Lesson 43….. *ArrayList*

You will recall from Lesson 42 the *ArrayList* is one of several classes that implement the *List* interface. As its name suggests, *ArrayList* also involves arrays. Basically, everything we learned in Lessons 18 and 19 concerning arrays can also be applied to *ArrayList* objects, however, with slightly different methods.

Comparing *ArrayList* to ordinary arrays:

So, a legitimate question to ask at this point is, "Why clutter our brains with a new set of commands for the *ArrayList* if it serves the same purpose as do ordinary arrays?" We are going to discuss the advantages of the *ArrayList* class over ordinary arrays and, to be fair, its disadvantages.

Advantages… Ordinary arrays are fixed in size. When we create an array, we anticipate the largest possible size it will ever need to be, and when instantiating the array, dimension it to that size. We call this the physical size of the array and it always remains that size even though at some point in your program you may wish to only use a portion of the array. The size of that portion is called the logical size. Your own code must keep up with that size. By contrast, the *ArrayList* expands and contracts to meet your needs. If you remove items from or add items to your *ArrayList*, the physical and logical sizes are always identical. This could be very important if you wish to be conservative of memory usage. With memory being so abundant and inexpensive today, this is no longer the advantage it once was.

One of the *add* methods allows very easy insertions of new items in the interior of the list without the nuisance of having to pre-move preexisting items.

A final advantage is that iterator objects are provided, whereby we can easily traverse the list. See Lesson 44 for an explanation of iterators.

Disadvantages… *ArrayList* can only store objects. If we wish to store primitives such as integers, *double*s, or *boolean*s, they must be converted into their wrapper class counterparts (see Lesson 21). This was once a nuisance because they had to be converted manually, but is now circumvented with the advent of Java 5.0+ and its autoboxing feature. Similarly, when we retrieve things from an *ArrayList*, they come out as objects. "Big deal", you say….certainly, if we store objects in the list, then we expect to get objects back when we retrieve from the list. Yes, but it's worse than one might think. When retrieving an object from the list, it doesn't come back as the same type object that was originally stored. Rather, it comes back as an *Object* type object (recall the cosmic superclass from Lesson 36). It will be necessary to cast it down to its original object type…yet another nuisance (partially circumvented by Java 5.0+ if type parameters are used as discussed below).

So, what are the methods we use with *ArrayList*? Look back at Lesson 42 on the *List* interface. Since ***ArrayList* implements the *List* interface**, those are the methods. We will now offer sample usage and/or discussion of several of the more important methods.

In each of the following examples we are to assume an *ArrayList* object has been created via

```
ArrayList aryLst = new ArrayList( );     or       List aryLst = new ArrayList( );
```

Type parameters:

With the addition of type parameters to Java 5.0+, it is also possible to create an *ArrayList* object as follows (See Appendix AF for the related topic of generics.):

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

The <*String*> part indicates that objects we add to the list can only be *String* types. (Instead of *String* we could use any object type.) This insures "type safety" and would result in a compile time error if we ever tried to add some other type object to *aryLst*. Type parameters also remove the burden of casting *Object* type objects retrieved from a list back to their original type. Unfortunately objects retrieved from an *ArrayList* using an iterator must still be cast **unless** the iterator also uses generics (see page 44-3).

In the following examples, assume that only *Integer* type objects have been stored in the list and that *aryLst* was created with *List<Integer>aryLst = new ArrayList<Integer>( );*

void add(Object o) //**signature**

Example:

```
aryLst.add(13);
        //pre Java 5.0
        Integer jw = new Integer(j);
        aryLst.add(jw);  // add jw to the end of the list
```

void add(int index, Object o) //**signature**

Example:

```
aryLst.add(3, 13);
        //pre Java 5.0
        Integer jw = new Integer(j);
        aryLst.add(3, jw);  //inserts jw at index 3 after moving the existing object at index
                            //3 and greater, up one notch.
```

