JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaqa/2001-06/03-qa-0622-vector.html

Vector or ArrayList -- which is better?

Find out the difference between Vector and ArrayList

By Tony Sintes, JavaWorld.com, 06/22/01

Q: `Vector` or `ArrayList` -- which is better and why?

A: Sometimes `Vector` is better; sometimes `ArrayList` is better; sometimes you don't want to use either. I hope you weren't looking for an easy answer because the answer depends upon what you are doing. There are four factors to consider:

- API
- Synchronization
- Data growth
- Usage patterns

Let's explore each in turn.

API

In *The Java Programming Language* (Addison-Wesley, June 2000) Ken Arnold, James Gosling, and David Holmes describe the `Vector` as an analog to the `ArrayList`. So, from an API perspective, the two classes are very similar. However, there are still some major differences between the two classes.

Synchronization

`Vectors` are synchronized. Any method that touches the `Vector`'s contents is thread safe. `ArrayList`, on the other hand, is unsynchronized, making them, therefore, not thread safe. With that difference in mind, using synchronization will incur a performance hit. So if you don't need a thread-safe collection, use the `ArrayList`. Why pay the price of synchronization unnecessarily?

Data growth

Internally, both the `ArrayList` and `Vector` hold onto their contents using an `Array`. You need to keep this fact in mind while using either in your programs. When you insert an element into an `ArrayList` or a `Vector`, the object will need to expand its internal array if it runs out of room. A `Vector` defaults to doubling the size of its array, while the `ArrayList` increases its array size by 50 percent. Depending on how you use these classes, you could end up taking a large performance hit while adding new elements. It's always best to set the object's initial capacity to the largest capacity that your program will need. By carefully setting the capacity, you can avoid paying the penalty needed to resize the internal array later. If you don't know how much data you'll have, but you do know the rate at which it grows, `Vector` does possess a slight advantage since you can set the increment value.

Usage patterns

Both the `ArrayList` and `Vector` are good for retrieving elements from a specific position in the container or for adding and removing elements from the end of the container. All of these operations can be performed in constant time -- *O(1)*. However, adding and removing elements from any other position proves more expensive -- linear to be exact: *O(n-i),* where *n* is the number of elements and *i* is the index of the element added or removed. These operations are more expensive because you have to shift all elements at index *i* and higher over by one element. So what does this all mean?

It means that if you want to index elements or add and remove elements at the end of the array, use either a `Vector` or an `ArrayList`. If you want to do anything else to the contents, go find yourself another container class. For example, the `LinkedList` can add or remove an element at any position in constant time -- *O(1)*. However, indexing an element is a bit slower -- *O(i)* where *i* is the index of the element. Traversing an `ArrayList` is also easier since you can simply use an index instead of having to create an iterator. The `LinkedList` also creates an internal object for each element inserted. So you have to be aware of the extra garbage being created.

Finally, in "PRAXIS 41" from *Practical Java* (Addison-Wesley, Feb. 2000) Peter Haggar suggests that you use a plain old array in place of either `Vector` or `ArrayList` -- especially for performance-critical code. By using an array you can avoid synchronization, extra method calls, and suboptimal resizing. You just pay the cost of extra development time.

Read more about Core Java in JavaWorld's Core Java section.