1. A Java array is an indexed data structure.

T

1. Classes which implement the Java List interface allow the traversal of the structure without having to manage a subscript.

T

1. ArrayList objects have a capacity.

F (well, in the sense that the capacity is doubled when needed. )

Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

1. Subscripts can be used with indexed collections in Java.

~~T~~

1. New applications should use the Vector class rather than the ArrayList class since its operations will generally be a little more efficient.

F

1. A data structure operation which executes in O(n) time will always be faster than an operation that executes in O(n log n) time.

F (strictly speaking, not “always”. For some very small values of n, O(nlogn) will be faster. However, at those small values it’s sort of a moot conversation - in most situations where the n is that small, it doesn’t matter which algorithm you use. Perhaps it would make a difference if one needed to calculate some extremely intensive thing several times ??)

1. A data structure operation which executes in O(log n) time will always be faster than an operation that executes in O(1) time.

F

1. Big-oh notation allows data structure performance to be compared across various platforms and aids in algorithm design.

T

1. Nested loops generally result in efficient O(m \* n) performance where m is the level of nesting.

F

1. A Singly-linked List implementation will in general use less memory than the equivalent Doubly-linked List.

T

1. Adding to the front end of a Singly-linked List generally executes in O(1) time.

F

1. Adding to the back end of a Doubly-linked List generally executes in O(1) time.

~~F~~

1. An Iterator can be used to ensure that all the data elements of a List are “visited”.

T (among other things..)

1. A ListIterator is more flexible than an Iterator when used with List structures.

T

1. To use a data structure with the Java for-each construct, the data structure must implement the Iterator interface.

T

1. Collections hold references to objects.

T ~~(although I’m pretty sure they can also hold primitives which are values as opposed to references~~)

1. Data structures which implement the RandomAccess interface must support a get method that executes in O(1) time.

F

1. A Stack is a FIFO data structure.

F (pretty sure a stack is LIFO)

1. A Queue is a FIFO data structure.

T

1. A List is a FIFO data structure.

F

1. A Queue can be implemented efficiently using a Singly-linked List.

T (?)

1. A Stack can be implemented efficiently using a Java array.

T (?, well, maybe except for the capacity part…)

1. A Stack can be implemented efficiently using a Doubly-linked List.

F (? You probably don’t need back links in a stack…)

1. The methods; push, pop and peek are associated with a Stack.

T

1. A ConcurrentModificationException is likely to be thrown when an item is offered (enqueued) and then immediately polled (dequeued) from an empty Queue.

F (my understanding is that this exception would be thrown if one tried to modify an list item while iterating over the list, like list.remove() – which is why we have the lovely ListIterator)