Object get(int index) //**signature**

Example:

```
int q = aryLst.get(3);
        //pre Java 5.0
        Object obj = aryLst.get(3);  //retrieve object at position 3
        Integer qw = (Integer)obj;  //cast down from Object to Integer
        int q = qw.intValue( );  //convert back to int type.
```

Object remove(int index) //**signature**

Example:

```
int q = aryLst.remove(3);
        //pre Java 5.0
        Object obj = aryLst.remove(3); // removes object at position 3 (then compacts the list)
        Integer qw = (Integer)obj;  //cast down from Object to Integer
        int q = qw.intValue( );  //convert back to int type.
```

Object removeLast( ) //**signature**

Example:

```
int q = aryLst.removeLast( );
        // pre Java 5.0
        Object obj = aryLst.removeLast( );  // removes object at end of list and returns that object
        Integer qw = (Integer)obj;  //cast down from Object to Integer
        int q = qw.intValue( );  //convert back to int type
        .
```

Object set(int index, Object o) //**signature**

Example:

```
int q = aryLst.set(3, 13);
        //pre Java 5.0
        Integer jw = new Integer(13);
        Object obj = aryLst.set(3, jw);  // replaces object at position 3 with jw and returns original object
        Integer qw = (Integer)obj;  //cast down from Object to Integer
        int q = qw.intValue( );  //convert back to int type.
```

boolean isEmpty( ) //**signature**

Example:

```
aryLst.isEmpty( );  // returns true if there are no objects in the list
```

int size( ) //**signature**

Example:

```
aryLst.size( );  //returns the number of objects in the list
```

void clear( ) //**signature**

Example:

```
aryLst.clear( );  //removes all objects from the list
```

With Java 5.0+, autoboxing makes the following three methods (signatures are shown) easy to use. For example, if we are seeking the integer 13, the argument sent to the method would simply be 13.

```
int indexOf(Object o)
int lastIndexOf(Object o)
boolean contains(Object o)
```

Constructors:

```
ArrayList( )       //Default constructor
ArrayList(Collection c) //Constructs a list with the elements of the specified collection.
ArrayList(int j)  //For fast storage, preallocates space for j elements; however, more
                  //than j can be stored.
```

Big O values:

Determine the efficiency of algorithms using *ArrayList* methods with the following table:

List Methods Used With ArrayList	**Big O values**
add(int index, Object o)	O(n)
add(Object o)	O(1)
contains(Object o)	O(n)
get(int index)	O(1)
indexOf(Object o)	O(n)
remove(int index)	O(n)
clear( )	O(1)
set(int index, Object o)	O(1)
size( )	O(1)

Table43-1

Two very important methods, *iterator( )* and *listIterator( )* will be discussed in Lesson 44.

Exercise on Lesson 43

1. Write code that will instantiate an *ArrayList* object called *alst* and have the restriction that only *String* objects can be stored in it.

2. *ArrayList*s are restricted in that only ____________ can be stored in them.

3. What is the main advantage in using an *ArrayList* object as opposed to an ordinary array?

4. What is an advantage of using an *ArrayList* object that was created with type parameters?

In problems 5 - 9 an operation is performed with the "ordinary" array *ary*. Write equivalent code that performs the same operation on the *ArrayList* object called *a*. Assume that Java 5.0+ is being used and give two answers for each problem (parts A and B). For A part assume that *a* was created with *List a = new ArrayList( );* and for B part assume *List<Integer>a = new ArrayList<Integer>( );* was used:

5.
```
int x = 19;
ary[5] = x;
```

6.
```
int gh = ary[22];
```

7.
```
int sz = ary.length;
```

8.
```
int kd = ary[101];
ary[101] = 17;
```
   Use the set method:

9.
```
//Before inserting a new number, 127, at position 59, it will be necessary to move all
//up one notch. Assume that the logical size of our array is logicalSize.
for(int j = logicalSize; j >=59, j--)
{
        ary[j+1] = ary[j];
}
ary[59] = 127; //insert the new number, 127, at index 59.
```

   What code using *List* method(s) does the equivalent of the above code?

