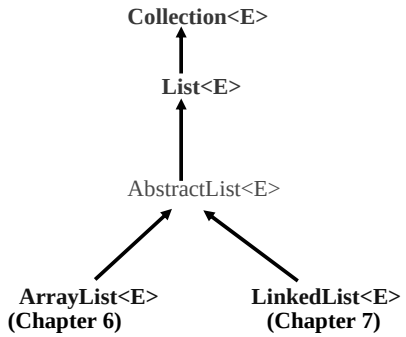
Chapter 6

Array-Based Lists




```
public interface Collection<E>
{
    int size();
    boolean isEmpty();
    boolean contains(Object obj);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E element);
    boolean remove(Object obj);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object obj);
    int hashCode();
}
```

Here are some method specifications in the List **interface:**

```
// Returns the element at position index in this List object.
E get (int index);

// Replaces the element that was at position index in this
// List object with the
// parameter element, and returns the previous occupant.
E set (int index, E element);

// Returns the index of the first occurrence of obj in this List
//  object, if
//  obj appears in this List object..  Otherwise, returns -1.
int indexOf (Object obj);
```

```
// Inserts element at position index in this List object; every
// element that
// was at a position >= index before this call is now at the
// next higher position.
void add (int index, E element);

// Removes and returns the element at position index in
// this List object; every
//  element that was at a position > index before this call is
//  now at the next lower position.
E remove (int index);
```

ArrayList**: A *classy* array**

Random access of an element from its index:
public E get (**int** index),
public E set (**int** index, E element)

Insertion or removal of an element at a given index:
public void add (**int** index, E element)
public E remove (**int** index)

Automatic resizing after
public void add (**int** index, E element)
public boolean add (E element) // at back

User's view of ArrayList **class:**

1. **public** ArrayList (**int** initialCapacity)
2. **public** ArrayList( )
3. **public** ArrayList (Collection<? **extends** E> c)
4. **public boolean** add (E element) // inserts at back
5. **public void** add (**int** index, E element)
6. **public boolean** addAll (Collection<? **extends** E> c)
7. **public boolean** addAll (**int** index, Collection<? **extends** E> c)
8. **public void** clear( ) // worstTime (n) is O (n)
9. **public** Object clone( )
10. **public boolean** contains (Object obj)
11. **public boolean** containsAll (Collection<?> c)

12. **public void** ensureCapacity (**int** minCapacity)
13. **public boolean** equals (Object obj)
14. **public** E get (**int** index) // worstTime(n) is constant
15. **public int** hashCode( )
16. **public int** indexOf (Object obj)
17. **public boolean** isEmpty( )
18. **public** Iterator<E> iterator( )
19. **public int** lastIndexOf (Object obj)
20. **public** ListIterator<E> listIterator( )
21. **public** ListIterator<E> listIterator (**final int** index)
22. **public boolean** remove (Object obj)

23. **public** E remove (**int** index)
24. **public boolean** removeAll (Collection<?> c)
25. **public boolean** retainAll (Collection<?> c)
26. **public** E set (**int** index, E element)
27. **public int** size( )
28. **public** List<E> subList (**int** fromIndex, **int** toIndex)
29. **public** Object[ ] toArray( )
30. **public** <T> T[ ] toArray (T[ ] a) // ClassCastException **//** unless T extends E
31. **public** String toString()
32. **public void** trimToSize( )

Example: Here is a processInput (String s) **method that starts by converting** s **to int** n **and then**

0. **Constructs an** ArrayList **that holds up to n** Double **values.**
   public ArrayList (**int** initialCapacity)
1. **In a loop with** i **going from** 0 **to** n – 1**, appends** i **to the** ArrayList**. public boolean** add (E element)
2. **Inserts** 1.4 **at index** n / 3**.**
   public void add (**int** index, E element)
3. **Removes the element at index** 2n / 3.
   public E remove (**int** index)
4. **Multiplies the middle element by** 3.5.
   public E get (**int** index)
   public E set (**int** index, E element)
5. **Prints out the** ArrayList**; public** String toString()

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                new ArrayList<Double> (n);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                new ArrayList<Double> (n);

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

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                    new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
    myList.remove (2 * n / 3);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                    new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
    myList.remove (2 * n / 3);
    double d = myList.get (n / 2) * 3.5;
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                    new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
    myList.remove (2 * n / 3);
    double d = myList.get (n / 2) * 3.5;
    myList.set (n / 2, d);
```

```
public void processInput (String s)
{
    int n = Integer.parseInt (s);
    ArrayList<Double> myList =
                    new ArrayList<Double> (n);

    for (int i = 0; i < n; i++)
        myList.add (i + 0.0);
    myList.add (n / 3, 1.4);
    myList.remove (2 * n / 3);
    double d = myList.get (n / 2) * 3.5;
    myList.set (n / 2, d);
    System.out.println (myList);
} // method processInput
```

If the input is 10, the output is

[0.0, 1.0, 2.0, 1.4, 3.0, 14.0, 6.0, 7.0, 8.0, 9.0]

The ArrayList **class heading:**

