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](http://pages.eimacs.com/eimacs/mainpage?epid=E2385368770&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 class Card   
{   
  private String mySuit;   
  private int myValue;   
  
  public Card( String suit, int value )   
  {   
    mySuit = suit;   
    myValue = value;   
  }   
  
  public String name()   
  {   
     String[] cardNames =    
      {   
        "Deuce", "Three", "Four", "Five",   
        "Six", "Seven", "Eight", "Nine", "Ten",   
        "Jack", "Queen", "King", "Ace"   
      };   
  
    return cardNames[ myValue - 2 ] + " of " + mySuit;   
  }   
}   
  
public class MainClass   
{   
  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 );  
  }    
}

Size is 2   
index 0: Four of spades   
index 1: King of clubs   
Size is 3   
index 0: Four of spades   
index 1: Ace of hearts   
index 2: King of clubs   
Size is 2   
index 0: Four of spades   
index 1: Ace of hearts

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 ].

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

{

  for ( int i = 0; i < n; i++ )

  {﻿

    if ( strings.size() > start )

      strings.remove( start );

  }

}

public static void main( String[] args )

{

  String[] data = { "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k" };

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

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

﻿    dataList.add( data[ i ] );

  deleteBlock( dataList, 6, 10 );

  System.out.println( dataList );

}

[a, b, c, d, e, f]

As [Exercise 108](http://pages.eimacs.com/eimacs/mainpage?epid=E2375531562&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:

public static void removeLongStrings( ArrayList<String> a )

{

  int n = 0;

  while ( n < a.size() )

  {

    if ( a.get( n ).length() > 5 )

    {

      a.remove(n);

    }

    else

    {

      n++;

    }

  }

}

public static void main( String[] args )

{

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

  String[] data = { "Avoid", "obfuscation", "censorship",

                    "embarrassing", "tics", "tergiversation",

                    "and", "lies" };

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

    list.add( data[ i ] );

  removeLongStrings( list );

  System.out.println( list );

}

[Avoid, tics, and, lies]

A different approach to this problem is to remove the items starting from the end. Complete and test this alternative definition (this time we provide less of the skeleton):

public static void removeLongStrings( ArrayList<String> a )

{

  int n = a.size() - 1;

  while ( n >= 0 )

  {

    if ( a.get( n ).length() > 5 )

    {

      a.remove(n);

    }

     n--;

  }

}

public static void main( String[] args )

{

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

  String[] data = { "Avoid", "obfuscation", "censorship",

                    "embarrassing", "tics", "tergiversation",

                    "and", "lies" };

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

    list.add( data[ i ] );

  removeLongStrings( list );

  System.out.println( list );

}

[Avoid, tics, and, lies]

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" );

a has 0 elements   
a has 1 elements

[Earlier](http://pages.eimacs.com/eimacs/mainpage?epid=E2355856912&cid=162149#BadTypeParams) we remarked that the keywords int, double, and boolean are not permissible ArrayList type parameters — that is, we are not allowed to use them as replacements for the formal type parameter E in the expression ArrayList<E>. These are three of the Java keywords that represent so-called primitive data types, and ArrayLists are not allowed to contain elements of any primitive data type.

Java requires that each element of an ArrayList shall be of a reference data type (that's the technical name for a non-primitive data type). The principal difference between elements of a primitive data type and those of a reference data type lies in how their values are accessed. The value of an object of a primitive data type is accessed directly when we use the name of the object. On the other hand, the value of an object of a reference data type is accessed indirectly via a pointer (or reference) that indicates where the actual value is located. Note, however, that you are not expected to understand such technical matters for the Advanced Placement examination.

