Computer Science 210
Data Structures
Siena College
Fall 2016

Topic Notes: The ArrayList/Vector

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*, now often called an *array list*.

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

And of course, we will want it to resize itself to have enough space for as many elements as we add.

We will look first at how such a structure as provided by the Java API can be used, then will consider how it works.

The Java `ArrayList` Class

As you continue to expand your programming skills, you will learn about a variety of ways that collections of data can be stored that vary in complexity, flexibility, and efficiency. The first of these structures that we will consider is the `ArrayList`.

`ArrayList` is a class that implements an *abstract data type (ADT)* provided by the standard Java utility library. We will look more closely later at what it means for the `ArrayList` to be an ADT.

Let's see how to use them through an example: we will enhance the "PurchaseTracker" example with an `ArrayList` that holds all of the `PurchasedItem` objects we create.

See Example: PurchaseTrackerAll

We consider each change that was made to the program to see the basic usage of an `ArrayList`.

- First, we need to add an `import` statement to the top of our program.

  ```
  import java.util.ArrayList;
  ```

  This allows us to use the class name `ArrayList` in the rest of the file and Java will know we mean to use the one in the `java.util` package.

- Next, we declare a local variable in `main` for our `ArrayList` and construct an instance:

  ```
      ArrayList<PurchasedItem> items = new ArrayList<PurchasedItem>();
  ```

  Since an `ArrayList` is a generic structure that can be used to hold objects of any type, we need to tell Java what type of objects will be stored in this particular `ArrayList`. In this case, it's `PurchasedItems`. So we specify this as a type parameter both in the variable declaration and the construction.

- The `PurchasedItem` instances are then created, and we need to insert each into the `ArrayList`. This is done with the `add` method:

  ```
          items.add(item);
  ```

  This will take the `PurchasedItem` named `item` and add it to the first available slot in the `ArrayList` named `items`.

Note that in this case, we are not requesting any specific location within the ArrayList for the item. We will later see that we can be more specific here.

Note also that we as users of the ArrayList do not know (though when you take data structures, you'll have a pretty good idea) of what's going on inside the ArrayList to add the item. We just know that it knows how to do it.

When we're done with the do..while loop, the ArrayList contains references to all of the PurchasedItem objects we constructed.

- In the rest of the main method, we need to access the PurchasedItem objects within the ArrayList. We do this with the get method:

  ```
  item = items.get(0);
  ```

  in the middle of the method gives us a reference to the first PurchasedItem that we had added to the ArrayList.

  We then see a for loop that uses itemNum is a loop index variable that will range from 1 to one less than the number of items in the ArrayList. How many items are there? We can get that information from the ArrayList itself using the size method.

  What we see here is that the ArrayList has assigned a number, often called an *index*, to each PurchasedItem we added to the ArrayList, and we can pass that number to the get method to get back a specific PurchasedItem from the ArrayList.

  This is our good example of a *search* operation on a collection – we are looking through each object in the collection to find ones that are the "winners" in each category. More precisely, this is a *linear search* and we will say more about this later.

One of the great things about using a construct like an ArrayList is that we can extend our programs to keep track of a much larger number of objects. No matter how many items we enter into the program (within the bounds of our computer's memory resources, at least) we can use a collection like an ArrayList to keep track of them.

For a second example, consider the use of an ArrayList of Association objects:

See Example: Spells

Again, we construct an ArrayList and add items to it. In this case, Associations which use String objects for both key and value.

There is just one ArrayList method here that was not in the previous: indexOf. This one searches through the ArrayList for an object that is equivalent (by the equals method) to the one passed as a parameter. It returns the index (position within the ArrayList) of the first match. If no match exists, it returns -1.

Note here that we make use of the fact that for two Associations to be considered equal, their keys must match, but their values do not.

Contrast this with the same program using primitive arrays instead:

See Example: SpellsArray

- Our variable declaration looks a bit different.
- When we construct the array in the `main` method, we need to tell it how many elements the array will hold (in this case, 10). With the `ArrayList`, we construct a list and we can add as many things to it as we want. The array can only ever hold the number of elements we provided when we constructed it.
- When we add items to the array, we need to specify the index explicitly. There is no way to say "just add it to the end" the way we do with `ArrayList`s.
- When we access array elements, we use the bracket notation in much the same way we use the `get` method of the `ArrayList`.
- The array remembers how many entries it contains, and we can access this information with the `.length`. This plays the role of the `size` method of the `ArrayList`.

