Computer Programming II Instructor: Greg Shaw

# COP 3337

***The ArrayList Class (Review)***

1. **Concepts and Terms**

***data structure***: a bunch of related memory locations for storing related values

***array***: a data structure consisting of consecutive memory locations for storing related values all of the same type. I.e., a *list* of related values

***ArrayList***: a Java class that implements a powerful, easy to-use, abstract list. Although the underlying implementation of the ArrayList is an array, this is hidden from the user

***index***: an integer expression that tells you the *position* of an object on a list

1. **When Do We Need a List?**

"*When we need to keep many related values in memory at the same time*"

Which of the following operations requires an ArrayList or an array? Why or why not?

1. Find the average of any number of test scores
2. Find the average of any number of test scores and the deviation of each score from the average
3. **Advantages of ArrayLists (and Arrays)**
4. Treats many related variables as a single entity (i.e., a *list*) with a single name
5. Allows us to process a list of any size (i.e., any number of objects on the list) in the same amount of code, using a loop
6. **ArrayList Processing - the Big Ideas**
7. The two stages of list processing
8. *populating* the list - creating objects and adding them to the list)

1. *processing* the list – “traversing” the list (i.e. “visiting” each object on the list one at a time), “getting” each object, and doing something with it
2. When you create an ArrayList object it is *empty*. When you add new objects to the list, it automatically expands to accommodate them. When you delete objects from a list, it contracts. In other words, ArrayLists are *variable-sized* (as opposed to *arrays*, which are *fixed-size*)
3. **The ArrayList class has a method called *size()* that returns the number of objects currently stored in the list (i.e., the *size* of the list).**
4. *We process a list one object at a time*, in a loop. The *size() method* tells us the number of objects on the list. We do one iteration for each object on the list and we generally use a *for* loop since the number of iterations is known in advance (courtesy of *size()*)
5. The *index* (i.e. "position") of the first object on a list is 0, not 1.
6. **Using a *for* Loop to "Traverse" an ArrayList**

The loop control variable is used as the *index* into the list (i.e., it tells you *which* object is being accessed). As the value of the *lcv* changes, we "visit" each object in turn. This is commonly known as "traversing" a list.

Here is a loop that traverses an ArrayList called *myList* and accesses each object on it:

**for** (**int** i = 0 ; i < myList.size() ; i++)

{

*variable* = myList.get(i) ;

// do something here with *variable* (see 3., below)

}

Note:

1. Method *size()* returns the number of objects on the list. Since the first object has index of 0, the lcv must go from 0 to myList.size()-1, *not* from 1 to myList.size().
2. ArrayList method *get* is used to return the object at a specified index. Note how the lcv, *i*, is used as the argument to *get*.
3. *variable* must be an object variable of the class of objects stored on the list. It may also be a *primitive* (int, double, etc) for lists of primitives (see **VI.**, below)
4. **Declaring "Generic" ArrayLists**

* A "generic" ArrayList is an ArrayList tailored to store objects of a specific class
* Syntax of the Generic ArrayList Declaration

ArrayList<*class*> *identifier* = **new** ArrayList<*class*>() ;

* *class* is the class of objects to be stored on the list (this may be any standard or programmer-defined class)
* *identifier* is the ArrayList object variable name
* Examples

ArrayList<String> names = new ArrayList<String>() ;

ArrayList<BankAccount> accounts =

new ArrayList< BankAccount>() ;

ArrayList<Integer> scores = new ArrayList<Integer>() ;

* Just as with any other class, we may declare the object variable and create the object in separate statements:

ArrayList<Boolean> seatAvailable ;

.

.

seatAvailable = new ArrayList<Boolean>() ;

1. **Major ArrayList Methods**

(Assume that *list* is an ArrayList object variable)

* **list.size()**

returns the number of objects in ArrayList *list*

* **list.get( *i* )**

returns a pointer to the object at index *i* (i.e., at position "i" on the list, where the first object is at index 0)

* **list.set( *i*, *object* ) ;**

stores *object* at index *i* of a list, *replacing* the object that was previously there. The size of the list does not increase

* **list.add(*object*) ;**

appends *object* to end of the list, increasing size by 1

* **list.add( *i*, *object*) ;**

adds *object* to a list at index i. The size of the list increases by 1. The object previously at index i and all other objects at higher indices are "moved down" one position to make room

* **list.remove( *i* )**

deletes the object at index *i* from the list and returns a pointer to it

The size of the list decreases by 1, and all objects from index *i+1* to the end of the list are "moved up" one position to fill the gap

E.g., if *i* were 3, then the object at index 3 (the 4th object on the list) is removed and the object formerly at index 4 (the former 5th object) is moved up to index 3, etc.

* **list.remove( *obj* )**

searches the list for an object equal to obj. If found, that object is removed and **true** is returned; otherwise returns **false**

* Although both *remove()* methods return values, they are sometimes called as if they were **void** methods. This may be done when we just want to remove the object from the list and are not interested in the value returned

1. **"Overflowing the Bounds" of an ArrayList - a Very Common Error**

Recall that the index of the first object on a list is always 0. Also recall how the *size()* method returns the number of objects on an ArrayList.

This means that "legal" indices into an ArrayList are 0 through *list.size() - 1*. If you use a value *smaller* than 0 or *larger* than *list*.size() - 1 as the index, then you are attempting to access an object that does not exist, and an **IndexOutOfBoundsException** will be thrown.

A common cause of this exception is a loop that generates one "extra" list index expression. E.g,

**for** (**int** index = 0 ; index <= myList.size() ; index++)

instead of

**for** (**int** index = 0 ; index < myList.size() ; index++)

1. **Import Statement**

The ArrayList class is in Java's "utilities" package, so use

**import** java.util.ArrayList ;

1. **ArrayLists and Primitive Types**

To create a list of a given primitive type, we just use the “wrapper” class for that type. (See the document“*ArrayLists and Primitive Types*”)