Computer Science 211
Data Structures
Mount Holyoke College
Fall 2009

Topic Notes: Vectors

Arrays are a very common method to store a collection of similar items.

Arrays work very well for a lot of situations, but they come with some very important restrictions.

- their size is specified on construction, and cannot be changed without constructing a new array and copying over the contents
- all array indices must be managed explicitly
- if you want to insert an item at the start of or in the middle of an array, you need to move one or more items out of the way to make room
- if you remove an item from the start or the middle of an array and you don't want to leave a "hole" in the middle, one or more items needs to be moved around to fill in the hole

This idea of a dynamically resizeable (or, *extensible*) array leads naturally to the idea of a *vector*.

The built-in Java class `java.util.Vector` allows its user to build something like an array, but it can change size dynamically.

We can add new elements or delete elements anywhere in the vector.

What kinds of operations would we like to have on something that behaves like a resizeable array?

We need the functionality of a regular array:

- construction
- add an item to the end
- insert an item in the middle
- retrieve value of an element
- remove an item

Most of the operators of a vector will assume that the elements are "packed" – that is:

- if we add an element, it will be added to the end by default
- if we add an element in the middle, all elements with higher subscripts are moved up to make room

- if we remove an element, all elements with higher subscripts are shifted down to fill in the space

So we already have some extra functionality that a regular array doesn't have.

Since the vector needs to be able to hold anything, its elements are of type `Object` (until we look at generics shortly), hence our initial implementation will use casts when items are retrieved.

Here are the key methods we will consider in the implementation of `Vector` in structure package (which mimics the one in `java.util`).

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

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

    // post: returns true iff Vector contains the value
    public boolean contains(Object elem)

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

    // post: returns index of element equal to object, or -1.
    //  Starts at 0.
    public int indexOf(Object elem)

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

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

    // post: vector is empty
    public void clear()

    // post: remove and return first element of vector equal to parameter
    //  Move later elts back to fill space.
    public Object remove(Object element)

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

```
    // pre: 0 <= index && index < size()
    // post: element value is changed to obj
    public void set(int index, Object obj)
}
```

Vectors are generally used any time size of an array must change dynamically.

For this initial `Vector` implementation, each element stored can by of any type. If we have a `Vector` called `myVect` and we wish to store the `String` value `"Hello"`, we can write

```
myVect.add("Hello");
```

This `String` is an instance of `Object`, so it matches the expected type for the `add` method.

However, when we retrieve an an element (e.g., `myVect.get(0)`), the return type is `Object`. To be able to treat this value as a `String` (or whatever class it is an instance of), we must *typecast* (or, simply, *cast*) it back to the original data type:

```
String val = (String)myVect.get(0);
```

Java will check for us to make sure the `Object` returned is actually a `String` and will throw an exception (which, for our purposes, means the program will crash).

We can simplify our `Spells` example by using a `Vector` to represent the spell list – see the program in `SpellsVector.java`.

See Example:
`/home/jteresco/shared/cs211/examples/Spells`

Let's consider another example that makes better use of a `Vector`:

See Example:
`/home/jteresco/shared/cs211/examples/PocketChange`

This is a "pocket change" container. It stores the collection of coins in your pocket by their integer values in cents, using a `Vector`. You can add and remove coins and get the total value of the money in the pocket.

This illustrates one of the restrictions on `Vector`s (and all other general-purpose classes): We cannot store base types in our `Vector` since base types are not `Object`s.

Luckily, there are builtin classes to "wrap them up" as `Object`s:

```
Integer seven = new Integer(7);
```

Others are `Boolean`, `Character`, `Double`, `Float`, `Long`, and `Number`.

We can retrieve the `int` equivalent of an `Integer` by calling `intValue`.

```
seven.intValue();
```

You can find the entire list of classes and associated methods in the `java.lang` package documentation.

Starting with JDK 1.5, the Java system will do the conversions between base types and their "wrapper" classes automatically as needed.

