28 Lists using ArrayList

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 **`T`** of its elements, i.e. an array list whose elements are each of type `T` has type **`ArrayList<T>`**. For example, an array list of strings has type `ArrayList<String>`. Throughout, we will use `T` as the type parameter. The constructors are as follows:

```
ArrayList<T>()          ArrayList<T>(ArrayList<T>)
```

`new ArrayList<T>()` creates an empty list, and **`new ArrayList<T>(c)`** creates a list with the same elements as `c` in the same order (copies are not made of the items in `c`, rather the items are referenced by both `c` and the newly created list). Examples include (where `list` is of type `ArrayList<Double>`)

```
ArrayList<String> ws = new ArrayList<String>();
ArrayList<Integer> ns = new ArrayList<Integer>();
ArrayList<Double> ds = new ArrayList<Double>(list);
```

It is permissible to omit the type in the constructor when it can be inferred. For example, the preceding examples can be written as

```
ArrayList<String> ws = new ArrayList<>();
ArrayList<Integer> ns = new ArrayList<>();
ArrayList<Double> ds = new ArrayList<>(list);
```

Array lists have indices just like arrays, i.e. the indices run from 0 up. The most fundamental methods supplied by `ArrayList<T>` 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. 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 given text. A word is defined to be "big" if its length exceeds the average length of *all* the words in the text. The text is supplied through the standard input, typically as a text file via redirection. 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.