31 Lists

1 ArrayList

One of the drawbacks of arrays is that they do not make it easy to accommodate collections of arbitrary size. We have to commit ourselves to a fixed size when we introduce an array, and this imposes lots of tedious coding if we are not to abandon the computation should the array fill up. Java provides class `ArrayList` to handle all this coding for us. An instance of an `ArrayList` is called an *array list*; you can think of it as an array that may grow as big as it needs to during program execution. `ArrayList` is part of the `java.util` package (which must be explicitly imported).

`ArrayList` is parametrised with respect to the type of its elements: **`ArrayList<T>`**. Throughout, we will use `T` as the type parameter. The constructors are as follows:

ArrayList() ArrayList(*ArrayList*<*T*>)

`new ArrayList()` creates an empty list, and **`new ArrayList(c)`** creates a list with the same elements as `c` in the same order. Actually, this constructor has a more general form:

ArrayList(*Collection*<*T*>)

We write `Collection` to stand for `ArrayList` or any one of several other kinds of collection we will meet later, including `LinkedList`, `HashSet` and `TreeSet`. In each case an array list is created using the elements from the supplied collection. We remark as an aside for readers who have studied inheritance that whenever a collection of elements of type `T` is expected, it is allowable to supply a collection whose elements inherit from `T`.

Array lists have indices just like arrays, i.e. the indices run from 0 up. The most fundamental methods supplied by `ArrayList` are:

int size()
T set(*int*, *T*)
T get(*int*)
boolean add(*T*)

String toString()

`t.size()` returns the number of elements in `t`. Remember to use `size()` for array lists and not `length` (which is used for arrays) or `length()` (which is used for strings). **`t.set(i,o)`** replaces the element at position `i` in `t` with `o`, and returns a reference to the object previously at position `i` (the returned reference is often ignored). The value of `i` must be in the range 0 to `t.size()-1`. `t.set(i,o)` is a rough analogue of `b[i]=o` for `b` an array (it is illegal to write `t[i]=o` for `t` an array list – you must write `t.set(i,o)`). **`t.get(i)`** returns a reference to the element at position `i` in `t`. `t.get(i)` is the analogue of `b[i]` for `b` an array. It is illegal to write `t[i]` for `t` an array list – you must write `t.get(i)`. **`t.add(o)`** appends `o` to the end of `t`. The boolean returned is not of interest and is always ignored. **`t.`** We can add as many elements as we like, limited only by the available memory. **`t.toString()`** returns a string representation of `t` (encased in square brackets). Each element is represented by the string returned by its `toString()` method.

The following code illustrates the behaviour of the above methods.

```
ArrayList<String> s = new ArrayList<String>();   // s is empty
s.add("dog"); s.add("cat");                       // s: dog, cat
s.add("pig"); s.add("cow");                       // s: dog, cat, pig, cow
String word2 = s.get(2);                          // word2: pig
String word3 = s.set(3, "cat");                   // s: dog, cat, pig, cat
System.out.print(word3);                          // cow appears
System.out.print(s.size());                       // 4 appears
System.out.print(s);                              // [dog, cat, pig, cat] appears
ArrayList<String> t = new ArrayList<String>(s);   // t: dog, cat, pig, cat
```

The enhanced for-each loop can be used with lists, just as for arrays. If `c1` is a list (or any collection) of strings, say, then the following for-each loop may be used:

```
for (String w: c1) {
    .....
}
```

This causes the body of the loop (represented by ..... above) to be executed once for each `w` in `c1` (in the order in which they occur in the list). Each time the body of the loop is executed, `w` stands for the element of `c1` being processed. The type of `w` must match the type of the elements in `c1` (here `w` has type `String` because `c1` is a list of strings).

Example 1: big words

The following program prints a list of the big words in a text file. A word is defined to be "big" if its length exceeds the average length of *all* the words in the file. The file is supplied through the standard input. For example, invoking

```
java BigWords < source.txt
```

will produce a list of the words in `source.txt` whose length exceeds the average length of all the words in `source.txt`. Any repetitions of words in the input show up in the output. We use array lists rather than arrays because we will have to store all the words in the input, and it is convenient to do so without committing to a maximum length.

```
import java.util.*;
class BigWords {
    public static void main(String[] args) {
        // Read input & compute average length of words
        ArrayList<String> words = new ArrayList<String>(); // for words in input
        int total = 0; // total of all word lengths
        while (!Console.endOfFile()) {
            String s = Console.readToken(); // next word
            total = total+s.length();
            words.add(s);
        }
        int meanLength = total/words.size(); // average word length (truncated)
        // Print big words
        for (String s: words) {
            if (s.length()>meanLength)
                System.out.println(s);
        }
    }
}
```

