**Being Dynamic ArrayLists**

Programmers often choose the array data structure by default, simply because it is straightforward and easy to use. Unfortunately, it has some limitations, such as a static pre-defined length and no built in support methods. Rather than force an array to handle a task for which it is ill-suited, Java provides several other data structures from which to choose, such as an arraylist.

Java's **ArrayList** class overcomes some of the limitations of arrays, but not without some concessions. **ArrayLists** are dynamic; they can change size. Furthermore, the **ArrayList** class offers a variety of methods that automate common data processing tasks (e.g., adding elements). Arrays can hold any variable type; however, **ArrayLists** can only hold objects.

**Part 1**

**ArrayLists** are more flexible than arrays, but there is some extra coding that requires great attention to detail. The practice exercises will identify many of the issues, so be sure to make the suggested changes in the interactive lessons so you do not waste time when you attempt to write your first program with an **ArrayList**.

**NOTES**

Arrays have the disadvantage that, once they have been created, their lengths can never be changed. We are often faced with situations, however, where the natural thing to do is to delete or insert array elements. One way to achieve this is to create a new array into which we then copy all the elements we want to keep together with any additional elements we want to insert. The following code illustrates how to use this technique to add a new element at the end of an array. Execute the program by clicking the **Run** button.

public static int[] addItemToIntArray( int[] oldArray, int newValue )   
  {   
    int i;   
    int[] newArray;   
  
    newArray = new int[ oldArray.length + 1 ];   
    for ( i = 0 ; i < oldArray.length ; i++ )   
      newArray[ i ] = oldArray[ i ];   
  
    newArray[ newArray.length - 1 ] = newValue;   
  
    return newArray;   
  }

**public static void main( String[] args )**

**{**

**int[] a = { 1, 2, 3, 4, 5 };**

**a = addItemToIntArray( a, 10 );**

**a = addItemToIntArray( a, 32 );**

**for ( int i : a )**

**System.out.println( i );**

**}**

**The technique used by addItemToIntArray causes a lot of work for Java. Each new addition of an element to the end of the array involves not only the creation of a brand new array, but it also requires all the old values to be copied into the new array one by one. With small arrays (such as the five-element array in the** [**example**](https://www.eimacs.com/eimacs/mainpage?epid=E2336182220&cid=162149#ExtendArrayEG) **on the previous page) this is not a problem, but arrays sometimes have thousands of elements; simply adding one element to the end of such an array would slow down the program significantly.**

**To overcome this difficulty, the Java language provides a class called ArrayList. An instance of ArrayList can be used to store data just like a regular Java array. Individual elements of arrays and ArrayList objects are accessed in similar ways, using an index. Unlike arrays, however, an ArrayList object allows us to change its length by deleting elements or inserting elements (at any point, not just at the end).**

**In declaring a variable of type ArrayList we use a statement like this:**

**ArrayList<String> aList;**

**The word between the *angle brackets*, <…>, indicates the data type of the elements that the ArrayList will store. In this case, aList is declared to be an ArrayList each of whose elements will be a String.**

**The following statement creates an ArrayList of Strings and then assigns it to aList:**

**aList = new ArrayList<String>();**

**We can also declare and assign to the variable in a single statement:**

**ArrayList<String> aList = new ArrayList<String>();**

If E is the name of a data type, the ArrayList<E> class is an example of a so-called generic class with type parameter whatever the replacement for E is. (Before we replace E by the name of an actual data type, it is called a formal type parameter.) It is also possible to declare an ArrayList without using a type parameter. Such a usage treats ArrayList as a so-called raw class. Raw ArrayLists require careful handling. However, they are not included in the AP subset and we have very little to do with them in this course.

**ArrayList<E> objects are provided with instance methods that allow us to**

* **discover the current length of the ArrayList;**
* **obtain the element at a particular index in an ArrayList;**
* **replace the element at a particular index in an ArrayList by something else;**
* **add a new element to an ArrayList;**
* **remove a specified element from an ArrayList; and**
* **remove *all* the elements from an ArrayList.**

**The following table provides all the necessary information concerning these instance methods, in each case specifying the method's return data type and signature and describing its operation:**

**ArrayList<E> instance methods**

**int size()**

***Return value:* The number of elements in the ArrayList.**

**E get( int index )**

***Return value:* The element of type E at the given index (which must be less than the *size* of the ArrayList).**

**E set( int index, E x )**

***Effect:* Replaces by x the element at the given index (which must be less than the *size* of the ArrayList).**

***Return value:* The element of type E that has been replaced.**

**boolean add( E x )**

***Effect:* Extends the ArrayList by inserting x after the former last element.**

***Return value:* The boolean true.**

**void add( int index, E x )**

***Effect:* Extends the ArrayList by inserting x as a new element at the given index (which must not be greater than the *size* of the ArrayList) and, if there is an element at that index, moving it and any elements beyond that point one place to the right.**