This state of affairs is unfortunate, because it would be really useful if we could store ints and doubles (in particular) in collections that can grow and shrink just as ArrayLists can. To get around this problem, Java provides a built-in reference data type corresponding to each primitive data type. These special reference data types are known as wrapper classes because they "wrap" primitive data type values in the additional layer of complexity possessed by reference data type objects. Each wrapper class must be used in place of the corresponding primitive data type when creating ArrayLists. As far as the three primitive data types we have just mentioned are concerned, the wrapper classes are as follows:

|  |  |
| --- | --- |
| Primitive data type | Wrapper Class |
| int | Integer |
| double | Double |
| boolean | Boolean |

With the help of these wrapper classes, we operate as follows: If we want to store some ints in an ArrayList, we declare the ArrayList as an

ArrayList<Integer>,

using the wrapper class that corresponds to the int primitive data type. We then add ints to the ArrayList without concerning ourselves about the fact that they are ints rather than Integers — Java automatically performs the necessary conversion. In the other direction, we can retrieve ints from an ArrayList of Integers using the get method. Whenever the result of such a retrieval needs to be an int rather than an Integer, Java automatically takes care of the necessary conversion. The following code, for example, uses an ArrayList to manipulate ints. First, an array of ints is copied into an ArrayList. Then all the 5s are removed from the ArrayList and the remaining elements are printed:

    ArrayList<Integer> a = new ArrayList<Integer>();   
  
    int[] someints = { 5, 2, 5, 7, 5, 8, 5, 2 };   
  
    // store elements of someints into the ArrayList   
    for ( int y : someints )   
      a.add( y );   
  
    // remove all the 5's   
    int q = 0;   
    while ( q < a.size() )   
    {   
       if ( a.get( q ) == 5 )   
         a.remove( q );   
       else   
         q++;   
    }   
  
    // print the result   
    for ( int p : a )   
      System.out.println( p );

2   
7   
8   
2

The automatic conversions that occur during the execution of this code are as follows:

* Each addition of an int to the ArrayList in the first for-each loop involves the automatic conversion of an int to an Integer.
* Each comparison of 5 with the result of getting an element from the ArrayList involves the automatic conversion of that result from an Integer to an int.
* In the final for-each loop, because the variable p is declared as an int, each successive assignment of an element of the ArrayList to p involves the automatic conversion of an Integer to an int.

When working with ArrayLists, there is nasty little "tiger trap" that must be avoided at all costs: for-each loops and operations that entail changing the size of an ArrayList do not mix. Consider the following example, in which the intention is to create an ArrayList containing the first 20 odd numbers and then remove all those that are multiples of 7.

ArrayList<Integer> a = new ArrayList<Integer>( 20 );

for ( int k = 0; k < 20; k++ )

    a.add( 2 \* k + 1 );

for ( int i : a )

    if ( i % 7 == 0 )

        a.remove( i );

System.out.println( a );

When you press the **Run** button you will find that an error is reported that starts like this:

Exception in thread "main" java.util.ConcurrentModificationException

This happens because for-each loops work their way down an ArrayList using a behind-the-scenes structure called an *iterator*. This keeps track of where the loop is in the ArrayList by using a technique that relies on knowing how long the ArrayList is. If you do anything to change the size of the ArrayList (by removing or adding items or by clearing the ArrayList, for example), you mess up the operation of the iterator, which then throws the ConcurrentModificationException as if to tell you: "Don't change the size of this ArrayList while I'm iterating my way along it!"

There are a number of ways to overcome this problem. One is to use a regular for loop, working from the end of the ArrayList toward the start, like this:

for ( int i = a.size() - 1; i >= 0; i-- )   
    if ( a.get( i ) % 7 == 0 )   
        a.remove( i );

In the interests of full disclosure, we should let you know that there is much more going on here than meets the eye. In the original for-each loop, because the loop variable i is declared as an int, each Integer in the ArrayList a is auto-unboxed one at a time and its int value is assigned to i. Then in the call to the remove method we are using a version of that method that has a different signature than the one that's in the AP Java subset. The input to the AP Java subset version of remove specifies the index of the item to be removed. But there is also a version of remove that accepts an object of the same type as the items in the ArrayList. This causes the item to be removed from the ArrayList that has the smallest possible index and that is equal to the input object. Of course, in this case, since i is an int, some auto-boxing has to occur to transform it into an Integer for the remove method to remove.

In the regular for-loop, the loop variable i is clearly intended to serve as an index. This is evident because the divisibility test requires the use of the get method to obtain the relevant item from the ArrayList, and that item is then auto-unboxed to obtain an int to which the modulus operation can be applied. In this loop, the version of the remove method that is being used is the one that is in the AP Java subset.