The for-each loop could also have been written as follows:

```
        for (int i=0; i<words.size(); i++) {
            String s = words.get(i);
            if (s.length()>meanLength) {
                System.out.println(s);
            }
        }
```

■

2 Inserting and deleting elements

`ArrayList<T>` provides the following additional methods, mainly concerned with inserting and deleting elements.

void add(*int*, *T*)
T remove(*int*)

boolean remove(*Object*)
boolean contains(*Object*)
int indexOf(*Object*)
void clear()
boolean isEmpty()

`t.add(i,o)` inserts `o` at position `i` in `t`, shifting any following elements one position "to the right". The value of `i` must be in the range 0 to `t.size()`. `t.add(t.size(),o)` appends `o` to `t`, i.e. its effect on `t` is the same as `t.add(o)`. **`t.remove(i)`** removes the element at position `i` in `t`, shifting any following elements to the left. A reference to the object deleted is returned (although it is often ignored by the caller). The value of `i` must be in the range 0 to `t.size()-1`. **`t.remove(o)`** removes the first occurrence of item `o` from list `t`. If `t` does not contain `o`, `t` is unchanged. A boolean is returned indicating whether `t` contained `o` (the returned boolean is often ignored).. **`t.contains(o)`** returns a boolean indicating whether `t` contains `o`. **`t.indexOf(o)`** returns the index of the first occurrence of `o` in `t`; it returns -1 if `o` is not present. **`t.clear()`** removes all the elements from `t`. **`t.isEmpty()`** tests if `t` is empty.

The following code illustrates the behaviour of the some of the above methods.

```
ArrayList<String> s = new ArrayList<String>();     // s is empty
s.add("pig"); s.add("cat");                         // s: pig, cat
s.add(1,"dog");                                     // s: pig, dog, cat
s.add(1,"cat");                                     // s: pig, cat, dog, cat
boolean b = s.remove("cat");                        // s: pig, dog, cat
System.out.print(b);                                // true appears
s.remove(1);                                        // s: pig, cat
```

Example 1: longest words

The following program reads the words in a text file, and prints the longest words in the file in the order of their first occurrence. For example, if the input file contains

```
The moving finger writes and having writ moves on
```

then the output will be

```
[moving, finger, writes, having]
```

because the longest word in the file has length 6, and the output consists of all the words of length 6 in the order of their first occurrence. No word occurs twice in the output. The square brackets in the output aren't required but are a by-product of our use of array lists. The name of the text file is passed as a command-line argument.

```
import java.util.*;
import java.io.*;
class LongWords {
```

```
    public static void main(String[] args) {
        Scanner file = null;
        try {
            file = new Scanner(new File(args[0]));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
        ArrayList<String> longs = new ArrayList<String>();  // list of longest words
        int len = 0; // all words in longs have length len
        while (file.hasNext()) {
            String w = file.next(); // next word in input
            if (w.length()>len) { //w is new longest word
                longs.clear();  longs.add(w);  //  longs contains only w
                len = w.length();
            }
            else if (w.length() == len) { // w's length is same as those in longs
                if (!longs.contains(w)) // a first occurrence of w
                    longs.add(w);
            }
        }
        System.out.print(longs);
    }
}
```

■

It is clear that it is much simpler to locate, insert, or delete an element in an array list than in an array. Remember, however, that this simplicity is only in the writing of the code. Just as much work has to be done by the machine behind the scenes – it's just that the coding has been done for us. In particular, all but the final two methods in the list at the start of this section require the array list to be scanned, and so the amount of work done by the machine in each case is proportional to the length of the array list.

There is an additional cost for array lists whose elements are of a primitive type. Elementary values have to be wrapped up as objects, although this is usually accomplished by automatic boxing and unboxing.

Although array lists are simpler to use than arrays, collections of limited size which are subject to only simple operations may be more easily and cheaply managed using arrays, especially when the items are elementary values. When performance is not critical, and particularly when the elements to be stored are not elementary, array lists will usually be preferred.