***Return value:* None.**

**E remove( int index )**

***Effect:* Shortens the ArrayList by removing the element at the given index (which must be less than the *size* of the ArrayList) and moving any elements beyond that point one place to the left.**

***Return value:* The removed element of type E.**

**void clear()**

***Effect:* Empties the ArrayList by removing all its elements, thereby reducing the *size* of the ArrayList to zero.**

***Return value:* None.**

**In this table, the formal type parameter E should be replaced by the name of the type of elements that the ArrayList contains. The resulting type parameter is therefore the name of a Java class, either a built-in one (such as String) or a defined one (such as Card or Book).**

**We remark in passing that int, double, and boolean are *not* permissible ArrayList type parameters. The reason for this is that these keywords name so-called *primitive data types* and ArrayLists do not accept primitive data types as elements. This problem can be sidestepped, however, in a very simple way that we explain in a short while.**

**The following code illustrates the creation and modification of an ArrayList<Card> a. Two elements are then stored in the ArrayList, a new element is added at index 1, and the element at index 2 is removed. At each stage the contents of a are printed out so that you can see the effect of the insertions and removals.**

**public static void displayCards( ArrayList<Card> a )**

**{**

**int i;**

**System.out.println( "Size is " + a.size() );**

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

**{**

**Card c = a.get( i );**

**System.out.println( "index " + i + ": " + c.name() );**

**}**

**}**

**public static void main( String[] args )**

**{**

**ArrayList<Card> a = new ArrayList<Card>();**

**a.add( new Card( "spades", 4 ) );**

**a.add( new Card( "clubs", 13 ) );**

**displayCards( a );**

**a.add( 1, new Card( "hearts", 14 ) );**

**displayCards( a );**

**a.remove( 2 );**

**displayCards( a );**

**}**

**Thanks to the remove method, we can easily delete items from an ArrayList. However, this method needs careful handling, as the following exercise shows:**

This exercise is concerned with the problem of deleting a block of items from an ArrayList.

1. Suppose you were asked to write a static method that, for any non-negative integer n, deletes the first n items from an ArrayList (or all the items of the ArrayList if it has fewer than n to begin with) and returns what remains of the initial ArrayList. Here is a first attempt (in the context of Strings):

**public static void deleteBlock( ArrayList<String> strings, int n )**

**{**

**for ( int i = 0; i < n; i++ )**

**{**

**if ( strings.size() > 0 )**

**strings.remove( i );**

**}**

**}**

**public static void main( String[] args )**

**{**

**String[] data = { "erosion", "rosion", "osion", "sion", "ion", "on", "n" };**

**ArrayList<String> dataList = new ArrayList<String>();**

**for ( int i = 0; i < data.length; i++ )**

**dataList.add( data[ i ] );**

**}**

**The cause of the undesirable behavior in part (a) is the fact that each time the remove method removes an item from an ArrayList, the size of that ArrayList is immediately reduced by 1 and the index of every item beyond the one that has been removed is also immediately reduced by 1. So during each iteration of the for loop in the above first attempt at defining a deleteBlock method — and as long as strings still has any items — the value of strings.size() is 1 less than during the previous iteration and the current value of i is now the index of a different ArrayList item than it was during the previous iteration.**

**By making a very small change in the above definition of deleteBlock, make it behave in the intended way. If you succeed, then Test Case #1 should result in an ArrayList that prints as [ sion, ion, on, n ] and Test Case #2 should result in an ArrayList that prints as [ on, n ].**

**Write a slightly more versatile version of the deleteBlock method, which, for any non-negative integer *n*, removes a block of *n* items from an ArrayList starting from the item with index start (or the entire "tail" of the ArrayList if there are fewer than *n* items from the item with index start to the end).**

**As** [**Exercise 108**](https://www.eimacs.com/eimacs/mainpage?epid=E2163741939&cid=162149#Exe100a) **has demonstrated, when invoking the remove method we must pay attention to the effect it has on the ArrayList, keeping in mind that this effect is threefold:**

1. **The item at the given index is removed;**
2. **any items to its right "slide down" to the left (that is, their indexes are reduced by 1); and**
3. **the size of the ArrayList is reduced by 1.**

