ArrayLists (343-354)

What is an ArrayList:

An ArrayList can be thought of as a cross between a simple one dimensional array and an abstract data type called a **List**. A List is an abstract data type that represents a collection of values that are all of the same type. It is a type of data structure that is a container for other values. Abstractly a List must provide:

- Storage for a set of values. The number of values in the list can increase or decrease as the USER of the list adds and removes values.
- The values stored in the list can be either in ordered (sorted) or unordered
- A mechanism to insert (add) values at a specific position in the list
- A mechanism to delete (remove) values at a specific position in the list
- A mechanism to test if the list is empty or full, and to retrieve the current size of the list
- Possibly a mechanism for sorting the values within the list
- Mechanism for searching for a value contained within the list

There are multiple ways that lists can be implemented, one ways is through the use of an **array** to store the values that the list contains. In class we learned how to do all of the previous operations on a simple array. However many of these operations involved shifting elements around when adding and removing elements.

Class ArrayList wraps the concept of a list around an array and provides a set of operations that allows the user to conveniently manipulate the contents of the ArrayList. NOTE: **Like arrays the first element in an ArrayList is at index 0.**

ArrayLists provides a significant advantage over simple arrays.

- Once an array is declared you cannot change its size. Class **ArrayList** allows us to create arrays that can grow and shrink automatically as needed. This is extremely valuable when we do not know how many values our program needs to process.
- It manipulates the contents of an array, providing us with methods that allow us to add, delete, change and reorder elements without having to directly manipulate elements.
- Once the **ArrayList** is full (all of the elements initially allocated contain a value), adding any additional values will cause the capacity of the **ArrayList** to be automatically increased. A new container **array** allocated and the values copied. AS YOU MIGHT GUESS THIS IS EXPENSIVE. It is better to allocate an **ArrayList** with an initial capacity if you have a reasonable guess as to the maximum number of values.
- As values are added and removed the “size” of the ArrayList grows and shrinks. This is an indication of the “currentSize” of the ArrayList.. or the number of values it actually contains. Let’s write some simple code. ArrayList manages a partially filled array for us

How do you create an ArrayList

Class **ArrayList** is a **generic** class. This means it is written to work with any type of data, as long as the data items are instances of a class. It is part of the java.util package; to create an **ArrayList** we declare it as follows:

ArrayList<OBJECTTYPE> variableName = new ArrayList<OBJECTTYPE>();

When we declare/instantiate and ArrayList object, we can state an initial capacity or allow it to default to 10.

```
// This creates array list of with a maximum size of  10 elements of type String

ArrayList<String>  studentList = new ArrayList<String>();
```

OR

ArrayList<OBJECTTYPE> variableName = new ArrayList<OBJECTTYPE>(int initial Capacity);

```
// this creates an array list of the size given by initialCapacity

ArrayList< Integer >  set1  = new ArrayList< Integer >( 50);
```

NOTE: The type of data stored in to an ArrayList must be of a **reference** type. So we can not use our primitive types such as **int, float, double, char**. Instead we must use their "wrapper class" counterparts defined in the Java API: **Integer, Float, Double, Character etc.** These are wrapper classes, that are fully described on page 351 of your text. Java automatically will convert a value of a primitive type to an object of its associated wrapper class and back again through assignment with no additional effort on the part of the user. This is called **auto-boxing and auto-unboxing.**

Example: of how autoboxing and auto-unboxing works

```
import java.util.*;

public class testBoxing{

public static void main(String[] args){
// declare two integer objects with an initial value
  Integer number1 = new Integer(0);
  Integer number2 = new Integer(44);
  int x = 55;

  System.out.println("x is " + x + " and number1 is " + number1 +
         " and number2 is " + number2);

 number1 = x;

 System.out.println("x is " + x + " and number1 is " + number1 +
         " and number2 is " + number2);


   x = number2;

   System.out.println("x is " + x + " and number1 is " + number1 +
         " and number2 is " + number2);

}
}
```

This outputs:

```
x is 55 and number1 is 0 and number 2 is 44
x is 55 and number1 is 55 and number 2 is 44
x is 44 and number1 is 55 and number 2 is 44
```

Some of the methods available for class ArrayList are

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
void	**trimToSize**() Trims the capacity of this ArrayList instance to be the list's current size.

See page 246 for samples of using methods on ArrayLists

Example 1: Creating and initializing an ArrayList of integers.

```
import java.util.*;

public class arrList{

  public static void main(String[] args){
   ArrayList<Integer> set = new ArrayList<Integer>();

   System.out.println("At the beginning set contains " + set.size() + " values.");


 for (int i=0; i< 15; i++)
             set.add(  i*i );


// write code to print the current size of the list and all of the values from beginning
// to end
    System.out.println("Now set contains " + set.size() + " values. They are :");


for( int j=0;  j< set.size(); j++)
      System.out.print( set.get(j)+" " );


 System.out.println();  // blank line

    System.out.println("The value 36 is located at position " + set.indexOf( 36 ));
    System.out.println("The value 19 is located at position " + set.indexOf( 19 ));


  }
}
```

Choosing between an array and an ArrayList

1. If you know the size of the set of data being manipulated, and that size will never change no matter how many times the program is executed. Use an array.
2. If you are manipulating a large set of primitive values and array is more efficient.
3. Use an ArrayList for flexibly manipulating sets of values whose size is unknown or may change each time the program is executed.