3 LinkedList

Apart from arrays and array lists, Java provides another class for handling lists, namely `LinkedList`. An instance of `LinkedList` is called a *linked list*. The constructors for `LinkedList<T>` are

LinkedList() LinkedList(*Collection*<*T*>)

They behave just like the corresponding constructors for `ArrayList`. All the methods of `ArrayList` are also provided by `LinkedList`. Indeed, if in a program you replace all occurrences of `ArrayList` with `LinkedList` the program will still work as before. The difference lies in the underlying implementations. Linked lists are not implemented using arrays, with the consequence that indexing into linked lists is expensive. So operations such as `t.get(i)` and `t.set(i,o)` are best avoided when `t` is a linked list. On the other hand, insertions and deletions to/from linked lists are more efficient than for array lists. Many applications require us to maintain lists which grow and shrink by adding or deleting items at either end of the list, and which do not require indexing into the list. For these applications, linked lists are to be preferred over array lists.

`LinkedList` (but not `ArrayList`) includes some methods for operating on either end of a list. These bring some added convenience but no added power.

void addFirst(*T*)
T getFirst()
T getLast()
T removeFirst()
T removeLast()

(As usual, `T` stands for the type of the elements in the list.) **`t.addFirst(o)`** inserts `o` at the beginning of list `t`. (Recall that `t.add(o)` appends `o` to the end of `t`.) **`t.getFirst()`** returns a reference to the first element in list `t`. `t` must not be empty. **`t.getLast()`** returns a reference to the last element in list `t`. `t` must not be empty. **`t.removeFirst()`** removes the first element from list `t` and returns a reference to it. `t` must not be empty. **`t.removeLast()`** removes the last element from list `t` and returns a reference to it. `t` must not be empty.

The following code illustrates the behaviour of the above methods.

```
LinkedList<String> s = new LinkedList<String>();  // s is empty
s.add("dog"); s.add("cat");                        // s: dog, cat
s.addFirst("pig");                                 // s: pig, dog, cat
s.addFirst("cow");                                 // s: cow, pig, dog, cat
System.out.print(s.getLast());                     // cat appears
s.removeFirst();                                   // s: pig, dog, cat
String word1 = s.removeLast();                     // s: pig, dog
System.out.print(word1);                           // cat appears
```

We can access all the elements in an array list in turn by indexing over the array, from 0 up to the list size less 1. However, indexing into linked lists is expensive, and it is preferable to use a for-each loop.

Example 1: reverse a list of integers

The following program reads a sequence of integers from the keyboard, and displays them in reverse order.

```
import java.util.*;
class ReverseInts {
   public static void main(String[] args) {
      LinkedList<Integer> nums = new LinkedList<Integer>(); // for numbers in input
      while (!Console.endOfFile()) {
         int k = Console.readInt();
         nums.addFirst(k); //using autoboxing
      }
      for (Integer k: nums)  {
         System.out.print(k + "  ");
      }
   }
}
```

■

A *stack* is a list of items to which elements are added and from which elements are removed at one end only. A stack is also called a *last-in-first-out* or *LIFO* list. Some languages provide specially for them. In the case of Java, `LinkedList` provide all the functionality needed. A *queue* is a list to which items are added at one end and from which items are removed at the other end. Items are never inserted are removed at any other position. A queue is also called a *first-in-first-out* or *FIFO* list. Queues occur commonly in some application areas. For example, a program controlling a printer shared by many users must keep a list of jobs waiting to be printed. It is only fair that when the printer becomes available, the job that has been waiting the longest should be the first to print. The list of waiting jobs in the controlling program will therefore be a queue. Some languages provide specially for queues, but in Java the functionality required is provided by `LinkedList`.

There's a big banana skin lurking whenever we use collections such as array lists or linked lists. Inserting an object in a collection is effected by aliasing. In other words, a *reference* to the original object is inserted in the collection, rather than copying the object and inserting a reference to the copy. This means that if you insert an object in a list (or any other kind of collection), and then subsequently in another part of the program alter its state, the change is also effected in the object as stored in the list. This may well be what you want. However, if you intend to change an object after inserting it in a list, and you want the original object to remain unchanged in the list, you should make a copy of the object and insert the copy. Note

that constructors with collection arguments always introduce aliasing. For example, `ArrayList<Integer> w = new ArrayList<Integer>(t)` results in the creation of a new list `w`, but the elements in `w` and `t` are shared via aliasing. However, `w` and `t` are themselves distinct; for example, appending a new element to `t` has no effect on `w`.

