the `Set` are adopted from the underlying `Vector` class. For example, the `remove` method simply calls the `remove` method of the `Vector`:

```
public Object remove(Object e)
// pre: e is non-null object
// post: e is removed from set, value returned
{
    return data.remove(e);
}
```

The `add` method, though, is responsible for ensuring that duplicate values are not added to the `Set`. It must first check to see if the value is already a member:

```
public void add(Object e)
// pre: e is non-null object
// post: adds element e to set
{
    if (!data.contains(e)) data.add(e);
}
```

To perform the more complex `Set`-specific operations (`addAll` and others), we must perform the specified operation for all the values of the other set. To accomplish this, we make use of an `Iterator`, a mechanism we will not study until Chapter 8, but which is nonetheless simple to understand. Here, for example, is the implementation of `addAll`, which attempts to `add` all the values found in the `other` structure:

```
public void addAll(Structure other)
// pre: other is a non-null structure
// post: add all elements of other to set, if needed
{
    Iterator yourElements = other.iterator();
    while (yourElements.hasNext())
    {
        add(yourElements.next());
    }
}
```

Other methods are defined in a straightforward manner.

3.8 Example: The Matrix Class

One application of the `Vector` class is to support a two-dimensional `Vector`-like object: the *matrix*. Matrices are used in applications where two dimensions of data are needed. Our `Matrix` class has the following methods:

Matrix

```
public class Matrix
{
    public Matrix(int h, int w)
    // pre: h >= 0, w >= 0;
    // post: constructs an h row by w column matrix

    public Object get(int row, int col)
    // pre: 0 <= row < height(), 0 <= col < width()
    // post: returns object at (row, col)

    public void set(int row, int col, Object value)
    // pre: 0 <= row < height(), 0 <= col < width()
    // post: changes location (row, col) to value

    public void addRow(int r)
    // pre: 0 <= r < height()
    // post: inserts row of null values to be row r

    public void addCol(int c)
    // pre: 0 <= c < width()
    // post: inserts column of null values to be column c
```

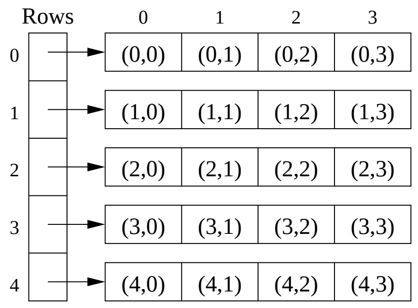


Figure 3.2 The `Matrix` class is represented as a `Vector` of rows, each of which is a `Vector` of references to `Objects`. Elements are labeled with their indices.

```
        public Vector removeRow(int r)
        // pre: 0 <= r < height()
        // post: removes row r and returns it as a Vector

        public Vector removeCol(int c)
        // pre: 0 <= c < width()
        // post: removes column c and returns it as a vector

        public int width()
        // post: returns number of columns in matrix

        public int height()
        // post: returns number of rows in matrix
    }
```

The two-parameter constructor specifies the width and height of the `Matrix`. Elements of the `Matrix` are initially `null`, but may be reset with the `set` method. This method, along with the `get` method, accepts two parameters that identify the row and the column of the value. To expand and shrink the `Matrix`, it is possible to insert and remove both rows and columns at any location. When a row or column is removed, a `Vector` of removed values is returned. The methods `height` and `width` return the number of rows and columns found within the `Matrix`, respectively.

To support this interface, we imagine that a `Matrix` is a `Vector` of rows, which are, themselves, `Vectors` of values (see Figure 3.2). While it is not strictly necessary, we explicitly keep track of the height and width of the `Matrix` (if we determine at some later date that keeping this information is unnecessary, the interface would hide the removal of these fields). Here, then, is the constructor for the `Matrix` class:

```
    protected int height, width; // size of matrix
```

```
protected Vector rows;       // vector of row vectors

public Matrix(int h, int w)
// pre: h >= 0, w >= 0;
// post: constructs an h row by w column matrix
{
    height = h;  // initialize height and width
    width = w;
    // allocate a vector of rows
    rows = new Vector(height);
    for (int r = 0; r < height; r++)
    {   // each row is allocated and filled with nulls
        Vector theRow = new Vector(width);
        rows.add(theRow);
        for (int c = 0; c < width; c++)
        {
            theRow.add(null);
        }
    }
}
```

We allocate a `Vector` for holding the desired number of rows, and then, for each row, we construct a new `Vector` of the appropriate width. All the elements are initialized to `null`. It's not strictly necessary to do this initialization, but it's a good habit to get into.