**In the next exercise, all three side-effects come into play. You are asked to write a public static method that inputs an ArrayList of Strings, and that removes any String whose length is greater than 5. Sound simple?**

**Your first inclination may be to use a for loop, or perhaps a for-each loop to iterate over the list. However, since the size of the list will change as Strings are removed, a for loop is not particularly suited to this task. This is a case where it is appropriate to use the more general while loop. Taking a to be the ArrayList, your program will start like this:**

**int n = 0;   
  
while ( n < a.size() )   
{   
  ...   
}**

**Notice that the while loop will continue looping as long as items remain in the ArrayList.**

**The next part of the code is also obvious — we need to check the next String in the ArrayList to see if its length exceeds 5:**

**int n = 0;   
  
while ( n < a.size() )   
{   
  if ( a.get( n ).length() > 5 )   
  {   
    ...   
  }   
  else   
  {   
    ...   
  }   
}**

**We leave the remainder of the program for you to finish. However, we *strongly* recommend that you run through an example *by hand* before completing the missing segments. For example, consider the ArrayList containing the Strings**

**"pea", "lentil", "kidney", "pinto", "black-eyed", "lima"**

**With pencil in hand to keep a note of the current value of n, run through the list removing Strings containing more than 5 characters, *paying careful attention to where and when n should be incremented*. Then complete this exercise:**

**Complete the definition below for a static method transplantBlock which, for any non-negative integer *n*, removes a block of *n* elements from an ArrayList fromList starting from the element at index start and inserts them (in order) into an ArrayList toList starting at index target. (If there are fewer than *n* elements from the element with index start to the end of fromList, then the method transplants the entire "tail" of the ArrayList from the element with index start to the end.)**

**In addition to a no-input ArrayList constructor like this:**

**ArrayList<String> a = new ArrayList<String>();**

**Java also provides a constructor that allows us to specify an *initial capacity* for the ArrayList. This provides information that Java may use in order to improve the efficiency with which it executes the code. Here, for example, we indicate that Java should create an ArrayList of Strings with an initial capacity of 25 elements:**

**ArrayList<String> a = new ArrayList<String>( 25 );**

**If we do not specify an initial capacity, then Java assumes a default initial capacity of 10. Note that the initial capacity is not the same as the number of elements. No elements are actually stored until they are added to the ArrayList. So the size of an ArrayList may be less than, equal to, or greater than its initial capacity (whether declared or default). Run the following code to see an example in which the actual size is less than the initial capacity.**

**ArrayList<String> a = new ArrayList<String>( 25 );   
     System.out.println( "a has " + a.size() + " elements" );   
  
     a.add( "fred" );   
     System.out.println( "a has " + a.size() + " elements" );**

**Part 2**

**ArrayLists** and arrays operate in very similar ways with regard to accessing elements based on index positions; however, the restriction that all elements in an **ArrayList** must be objects (even if they are numbers) requires some additional coding. First, let us take a look at how **ArrayLists** of numbers and **Strings** are handled. Pay close attention to the methods used with ArrayLists.

* Create a new project called 07.01 ArrayLists in the Mod07 Lessons folder.
* Download the [IntegerArrayList.java](https://lti.flvsgl.com/flvs-cat-content/le3jek5jruhrbfo0sq9cet9nvn/flvs-cat-session/apcomputersciencea_v20/module07/lesson01/docs/08_10b/IntegerArrayList.java) class to the newly-created project.
* Open the [07.01 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/le3jek5jruhrbfo0sq9cet9nvn/flvs-cat-session/apcomputersciencea_v20/module07/lesson01/pop/08_10b/08_10b_pop01.htm).
* Very carefully read each document and make the recommended modifications to the demo programs.
* Download the [StringArrayList.java](https://lti.flvsgl.com/flvs-cat-content/le3jek5jruhrbfo0sq9cet9nvn/flvs-cat-session/apcomputersciencea_v20/module07/lesson01/docs/08_10b/StringArrayList.java) class to the 07.01 project folder. Carefully study the source code. Much of the code is commented out, so you will have to modify it step by step as you study how Strings can be used with **ArrayLists**.

**Part 3**

Even though integer and decimal numbers are stored as objects in an ArrayList, it does not affect using them in calculations.

Download the [CalculationsWithArrayLists.java](https://lti.flvsgl.com/flvs-cat-content/le3jek5jruhrbfo0sq9cet9nvn/flvs-cat-session/apcomputersciencea_v20/module07/lesson01/docs/08_10b/CalculationsWithArrayLists.java) file to the 07.01 ArrayLists project folder. Carefully study the code. Try to modify the program to perform different arithmetic or to use doubles instead of integers.