```
public class ArrayList<E> extends AbstractList<E>
                          implements List<E>,
                                     RandomAccess,
                                     Cloneable,
                                     java.io.Serializable
```

ArrayLists **are** Serializable**:**

They can be saved to disk as a series of bytes, and can later be read from disk (de-serialized). See Appendix 3.

ArrayLists **are** Cloneable**:**

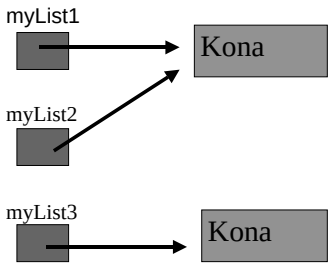
```
// Postcondition: a distinct copy of this ArrayList has been
//                returned. The worstTime (n) is O (n).
public Object clone( );
```

Don't use clone( ): It creates an object without calling a constructor, so you lose type safety:

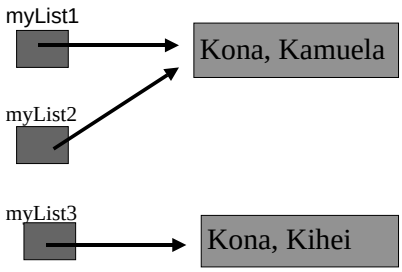
```
ArrayList<String> original = new ArrayList<String>();
original.add ("yes");
ArrayList<Integer> copy =
    (ArrayList<Integer>)original.clone();

System.out.println (copy.get (0));
```

The output is "Yes"!!!!!!

```
ArrayList<String> myList1 = new ArrayList<String>( ),
                            myList2,
                            myList3;


myList1.add ("Kona");
myList2 = myList1;  // two references to same ArrayList
myList3 = new ArrayList<String> (myList1);
                            // copy of myList1
```




```
ArrayList<String> myList1 = new ArrayList<String>( ),
                            myList2,
                            myList3;


myList1.add ("Kona");
myList2 = myList1;  // two references to same ArrayList
myList3 = new ArrayList<String> (myLis1);
                            // copy of myList1

myList2.add ("Kamuela");
myList3.add ("Kihei");
```



Fields in the ArrayList **class**

private transient E[ ] elementData;
// "transient" means that the array elementData need // not be saved if the ArrayList object is serialized. The // individual elements will be saved, but not the array.

private int size;

```
// Initializes this ArrayList object to be empty
and with a // capacity given by initialCapacity.
public ArrayList (int initialCapacity)
{
        elementData = (E[ ]) new Object [initialCapacity];
} // constructor with int parameter
```

```
  // Initializes this ArrayList object to be empty.
  public ArrayList ( )
  {
        this (10);
  }
```

```
 // Appends element to this ArrayList object and
 // returns true.  The averageTime(n) is constant
 and // worstTime(n) is O (n).
 public boolean add (E element)
 {
       ensureCapacity (size + 1);

       elementData [size++] = element;

       return true;
 }
```

```
public void ensureCapacity(int minCapacity) {

        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {

            // Increase the capacity by at least 50%,
            // and copy the old array to the new array.
        }
}
```

```
public void ensureCapacity(int minCapacity) {

        modCount++;   // See Appendix 2
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {

            E oldData[] = elementData;
            int newCapacity = (oldCapacity * 3) / 2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            elementData = (E[ ]) new Object[newCapacity];
            System.arraycopy(oldData, 0, elementData, 0,
                                    size);
        }
}
```

```
// Initializes this ArrayList to a copy of c.
public ArrayList(Collection<? extends E> c)
{
        size = c.size();

        // Allow 10% room for growth
        elementData =
                (E[])new Object[(int)Math.min(
                (size*110L)/100,Integer.MAX_VALUE)];
        c.toArray(elementData);
}
```

Note: This is called the copy constructor.

Exercise: Complete the definition of the following method:

```
public boolean addAll(Collection<? extends E> c)
{
    Object [ ] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);
    System.arraycopy(/*   */);
    size += /*  */;
    return /*  */;
}
```

Iterators – not needed for ArrayList**S**

```
for (int j = 0; j < myList.size( ); j++)
        System.out.println (myList.get (j));
```

But iterators are legal:

```
Iterator<Double> itr = myList.iterator( );
while (itr.hasNext( ))
        System.out.println (itr.next( ));
```

Even better:

```
for (Double d : myList)
        System.out.println (d);
```

Application

High-Precision Arithmetic

In public-key cryptography, the integers are hundreds of digits long.

Key facts:

1. **To generate a very long integer of n digits that is prime, averageTime(n) is $O(n^3)$.**

   If n = 200, $n^3 = 200^3 = 8{,}000{,}000$.

2. **To factor a very long integer of n digits that is not prime, averageTime(n) is $\Omega(10^{n/2})$ for any known method.**

   If $n = 200$, $10^{n/2} = 10^{100}$, A GOOGOL!

3. **Given primes p and q, pq is used to encode a public message.**

4. To decode the message, p and q must be known.

We will now develop a VeryLongInt **class to handle very long integers.**

In the method descriptions, *n* refers to the number of digits in the calling object.