The process of manipulating individual elements of the matrix is demonstrated by the `get` and `set` methods:

```
public Object get(int row, int col)
// pre: 0 <= row < height(), 0 <= col < width()
// post: returns object at (row, col)
{
    Assert.pre(0 <= row && row < height, "Row in bounds.");
    Assert.pre(0 <= col && col < width, "Col in bounds.");
    Vector theRow = (Vector)rows.get(row);
    return theRow.get(col);
}

public void set(int row, int col, Object value)
// pre: 0 <= row < height(), 0 <= col < width()
// post: changes location (row, col) to value
{
    Assert.pre(0 <= row && row < height, "Row in bounds.");
    Assert.pre(0 <= col && col < width, "Col in bounds.");
    Vector theRow = (Vector)rows.get(row);
    theRow.set(col,value);
}
```

The process of manipulating an element requires looking up a row within the rows table and finding the element within the row. It is also important to notice

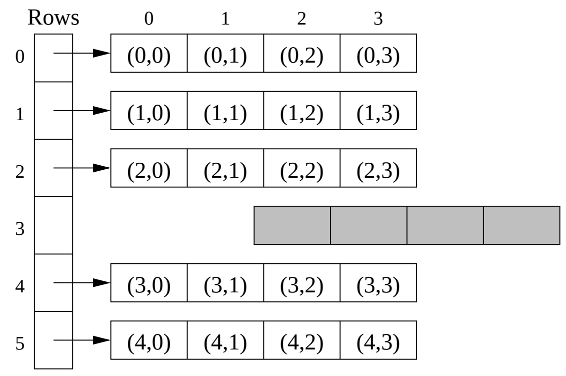


Figure 3.3 The insertion of a new row (gray) into an existing matrix. Indices are those associated with matrix *before* `addRow`. Compare with Figure 3.2.

that in the `set` method, the row is found using the `get` method, while the element within the row is changed using the `set` method. Although the element within the row changes, the row itself is represented by the same vector.

Many of the same memory management issues discussed in reference to `Vectors` hold as well for the `Matrix` class. When a row or column needs to be expanded to make room for new elements (see Figure 3.3), it is vital that the management of the arrays within the `Vector` class be hidden. Still, with the addition of a row into the `Matrix`, it is necessary to allocate the new row object and to initialize each of the elements of the row to `null`:

```
public void addRow(int r)
// pre: 0 <= r < height()
// post: inserts row of null values to be row r
{
    Assert.pre(0 <= r && r < width, "Row in bounds.");
    height++;
    Vector theRow = new Vector(width);
    for (int c = 0; c < width; c++)
    {
        theRow.add(null);
    }
    rows.add(r,theRow);
}
```

We leave it to the reader to investigate the implementation of other `Matrix` methods. In addition, a number of problems consider common extensions to the `Matrix` class.

3.9 Conclusions

Most applications that accept data are made more versatile by not imposing constraints on the number of values processed by the application. Because the size of an array is fixed at the time it is allocated, programmers find it difficult to create size-independent code without the use of extensible data structures. The `Vector` and `Matrix` classes are examples of extensible structures.

Initially `Vectors` are empty, but can be easily expanded when necessary. When a programmer knows the upper bound on the `Vector` size, this information can be used to minimize the amount of copying necessary during the entire expansion process. When a bound is not known, we saw that doubling the allocated storage at expansion time can reduce the overall copying cost.

The implementation of `Vector` and `Matrix` classes is not trivial. Data abstraction hides many important housekeeping details. Fortunately, while these details are complex for the implementor, they can considerably reduce the complexity of applications that make use of the `Vector` and `Matrix` structures.

Self Check Problems

Solutions to these problems begin on page 442.