Other `ArrayList` methods

The examples above demonstrated just a few of the capabilities of the `ArrayList` class: construction, `add`, `get`, and `size`.

The full documentation for the `ArrayList` can be found on Oracle's Java documentation site:

Java API Documentation: ArrayList at
`http://docs.oracle.com/javase/8/docs/api/util/ArrayList.html`

Here are a couple of additional methods, some of which will come up in later examples.

- `remove` – remove an object from the list
- `clear` – remove all objects from the list
- `contains` – determine if a given object is in the list
- `set` – replace the contents at an index with a new element

`ArrayList`s of Primitive Types

Java places a significant restriction on the use of primitive types as the type parameters for generic data structures such as the `ArrayList`. The following would not be valid Java:

```
ArrayList<int> a = new ArrayList<int>();
```

The type in the <> must be an object type. Fortunately, Java provides object types that correspond to each primitive type. An `Integer` object is able to store a single `int` value, a `Double` value is able to store a single `double` value, etc. So the declaration and construction above would need to be:

```
ArrayList<Integer> a = new ArrayList<Integer>();
```

In older versions of Java, programmers would need to be careful to convert back and forth between values of the primitive types and their object encapsulators. To construct an `Integer` from an `int i`, one would need to do so explicitly:

```
a.add(new Integer(i));
```

And to retrieve the `int` value from an `Integer`, one would also do so explicitly:

```
a.get(pos).intValue();
```

However, recent versions of Java automatically convert between the primitive types and their object encapsulating classes. This is called *autoboxing* when converting from primitive to "boxed" encapsulating classes, and *autounboxing* when going back the other way.

However, the effective programmer should always keep in mind that these conversions are occurring, as there is a computational cost to each.

Another Example

Suppose we have an `ArrayList` of Integer values, and someone (by a mechanism which is not our concern) has asked us to write a method that will find the largest value in the `ArrayList`. The following method will achieve this (we assume at least one element in the `ArrayList`):

```
private static int findMax(ArrayList<Integer> a) {

  int max = a.get(0);
  for (int i=1; i<a.size(); i++) {
    int val = a.get(i);
    if (val > max) max = val;
  }
  return max;
}
```

The Enhanced `for` Loop

We have seen that a common task with a collection such as an `ArrayList` is to *iterate* over its contents. That is, "visit" every element in the list exactly once to do something to it.

It is often the case (and was in many of the examples here) that the specific index of an entry in an `ArrayList` is not important as we are iterating over its contents.

In these cases, the counting `for` loops can be replaced with a related Java construct often called the *enhanced* `for` *loop*, or a "foreach" loop.

If we have an `ArrayList` of objects of some type `T` and we wish to loop over all entries in the loop, we can replace a counting loop:

```
ArrayList<T> a = new ArrayList<T>();

...

for (int i = 0; i < a.size(); i++) {
  T item = a.get(i);
  // do something with item
}
```

with an enhanced `for` loop:

```
ArrayList<T> a = new ArrayList<T>();

...

for (T item : a) {
  // do something with item
}
```

This construct will loop enough times so that the variable `item` will be assigned to each entry in `a` exactly once through the body of the loop.

The enhanced `for` construct is not always appropriate, however. For example, in the `findMax` method above, it is more convenient to be able to get the item at position 0 as the initial "max" and then loop over the entries from positions 1 and up to check for larger values.

As you learn more Java, you will see a number of other data structures that can be used with the for-each loop construct.

An `ArrayList` Within a Custom Class

It may or may not have become clear so far that you can use `ArrayLists` in pretty much any context that you can use other data types. This includes as an instance variable in a custom class.

See Example: CourseGrades

In the above example, which you will expand as part of your next lab, `ArrayLists` are used to keep track of a list of students and course grades, and within the class that represents one student's information, the list of the grades.

Implementing and Analyzing the `Vector` class

The structure package includes implementations of many of the data structures we will consider. The structure implementation that behaves like Java's `ArrayList` is called a `Vector`. (Note: Java's API also provides a `Vector` that is nearly identical in functionality to the `ArrayList`.)