We mention some additional methods provided by both list classes (i.e. `ArrayList` and `LinkedList`). They are not used so much (and they are expensive) so it suffices to pass lightly over them. We will not be using them.

boolean addAll(*Collection*<*T*>)
boolean retainAll(*Collection*<*?*>)
boolean removeAll(*Collection*<*?*>)
boolean containsAll(*Collection*<*?*>)

As before, the question mark in `<?>` stands for an arbitrary type, although in practice it will always be the same as `T`. **`t.addAll(c)`** appends all the elements in collection `c` to the end of list `t`. When `c` is of type `LinkedList` or `ArrayList`, the elements are appended in the same order as they occur in `c`. When `c` is of type `HashSet`, the order in which the elements are appended is not specified. When `c` is of type `TreeSet`, the elements are appended in ascending order. A boolean is returned indicating whether `t` changed as a result of the call. **`t.retainAll(c)`** removes from list `t` every element not in collection `c`. A boolean is returned indicating whether `t` changed as a result of the call. The method is more expensive for `t` of type `ArrayList`. **`t.removeAll(c)`** removes from list `t` all the elements in collection `c`. A boolean is returned indicating whether `t` changed as a result of the call. The method is more expensive for `t` of type `ArrayList`. **`t.containsAll(c)`** returns a boolean indicating whether list `t` contains all the elements in collection `c`.

4 List utilities

The collections classes includes a class **`Collections`** which provides some useful static methods on lists, including the following:

static void sort(*List*<*E*>)
static int binarySearch(*List*<*E*>, *E*)
static void shuffle(*List*<*?*>)

We write `List` above to indicate that the actual parameter may be an instance of `LinkedList` or `ArrayList`. `E` is a type parameter. **`Collections.sort(t)`** sorts list `t` into ascending order; elements are compared using `compareTo()` which must be provided by class `E`. **`Collections.binarySearch(t,o)`** returns an index of `o` in `t`, or –1 if `o` is not present. Elements are compared using `compareTo()` which must be provided by class `E`. List `t` must be sorted. It is not recommended for `t` an instance of `LinkedList`. `binarySearch(t,o)` behaves like `t.indexOf(o)` except it does not guarantee to locate the first occurrence of `o`; in the case of `ArrayList` it is more efficient than `t.index(o)`. **`Collections.shuffle(t)`** rearranges the elements in list `t`

in some random way.

The following code illustrates these methods.

```
ArrayList<String> s = new ArrayList<String> ();   // s is empty
s.add("dog"); s.add("hen");                       // s: dog, hen
s.add("cat"); s.add("hen");                       // s: dog, hen, cat, hen
System.out.print(s);                              // [dog, hen, cat, hen] appears
Collections.sort(s);                              // s: cat, dog, hen, hen
int k = Collections.binarySearch(s,"dog");        // k: 1
Collections.shuffle(s);                           // s (e.g.): hen, dog, hen, cat
```

Example 1: word frequency count

Literary scholars are often interested in knowing how frequently an author uses a particular word. The program `FrequencyCount` below is for them. For example, a Shakespearean scholar wishing to investigate word frequencies in *The Tempes*t would invoke

```
java FrequencyCount TheTempest.txt
```

where `TheTempest.txt` contains the text of *The Tempest* (you can find Shakespearean plays and many other literary works on the web). `FrequencyCount` accepts words from the keyboard and for each word reports the number of times the word occurs in `TheTempest.txt` (or whatever text file is supplied in the command line). A typical interaction is:

```
Antonio
Frequency of antonio: 70
prospero
Frequency of prospero: 139
exeunt
Frequency of exeunt: 14
```

```
import java.util.*;
import java.io.*;
class FrequencyCount {
    public static void main(String[] args) {
        // Phase 1: generate a sorted list of words in the input file
        ArrayList<String> words = new ArrayList<String>(); // for words in input file
        Scanner text = null;
        try {
            text = new Scanner(new File(args[0]));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
        while (text.hasNext()) {
```

```
            String w = text.next();  // next word
            // convert w to lower case, & strip any trailing punctuation mark
            w = w.toLowerCase();
            if (!Character.isLetterOrDigit(w.charAt(w.length()-1)))
                w = w.substring(0, w.length()-1);
            words.add(w);
        }
        Collections.sort(words);
        // Phase 2: process user's queries
        System.out.println("Enter words: ");
        while (!Console.endOfFile()) {
            String w = Console.readString();  // query word
            w = w.toLowerCase();
            int count = 0; // number of occurrences of w
            int i = words.indexOf(w); // look up w in words
            if (i>=0) { // w occurs in words
                // count number of occurrences of w in words
                while(i<words.size() && w.equals(words.get(i))) {
                    count++;
                    i++;
                }
            }
            System.out.println("Frequency of " + w + ": " + count);
        }
    }
}
```