3.1 How are arrays and `Vectors` the same? How do they differ?

3.2 What is the difference between the `add(v)` and `add(i,v)` methods of `Vector`?

3.3 What is the difference between the `add(i,v)` method and the `set(i,v)` method?

3.4 What is the difference between the `remove(v)` method (`v` is an `Object` value), and the `remove(i)` (`i` is an `int`)?

3.5 What is the distinction between the *capacity* and *size* of a `Vector`?

3.6 Why is the use of a `Vector` an improvement over the use of an array in the implementation of Hangman in Section 3.2?

3.7 When inserting a value into a `Vector` why is it necessary to shift elements to the right starting at the high end of the `Vector`? (See Figure 3.1.)

3.8 By default, when the size first exceeds the capacity, the capacity of the `Vector` is doubled. Why?

3.9 What is the purpose of the following code?

```
elementData = new Object[initialCapacity];
```

What can be said about the values found in `elementData` after this code is executed?

3.10 When there is more than one constructor for a class, when and how do we indicate the appropriate method to use? Compare, for example,

```
    Vector v = new Vector();
    Vector w = new Vector(1000);
```

3.11 Is the row index of the `Matrix` bounded by the matrix height or width? When indexing a `Matrix` which is provided first, the row or the column?

Problems

Solutions to the odd-numbered problems begin on page 457.

3.1 Explain the difference between the *size* and *capacity* of a vector. Which is more important to the user?

3.2 The default capacity of a `Vector` in a `structure` package implementation is 10. It could have been one million. How would you determine a suitable value?

3.3 The implementation of `java.util.Vector` provides a method `trimToSize`. This method ensures that the capacity of the `Vector` is the same as its size. Why is this useful? Is it possible to trim the capacity of a `Vector` without using this method?

3.4 The implementation of `java.util.Vector` provides a method `setSize`. This method explicitly sets the size of the `Vector`. Why is this useful? Is it possible to set the size of the `Vector` without using this method?

3.5 Write a `Vector` method, `indexOf`, that returns the index of an object in the `Vector`. What should the method return if no object that is `equals` to this object can be found? What does `java.lang.Vector` do in this case? How long does this operation take to perform, on average?

3.6 Write a class called `BitVector` that has an interface similar to `Vector`, but the values stored within the `BitVector` are all known to be `boolean` (the primitive type). What is the primary advantage of having a special-purpose vector, like `BitVector`?

3.7 Suppose we desire to implement a method `reverse` for the `Vector` class. One approach would be to `remove` location 0 and to use `add` near the end or *tail* of the `Vector`. Defend or reject this suggested implementation. In either case, write the best method you can.

3.8 Suppose that a precisely sized array is used to hold data, and that each time the array size is to be increased, it is increased by exactly one and the data are copied over. Prove that, in the process of growing an array incrementally from size 0 to size n, approximately n^2 values must be copied.

3.9 What is the maximum length array of `String`s you can allocate on your machine? (You needn't initialize the array.) What is the maximum length array of `boolean` you can allocate on your machine? What is to be learned from the ratio of these two values?

3.10 Implement the `Object`-based `remove` method for the `Vector` class.

3.11 In our discussion of L-systems, the resulting strings are always linear. Plants, however, often branch. Modify the `LSystem` program so that it includes the following five productions:

Before	After	Before	After	Before	After
`S`	`T`	`U`	`V`	`W`	`[S]U`
`T`	`U`	`V`	`W`		

where `[S]` is represented by a new `Vector` that contains a single `S`. (To test to see if an `Object`, `x`, is a `Vector`, use the test `x instanceof Vector`.)

3.12 Finish the two-dimensional `Vector`-like structure `Matrix`. Each element of the `Matrix` is indexed by two integers that identify the row and column containing the value. Your class should support a constructor, methods `addRow` and `addCol` that append a row or column, the `get` and `set` methods, and `width` and `height` methods. In addition, you should be able to use the `removeRow` and `removeCol` methods.

3.13 Write `Matrix` methods for `add` and `multiply`. These methods should implement the standard matrix operations from linear algebra. What are the preconditions that are necessary for these methods?