```
/** Initializes this VeryLongInt object from a given String
 * object. The worstTime(n) is O(n), where n represents
 *  the number of characters in s.
 *
 *  @param s – the given String object.
 *
 *  @throws NullPointerException – if s is null.
 *
 */
public VeryLongInt (String s)
```

```
/**
 *  Returns a String representation of this VeryLongInt
 *  object. The worstTime(n) is O(n), where n represents
 *  the number of digits in this VeryLongInt object.
 *
 *  @return a String representation of this VeryLongInt
 *          object: [highest digit, next highest digit, …].
 *
 */
public String toString()
```

```
/**
 * Increments this VeryLongInt object by a specified
 * VeryLongInt object.
 * The worstTime(n) is O(n), where n  is the number of
 * digits in the larger of this VeryLongInt object (before the
 * call) and the specified VeryLongInt object.
 *
 *  @param otherVeryLong – the specified VeryLongInt
 *          object to be added to this VeryLongInt object.
 *
 *  @throws NullPointerException – if otherVeryLong is
 *              null
 *
 */
public void add (VeryLongInt otherVeryLong)
```

VeryLongInt Applet:

http://www.cs.lafayette.edu/~collinsw/verylong/long.html

For example, the following code constructs two VeryLongInt **objects with values 345 and 6789 and prints out their sum.**

```
VeryLongInt very1 = new VeryLongInt ("345");
VeryLongInt very2 = new VeryLongInt ("6789");

very1.add (very2);
System.out.println (very1);    // = System.out.println
                               // (very1.toString( ));
```

Fields and method definitions for the VeryLongInt **class**

The only field is:

```
protected ArrayList<Integer> digits;   // holds all of the
                    // digits in this VeryLongInt
```

```
public VeryLongInt (String s)
{
        final char LOWEST_DIGIT_CHAR = '0';

        digits = new ArrayList<Integer> (s.length());

        for (int i = 0; i < s.length(); i++)
        {
                char c = s.charAt (i);
                if (Character.isDigit(c))
                   digits.add (c – LOWEST_DIGIT_CHAR);
         } // for
} // constructor with string parameter
```

```
public String toString( )
{
       return digits.toString();
 } // method toString
```

For the add **method:**

```
VeryLongInt a1,
            a2;

// a1 = {3, 8, 7, 4, 9, 8}

// a2 = {5, 3, 6, 4}

a1.add (a2);
```

Loop 6 times:

0: 8 + 4 = 12
Append 2, and the carry is 1.

1: 9 + 6 + 1 (the carry) = 16
Append 6 and the carry is 1.

...

We end up with 268293, so reverse.

```
public void add (VeryLongInt otherVeryLong) {

        final int BASE = 10;

        int largerSize,
            partialSum,
            carry = 0;

        if (digits.size() > otherVeryLong.digits.size())
                largerSize = digits.size();
        else
                largerSize = otherVeryLong.digits.size();

        ArrayList<Integer>sumDigits =
                new ArrayList<Integer>(largerSize + 1);
```

```
        for (int i = 0; i < largerSize; i++) {

                // Add the ith least significant digit in the
                // calling object, the ith least significant digit
                // in otherVeryLong, and the carry.

                // Append that sum % 10 to sumDigits
                // The new carry is that sum / 10.

        } // for

        if (carry == 1)
                sumDigits.add (carry);
        Collections.reverse (sumDigits);
        digits = sumDigits;
} // method add
```

```
        for (int i = 0; i < largerSize; i++)
        {
                partialSum = least (i) +
                        otherVeryLong.least (i) + carry;
                carry  = partialSum / BASE;
                sumDigits.add (partialSum % BASE);
        } // for

        if (carry == 1)
                sumDigits.add (carry);
        Collections.reverse (sumDigits);
        digits = sumDigits;
  } // method add
```

Suppose the VeryLongInt **object has the value 13579. Then**

least (0) **returns 9**
least (1) **returns 7**
least (2) **returns 5**
least (3) **returns 3**
least (4) **returns 1**
least (5) **returns 0**

```
/** Returns the ith least significant digit in digits if i is a non-
 *  negative int less than digits.size().  Otherwise, returns 0.
 *
 *  @param i – the number of positions  from the right-most digit in
 *                digits to the digit sought.
 *
 *  @return ith least significant digit in digits, or 0 if no such digit.
 *
 *  @throws IndexOutOfBoundsException – if i is negative.
 *
 */
protected int least (int i)
{
        if (i >= digits.size())
                return 0;
        return digits.get (digits.size() - i - 1);
} // least
```

How long does the add **method take, on average? Recall that *n* refers to the number of digits.**

The least **method takes constant time, and appending to an** ArrayList **takes constant time, on average. So the average time for** add **depends only on the number of iterations of the** for **loop: *n*.**

The averageTime(*n*) is linear in *n*.

Exercise: In the VeryLongInt class, define the size() method:

```
// Returns the number of digits in this VeryLongInt
// object.
public int size( )
```