The term is *autoboxing*.

See `PocketChangeAutobox.java`

This addresses a repeated complaint among Java programmers that they were always packaging up values and using the `intValue()` and similar functions.

Vector Implementation

How can we implement a `Vector`? We can't look at or modify the Sun implementation in `java.util`, which is why we have the structure package.

Structure was developed at Williams College to go with our text and is now used by lots of people who use this text.

We will look at the implementation of `Vector` in structure.

See Structure Source:
`/home/jteresco/shared/cs211/src/structure/Vector.java.java`

A `Vector` uses an array for the internal storage of elements it contains. It could also use lists or whatever else it would like, but an array is a good choice.

The array-based `Vector` implementation has two essential fields:

```
protected Object elementData[];
protected int elementCount;
```

the array and the number of elements of array currently in use.

Note that there is an important distinction between the size of the array and the number of elements in use by the `Vector`.

We don't need to store the size of the array, since Java arrays come equipped with that information in the `.length` field.

When the Vector is about to exceed capacity, we copy its elements into a larger array. We need an efficient strategy for this, which we will discuss shortly.

Some other items of note in the implementation:

- There are several constructors, but we will focus on just three:

```
public Vector();
```

The parameterless constructor simply calls the single parameter constructor with a constant value of 10, so we will start with the single parameter constructor.

```
public Vector(int initialCapacity);
```

This constructor creates an empty `Vector` with an array allocated with `initialCapacity` entries.

```
public Vector(int initialCapacity, int capacityIncr);
```

This does the same, but also sets the instance variable `capacityIncrement` to the value specified. We will look at the use of this value soon.

- There are two `add` methods, one that adds an element at the end of the `Vector` and another that adds an element at a specific position.
  - both call `ensureCapacity` to make sure there is space for the new element (more soon)
  - the version that inserts at a location needs to move up any elements beyond the insertion point to make room (up to n copy operations for an n-element `Vector`!)
- The `remove` method returns the item at a given index and then shifts down the contents beyond that index to avoid a "hole" in the array. Again, we have up to n copy operations for an n-element `Vector`.
- The `get` and `set` methods are very straightforward. These retrieve or modify the entry at a given index in our `Vector`.
- A variety of other useful methods are less interesting (implementation-wise): `contains`, `indexOf`, `isEmpty`, `clear`, and `size`.

Managing the Internal Array Size

What if we run out of space in the array when adding new elements?

Arrays cannot be resized in place. In either case, we need to create a new, larger array then copy the contents from the old array to the new one.

This is an expensive operation: n copy operations for an n-element `Vector`.

But how much larger should we make the array?

Options:

1. Increase the array size by 1 (or some other constant value)
2. Double (or triple, ...) the array size

Consider the first option, starting with an empty `Vector` and an initial capacity of 1.

Over the course of n `add` operations, we will peform about $\frac{n^2}{2}$ copy operations:

$$0 + 1 + 2 + 3 + 4 + ... + n = n * \frac{n-1}{2}$$

With the second option (assuming n is power of 2 for simplicity), we have to copy

$$0 + 1 + 2 + 4 + 8 + ... + \frac{n}{2} = n - 1$$

elements.

Copying about n elements is much less painful than copying $\frac{n^2}{2}$.

Of course, no copies would need to be made if we just allocated space for n elements at beginning (a good idea, if you know n ahead of time, but if you did, you might just be using an array...).

Our `Vector`s let the user decide which strategy to use.

If the `Vector` is constructed with a `capacityIncrement` of 0 (either by using a constructor that does not specify one, or by passing 0 to that constructor parameter), the `Vector` will double its array's length each time it needs to expand.

If a non-zero `capacityIncrement` is specified, the `Vector` will be expanded by that (fixed) amount each time it needs to grow.

So it is up to the user to decide which strategy would be more beneficial, given the expected usage patterns of the `Vector`.