The program consists of two phases. The program does not interrogate the file each time the user enters a word, because searching large files is slow. Instead the program reads the file at the outset, builds a list of the words it contains, and sorts the list. All this happens in the first phase. Recall that `readToken()` reads *tokens*, and tokens are almost but not quite the same as words. Tokens may end in a punctuation mark such as a comma or period, and these have to be removed. The words are stored in lower case. The second phase handles the user's queries. A word is looked up by locating the index of its first occurrence in the list. Any subsequent occurrences must immediately follow in the list because the list is sorted. Because we are indexing in the list, a list of type `ArrayList` is computationally cheaper than one of type `LinkedList`. An alternative and neater solution will be developed when we discuss maps.

5 Equality and comparisons

Many of the methods in the list classes compare elements using `equals()`. Although `equals()` comes for free with all classes, it compares objects based on references rather than contents. If an `equals()` based on object contents is not provided, then some methods in the collection classes will not behave as expected. Suppose class `Person` is defined as follows:

```
class Person {
    private String name;
    private int age;

    Person (String s, int years) {
        name = s; age = years;
    }

    .....
}
```

and suppose some client code includes the following:

```
LinkedList<Person> list = new LinkedList<Person>();
list.add(new Person("Bill", 23));
System.out.println(list.contains(new Person("Bill", 23)));
```

We might expect to see `true` displayed when the above code executes, but in fact `false` appears unless class `Person` defines `equals()` based on object contents:

```
public boolean equals(Object p) {
    if (p==null) return false;
    Person other = (Person) p;
    return name.equals(other.name) && age==other.age;
}
```

Both `Collections.sort(t)` and `Collections.binarySearch(t,o)` assume that the constituent elements come equipped with a comparison method `compareTo()` that behaves like `compareTo()` for strings. An invocation of `p.compareTo(q)` returns (i) a negative integer if `p` precedes `q` in the ordering we have in mind; (ii) a positive integer if `q` precedes `p`; and (iii) 0 if `p` and `q` are equal. Below is a `compareTo()` method for class `Person`:

```
public int compareTo(Object obj) {
    Person other = (Person) obj;
    if (name.equals(other.name)) {
        if (age==other.age) return 0;
        else if (age<other.age) return -1;
        else return 1;
    }
    else return name.compareTo(other.name);
}
```

Method `compareTo()` compares `Person` objects alphabetically by name, with any tie being

resolved by comparing ages. A class with `compareTo()` must include the phrase `implements Comparable` after the class name:

```
class Person implements Comparable { … }
```

If you omit this, the system will fail to find `compareTo()` (You may or may not recognise `Comparable` as an interface, but you need no understanding of interfaces for the present purposes). The header of `compareTo()` must be precisely `public int compareTo(Object obj)` (the name of the parameter can be any identifier, however). The argument supplied for `obj` will be another object of the class (here `Person`), but Java nevertheless insists that the type of `obj` be `Object`, and so a type cast is needed in the body. It is common practice to return –1 for *less than* and 1 for *greater than*, but this is not required. For example, the following alternative version of `compareTo()` for `Person` acceptable:

```
public int compareTo(Object obj) {
    Person other = (Person) obj;
    if (name.equals(other.name))
        return age-other.age;
    else return name.compareTo(other.name);
}
```

You must define `compareTo()` so that it really is a comparison operation – just returning 257, say, for every invocation will not give you meaningful results. If you proceed sensibly you will not meet any difficulties. If you are in doubt, check that your definition satisfies the following requirements (the *sign* of a number is –1, 0, or 1 according to whether it is negative, zero, or positive, respectively):

(i) The sign of `p.compareTo(q)` equals `-1 *` the sign of `q.compareTo(p)`.

(ii) If `p.compareTo(q)>0` and `q.compareTo(r)>0` then `p.compareTo(r)>0`.

(iii) If `p.compareTo(q)` equals `0` then `p.compareTo(r)` and `q.compareTo(r)` have the same sign.

If objects of the class are also subject to equality testing, `equals()` and `compareTo()` must be consistent in the sense:

(iv) `p.equals(q)` yields `true` if and only if `p.compareTo(q)` yields `0`.

The primitive wrapper classes all provide a meaningful `compareTo()`.