10. What does the following code accomplish? (*alist* is an *ArrayList* object)
```
while(!alist.isEmpty( ) )
{
        alist.removeLast( );
}
```

11. What one line of code will accomplish the same thing as does the code in #10 above?

12. Write a single line of code that will retrieve the *String* object stored at index 99 of the *ArrayList* object *buster* and then store it in a *String* called *myString*.

13. What type variable is always returned when retrieving items from an *ArrayList* object?

ArrayList… Contest Type Problems

1. What replaces **<*1>** in the code to the right to throw an appropriate exception when it violates the precondition?

   A. throws RunTimeException;
   B. throw new RunTimeException( );
   C. throw new NumberFormatException( );
   D. throws NumberFormatException( );
   E. throw new NullPointerException( );

2. What replaces **<*2>** in the code to the right to access the individual weight of the player in the *plyrs* list with index *i*?

   A. ((Player) plyrs.get(i).weight( ))
   B. ((Player)plyrs.get(i)).weight( )
   C. (Player) (plyrs).get(i).weight( )
   D. plyrs.get(i).weight( )
   E. More than one of these

3. The *class PlayerInfo* is an implementation of the interface *Player*, and the *PlayerInfo* constructor receives no parameters. Which of the following are valid declarations/instantiations?

   A. Player p = new PlayerInfo( );
   B. PlayerInfo pi = new Player( );
   C. Player p = new Player;
   D. PlayerInfo pi = new PlayerInfo;
   E. More than one of these

```
public interface Player
{
   public double height( );
   public int weight( );
}

public class Team
{
    public Team( )
    {
         plyrs =  new ArrayList( );
    }

    //precondition: p != null
    public Computer addPlayer(Player p)
    {
         if (p = = null) <*1>
         else  {
              plyrs.add(p);
              return this;
         }
    }
         … more methods

    public double weight( ) //weight of entire team
    {
         int sum = 0;
         for (int k=0; k<=plyrs.size( ); ++k)
              sum += <*2>;
         return sum;
    }

    private ArrayList plyrs;
}
```

4. What could be the type of *kbal* to insure that the *add* method is used correctly?

   A. int
   B. String
   C. Both A and B
   D. Both A and B, but first cast as *Object* types
   E. None of these

```
ArrayList al = new ArrayList( );
al.add(kbal);
```

5. What would replace **<*1>** in the code to the right so that the *Integer* stored at index 3 of the list be stored in the primitive integer *j*?

   A. Integer j = (Integer) aList.get(3);
   B. Integer j = aList.get(3);
   C. int j= (Object)aList.get(3).intValue( );
   D. int j= (Integer)aList.get(3);
   E. None of these

```
ArrayList aList = new ArrayList( );
//add some integers to the list
<*1>
```

6. Which of the following is an appropriate way to create an *ArrayList* object to which we could immediately begin adding *Cabinet* type objects?

A. ArrayList obj = new ArrayList(Cabinet);
B. ArrayList obj;
C. ArrayList(Cabinet) = new ArrayList( );
D. ArrayList obj = new ArrayList( );
E. None of these

7. If *ArrayList objAL* contains objects of type *Cabinet*, which of the following will cause *Cabinet cab* to be set equal to the object at index 8 of *objAL*?

A. cab = (Cabinet)(objAL.get(8));
B. cab = (Cabinet)(objAL).get(8);
C. cab = objAL.get(8);
D. More than one of the above
E. None of these

8. Suppose *cab1* is an object of type *Cabinet*. Which of the following returns a *Description* object?

A. Cabinet.cab1.getDescription( );
B. cab1.getDescr( );
C. cab1.descr;
D. (Description)cab1.getDescription( );
E. None of these

```
//Assume that the classes PositionName, Officer,
//Assistant, and Description already exist
public class Cabinet
{
        public Description getDescr( )
        {
                return descr;
        }
        …constructor and other methods not
        shown…

        //State variables
        private PositionName positionName;
        private Officer indivName;
        private Assistant underlings;
        private Description descr;
}
```

9. What replaces **<#1>** in the code to the right so that the value of the *Integer* stored at index 2 of the *lst* object is placed into *int j*?