3.14 A `Matrix` is useful for nonmathematical applications. Suppose, for example, that the owners of cars parked in a rectangular parking lot are stored in a `Matrix`. How would you design a new `Matrix` method to return the location of a particular value in the `Matrix`? (Such an extension implements an *associative memory*. We will discuss associative structures when we consider `Dictionarys`.)

3.15 An $m \times n$ `Matrix` could be implemented using a single `Vector` with mn locations. Assuming that this choice was made, implement the `get` method. What are the advantages and disadvantages of this implementation over the `Vector` of `Vectors` approach?

3.16 A *triangular matrix* is a two-dimensional structure with n rows. Row i has $i+1$ columns (numbered 0 through i) in row i. Design a class that supports all the `Matrix` operations, except `addRow`, `removeRow`, `addCol`, and `removeCol`. You should also note that when a row and column must be specified, the row must be greater than or equal to the column.

3.17 A *symmetric matrix* is a two-dimensional `Matrix`-like structure such that the element at $[i][j]$ is the same element found at $[j][i]$. How would you implement each of the `Matrix` operations? The triangular matrix of Problem 3.16 may be useful here. Symmetric matrices are useful in implementing undirected graph structures.

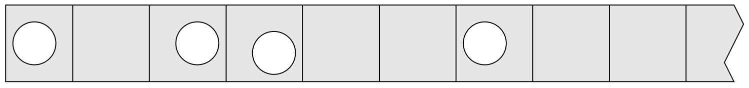
3.18 Sometimes it is useful to keep an unordered list of characters (with ASCII codes 0 through 127), with no duplicates. Java, for example, has a `CharSet` class in the `java.util` package. Implement a class, `CharSet`, using a `Vector`. Your class should support (1) the creation of an empty set, (2) the addition of a single character to the set, (3) the check for a character in the set, (4) the union of two sets, and (5) a test for set equality.

3.10 Laboratory: The Silver Dollar Game

Objective. To implement a simple game using `Vectors` or arrays.

Discussion. The Silver Dollar Game is played between two players. An arbitrarily long strip of paper is marked off into squares:

The game begins by placing silver dollars in a few of the squares. Each square holds at most one coin. Interesting games begin with some pairs of coins separated by one or more empty squares.



The goal is to move all the n coins to the leftmost n squares of the paper. This is accomplished by players alternately moving a single coin, constrained by the following rules:

1. Coins move only to the left.
2. No coin may pass another.
3. No square may hold more than one coin.

The last person to move is the winner.

Procedure. Write a program to facilitate playing the Silver Dollar Game. When the game starts, the computer has set up a random strip with 3 or more coins. Two players are then alternately presented with the current game state and are allowed to enter moves. If the coins are labeled 0 through $n-1$ from left to right, a move could be specified by a coin number and the number of squares to move the coin to the left. If the move is illegal, the player is repeatedly prompted to enter a revised move. Between turns the computer checks the board state to determine if the game has been won.

Here is one way to approach the problem:

1. Decide on an internal representation of the strip of coins. Does your representation store *all* the information necessary to play the game? Does your representation store *more* information than is necessary? Is it easy to test for a legal move? Is it easy to test for a win?
2. Develop a new class, `CoinStrip`, that keeps track of the state of the playing strip. There should be a constructor, which generates a random board. Another method, `toString`, returns a string representation of the coin strip. What other operations seem to be necessary? How are moves performed? How are rules enforced? How is a win detected?

3. Implement an application whose `main` method controls the play of a single game.

Thought Questions. Consider the following questions as you complete the lab:

Hint: When flipped, the Belgian Euro is heads 149 times out of 250.

1. How might one pick game sizes so that, say, one has a 50 percent chance of a game with three coins, a 25 percent chance of a game with four coins, a $12\frac{1}{2}$ percent chance of a game with five coins, and so on? Would your technique bias your choice of underlying data structure?
2. How might one generate games that are not immediate wins? Suppose you wanted to be guaranteed a game with the possibility of n moves?
3. Suppose the computer could occasionally provide good hints. What opportunities appear easy to recognize?
4. How might you write a method, `computerPlay`, where the computer plays to win?
5. A similar game, called Welter's Game (after C. P. Welter, who analyzed the game), allows the coins to pass each other. Would this modification of the rules change your implementation significantly?

Notes: