The List<E> interface represents an ordered collection (or sequence) of objects. (As in the case of the ArrayList<E> class, the E that appears in the name of the List<E> interface is a *formal type parameter*. Each time it is replaced by the name of a [reference data type](https://www.eimacs.com/eimacs/mainpage?epid=E1991301592&cid=162149), we get an interface that prescribes methods for lists of that type of object.) The following partial definition of the List<E> interface omits most of the details that are *not* required by the Advanced Placement examination:

public interface List<E> // inheritance not shown   
{   
  boolean add( E x );   
  void add( int index, E x );   
  E get( int index );   
  E remove( int index );   
  E set( int index, E x );   
  int size();   
  
  // the next two methods are not in the AP Java subset   
  Iterator<E> iterator();    
  ListIterator<E> listIterator();   
  
  // other methods not shown   
}

And the following table describes how these methods are intended to behave when defined in a concrete class that implements the List<E> interface. (Bear in mind that, like arrays, List<E>s use zero-based indexing; that is, the first element in a List<E> has index 0.)

List<E> instance methods

boolean add( E x )

*Effect:* Extends the list by inserting x after the former last element and adjusts the size of the list.

*Return value:* The boolean true.

void add( int index, E x )

*Effect:* Extends the list by inserting x as a new element at the given index and, if there is an element at that index, moving it and any elements beyond that point one place to the right; adjusts the size of the list.

*Return value:* None.

*Throws:* An *IndexOutOfBoundsException* if index is negative or greater than the size of this list.

E get( int index )

*Return value:* The element at the given index.

*Throws:* An *IndexOutOfBoundsException* if index is negative or greater than or equal to the size of this list.

E remove( int index )

*Effect:* Shortens the list by removing the element at the given index and moving any elements beyond that point one place to the left; adjusts the size of the list.

*Return value:* The element formerly at the given index.

*Throws:* An *IndexOutOfBoundsException* if index is negative or greater than or equal to the size of this list.

E set( int index, E x )

*Effect:* Replaces the element at the given index by x.

*Return value:* The element formerly at the given index.

*Throws:* An *IndexOutOfBoundsException* if index is negative or greater than or equal to the size of this list.

int size()

*Return value:* The number of elements in this list.

Iterator<E> iterator()

*Return value:* An Iterator<E> object that will iterate over the elements in this list in proper sequence.

ListIterator<E> listIterator()

*Return value:* A ListIterator<E> object that will iterate over the elements in this list in proper sequence.

(We tell you what [Iterator<E>s](https://www.eimacs.com/eimacs/mainpage?epid=E2306680663&cid=162149) and [ListIterator<E>s](https://www.eimacs.com/eimacs/mainpage?epid=E2395214240&cid=162149) are later on, in the *Advanced Topics* section.) There are two essential features of the List<E> interface that distinguish it from other interfaces, namely:

* Elements in a List<E> can be referenced by index.
* A List<E> may contain duplicate elements. That is, the same object may appear several times.

The above listing of methods of the List<E> interface probably reminds you of a similar listing we gave of the [instance methods](https://www.eimacs.com/eimacs/mainpage?epid=E2375531528&cid=162149#ArrayListMethods) of the ArrayList<E> class. This is no surprise since ArrayList<E> implements the List<E> interface. However, it is not the only implementation, nor is it always the most efficient, as we shall see in the *Advanced Topics* section.

[Earlier](https://www.eimacs.com/eimacs/mainpage?epid=E2001138886&cid=162149), we mentioned that for-each loops may be used to iterate over the elements of ArrayLists because ArrayLists are *iterable*:

    ArrayList<String> a = new ArrayList<String>();   
  
    a.add( "fred" );   
    a.add( "jane" );   
    a.add( "emma" );   
  
    for ( String t : a )   
      System.out.println( t );

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E2124394871)

fred   
jane   
emma

We are now in a position to tell you exactly what "iterable" means. In fact, it is almost laughably simple: for each reference data type E, Java provides a built-in interface, Iterable<E>, which is designed solely for the purpose of allowing the use of for-each loops on instances of classes that implement the interface. For each reference data type E, the ArrayList<E> class implements the List<E> interface (as we have just informed you), and the List<E> interface is a subinterface of the Iterable<E> interface. Consequently, for each reference data type E, the ArrayList<E> class implements the Iterable<E> interface, and that is why we can use for-each loops to iterate over the elements of an ArrayList<E> instance.

The Iterable<E> interface (where E is a reference data type) is *not* part of the AP Java subset. We provide an optional opportunity for you to learn more about it, however, [later on](https://www.eimacs.com/eimacs/mainpage?epid=E2085051264&cid=162149), in the *Advanced Topics* section of this course.

Previously, when you would use an array list, you would declare a variable to be of type ArrayList. For example, in the **CalculationsWithArrayLists.java** from lesson 8.10, this is used:

ArrayList<Integer> numList1 = new ArrayList<Integer>();

While it is written correctly, this code is not considered proper as it lacks some flexibility. To make the code more in line with object-oriented programming practices, you should actually create the array like this:

List<Integer> numList1 = new ArrayList<Integer>();