A. int j = lst.get(2);
B. Object ob = lst.get(2);
   Integer ij = (Integer)ob;
   int j = ij;
C. int j = (Integer)lst.get(2);
D. int j = lst.getValue(2);
E. More than one of these

```
List<Integer> lst = new ArrayList<Integer>( );
lst.add(57);
lst.add(-102);
lst.add(57);
<#1>
```

Project… Big Bucks in the Bank

Create a project called *BigBucks*. It will have two classes in it, a *Tester* class and a *BankAccount* class. If you still have your *BankAccount* class from Lesson 15, just paste it into the new project. If not, the code for *BankAccount* follows:

```
public class BankAccount
{
        public BankAccount(String nm, double amt)
        {
           name = nm;
           balance = amt;
        }

        public void deposit(double dp)
        {
           balance = balance + dp;
        }

        public void withdraw(double wd)
        {
           balance = balance - wd;
        }
        public String name;
        public double balance;
}
```

You will need to create a *Tester* class that has a *main* method that provides a loop that lets you enter several *BankAccount* objects. As each is entered, it will be added to an *ArrayList* object. After several accounts have been entered, a loop will step through each *BankAccount* object in the *ArrayList* and decide which account has the largest balance that will then be printed. Following is the output screen after a typical run:

```
Please enter the name to whom the account belongs. ("Exit" to abort) Jim Jones
Please enter the amount of the deposit. 186.22

Please enter the name to whom the account belongs. ("Exit" to abort) Bill Gates
Please enter the amount of the deposit. 102.15

Please enter the name to whom the account belongs. ("Exit" to abort) Helen Hunt
Please enter the amount of the deposit. 1034.02

Please enter the name to whom the account belongs. ("Exit" to abort) Charles Manson
Please enter the amount of the deposit. 870.85

Please enter the name to whom the account belongs. ("Exit" to abort) exit

The account with the largest balance belongs to Helen Hunt.
The amount is $1034.02.
```

A partially complete *Tester* class is presented below. You are to complete the parts indicated in order to achieve the screen output above.

```
import java.io.*;
import java.util.*;  //includes ArrayList
import java.text.*;  //for NumberFormat
public class Tester
{
   public static void main(String args[])
   {
         NumberFormat formatter = NumberFormat.getNumberInstance( );
          formatter.setMinimumFractionDigits(2);
          formatter.setMaximumFractionDigits(2);
          String name;
          //Instantiate an ArrayList object here called aryList
          do
          {
                     Scanner kbReader = new Scanner(System.in);
                     System.out.print("Please enter the name to whom the account belongs.
                                                                          (\"Exit\" to abort)");
                     name = kbReader.nextLine( );

                     if( !name.equalsIgnoreCase("EXIT") )
                     {
                                System.out.print("Please enter the amount of the deposit. ");
                                double amount = kbReader.nextDouble();
                                System.out.println(" ");  //gives an eye-pleasing blank line
                                // Create a BankAccount object
                                // Add it to the ArrayList object
                     }
          }while(!name.equalsIgnoreCase("EXIT"));

          //Search aryList and print out the name and amount of the largest bank account
          BankAccount ba = get first account in the list
          double maxBalance = ba.balance;
          String maxName = ba.name;
          for(int j = 1; j < aryLst.size( ); j++)
          {
                     ?
                     ? Step through the remaining objects and decide which one has
                       largest balance (compare each balance to maxBalance)
                     ?
          }
          Print answer
      }
   }
```