Since the `Vector` needs to be able to hold anything, its elements are of type `Object` (until we look at the generic version 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`:

```
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)
}
```

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).

Here is our `Spells` using an `Object`-based `Vector` to represent the spell list.

See Example: SpellsVector

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

See Example: 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 `Vectors` (and all other general-purpose classes): We cannot store base types in our `Vector` since base types are not `Objects`.

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

```
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.

We have already seen that Java will do the conversions between base types and their "wrapper" classes automatically as needed with autoboxing and autounboxing.

The term is *autoboxing*.

See Example: PocketChange file `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 Oracle (formerly Sun Microsystems) implementation in `java.util`, which is why we have the structure package.

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

See Structure Source:
`structure/Vector.java`

A `Vector` uses an array for the internal storage of elements it contains. Other internal data structures could be used, but a `Vector` is built upon an array.

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 default 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...).

`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`.

The Java API's `ArrayList` does not allow the specification of a capacity increment. The actual strategy to manage any needed growth is not specified beyond that it allows elements to be added in "amortized constant time."

Adding Generics

The items stored within our `Vector`s are treated as instances of `class Object`. As we saw with the `Association` implementation, we can also create a generic `Vector`.

The `Object`-based implementation is quite convenient in that we can store whatever type of objects we wish (and until version 1.4 of the Java Development Kit, this was the preferred approach).

This approach has a few disadvantages:

1. When we retrieve an item from our `Vector`, we need to use a cast before we can treat it as an instance of its own specific type.

2. If we make a programming error and mistakenly place an object of one type into the `Vector` but then cast it to a different type upon retrieval, your program will crash with a *run time error*. Ideally, we would be able to detect such errors sooner – at *compile time*.

One approach to dealing with these disadvantages is to implement a *specialized* version of our `Vector` (or whatever other) data structure that holds exactly the type of items we wish, much like we can declare arrays of any type.

To implement, for example, a `Vector` that holds `Integers` (we could call it `class IntegerVector`), we could take the `Vector` implementation, and instead of using `Objects` as the type for our internal array and for the method parameter and return types, we would use `Integer`.

This would take care of both disadvantages we noted in the original `Vector` implementation. Casts are no longer needed because the return type of methods such as `get` would be `Integer`. And perhaps more importantly, if we attempted to write code that stored anything other than an `Integer` (or a subclass of `Integer`), the Java compiler would flag the error (a *compile-time error*), which is much more convenient time to detect an error than at run time

But unfortunately, this "solution" means writing a brand new specialized `Vector`-like class for each data type we need to store.

Starting with JDK 1.5 (Java 5), Java was extended to allow class definitions to include *generic*, or *parameterized data types*. This means that we can write a definition of the structure using data types that are unspecified (much like the value of a method parameter is unspecified) until we create an instance of the class.

See Example: PocketChange file `PocketChangeT.java` for an example of the usage of the generic `Vector`.

As we have seen, we specify the data type of the items we will be storing in the generic data structure in angle brackets after the structure type. For example, the `Vector` of `Integer`:

```
Vector<Integer> intVec = new Vector<Integer>();
```

With this declaration, any attempt to store an item which is not of type `Integer` or any treatment of an item retrieved as a non-`Integer` type will result in a compile-time error.

The generic data types, including `Vector` and `Association` are provided in the `bailey.jar` library, but you will need to `import structure5.*;` instead of `import structure.*;` at the top of your program.

Note: we would like to be able to use a primitive type as a type parameter:

```
Vector<int> intVec = new Vector<int>();
```

but this is not permitted – the type parameters must be object types. Fortunately, with autoboxing, this is not much of an inconvenience to programmers.

From here on, we will make use of the generic classes.

The generic `Vector` class is parameterized on the type of the items (elements) it will contain. The implementation uses `E` as the type.

See Structure Source:
`structure5/Vector.java`

For the most part, the `Vector` implementation is straightforward. However, a technical problem comes into play when we declare the `Vector`'s internal array. This is not something we will concern ourselves with at this point, but the description of the problem and of its solution within structure is worth reading in Bailey.