

CS1Ah Lecture Note 17

Case Study: Using Arrays

This note has three main aims:

1. To illustrate the use of arrays introduced in Lecture Note 16, on a slightly larger example.
2. To introduce the idea of using arrays to maintain a collection of similar objects and to allow us to search the collection for an object that meets some criterion.
3. To begin considering the running times of programs and how this might influence our choice of implementation in a particular situation.

17.1 Maintaining a collection of student objects

In Lecture Note 9 we saw how to use objects to model a student. In any real record keeping situation we would be dealing with a collection of `Student` objects, one per student in the class. Typically we would expect to be able to:

- **add** students to the collection,
- **delete** students from the collection,
- **modify** the information for a particular student in the collection, and
- **search** the collection to find students who match particular criteria.

In this note we consider implementing a subset of these functions, namely the ability to add a new student to the collection and to search for a student with a particular surname. This is sufficient to illustrate some interesting variations in approach. More generally, the issue of the efficient organisation of data storage and retrieval has been the subject of intensive study in Computer Science since its inception (see the chapters on Data Models in Aho and Ullman for a taste of this work.) The results of this work have found widespread application and are in common use in most computer systems.

We will study two different ways of using arrays to maintain a collection of `Student` objects. At the risk of some confusion, we call a collection of students a *class*, so the object which represents the collection is a `ClassofStudent` object. Both mechanisms are based on using a large array, which gradually fills up as we add more students. The second mechanism maintains an order on the `Student` objects in the array to speed up searching.

17.2 A simple implementation

In both the implementations we consider here we use an array of a fixed size to provide the means to store up to that number of `Student` objects. Figure 17.1 illustrates the approach by showing a snapshot of a collection at some point during the execution of a program:

- `N` is the size of the array, this is the maximum number of objects we can keep in the collection.
- `number` is the count of the number of `Student` objects we have in the collection at the moment.
- The `Student` objects we have in the collection are stored in an array at indices 0 to `number-1`
- The indices `number` up to `N-1` are free space that may be used later in the execution of the program.
- When `number` is zero then there are no students in the collection and all the cells of the array are free space.

The class definition for `ClassOfStudents1` illustrates one approach to using this arrangement to store `Student` objects:

```

1  class ClassOfStudents1 {
2      private static final int N = 1000;
3      private Student[] classMembers;
4      private int number;
5
6      // Constructor: make array, set number to zero.
7      public ClassOfStudents1() {
8          classMembers = new Student[N];
9          number = 0;
10     }

```

This part of the class definition shows how we implement the structure described in Figure 17.1 in Java. Notice that we make the attributes `private` to ensure they can only be changed in a consistent manner by the methods we define.

To insert an object into this collection, the simplest approach is just to add the new object in the next available space. The method `insert` does this, and returns a boolean value to indicate success or failure:

```

12     public boolean insert(Student newStudent) {
13         if (number==N)
14             return false;
15         classMembers[number++] = newStudent;
16         return true;
17     }

```

The simplest approach to looking for a particular `Student` object whose surname is equal to a given string is to compare the surname of each object in the collection to the given object, returning when a match is found. If there is no match then `null` is returned, indicating no matching student can be found.

```

19     public Student find(String findsurname) {
20         for (int i = 0; i < number; i++) {
21             if (classMembers[i].surname.equals(findsurname)) {
22                 return classMembers[i];
23             }
24         }
25         return null;
26     }
27 }

```

Figure 17.2 illustrates the situation as the method `find` does its work. Those objects with indices less than `i` have been eliminated as possible results. The loop in the program increases `i` on each iteration.

Informally we might want to consider how long it takes each of these methods to do their work. Looking at the code we can see that adding an item takes a constant amount of time because there are no loops in the code. By contrast we can see that in the worst case (this arises when there is no `Student` object with a matching surname) the `find` method can take time proportional to the number of objects in the collection to do its work. We can see that adding is fast while searching is slower. This approach might be acceptable if we are using the program in a situation where we have to add many objects but we search relatively infrequently. In the situation of a student record system this is highly unlikely. We are far more likely add relatively few objects (a few hundred) and then search many times. The approach considered in this section does not work well for this application. We need to make searching faster. We consider how to this next.

17.3 An implementation using order

A widely used approach to speeding up data access is to use the properties of order. If our dictionaries were organised as a list of words in random order they would be useless. Instead because they are organised in dictionary order we can use properties of that order to search the dictionary efficiently. In this section we consider how to use order to make our searching method go faster.

First we need to consider how to maintain the collection of `Student` objects in dictionary order of surname. To do this necessitates changing the `insert` method so that it inserts objects in the right place. The code for the method is:

```

12     public boolean insert(Student newStudent){
13         if (number==N)
14             return false;
15         int i;
16         for(i = number++; i>0; i--) {

```

```

17     int comparison =
18     classMembers[i-1].surname.compareTo(newStudent.surname);
19     if (comparison>0)
20         classMembers[i] = classMembers[i-1];
21     else
22         break;
23 }
24 classMembers[i] = newStudent;
25 return true;
26 }

```

This code is best explained by considering Figure 17.3. In this revised version of the method we begin inserting from the last object in the collection and if the surname of the object to be inserted is smaller in dictionary order than the last element we move the last element one to the right and consider the next largest element and so on. By moving objects to the right as we search for the place to insert the object we ensure there is a free space in the array at the correct place in the array and we can insert the new object there.

The time taken to do this in the worst case is proportional to the number of objects in the collection (this arises when we want to insert a student whose surname is smaller than all the students in the collection). So in this case adding an object is much harder work than our first method. But as we will see there are benefits when it comes to doing searches.

Figure 17.4 illustrates the approach taken to searching an array whose elements are ordered by surname. The approach is usually called *binary search* because we halve the size of the search space each time we look at one element of the array. The method works as follows:

- We know the object we are looking for lies between indices i and $j-1$. Initially we can set i to zero and j to number; the object could be any one in the collection.
- We then consider the object midway between i and j . If the surname of that object is less than the one we are looking for then we can eliminate all objects in the array up to and including the midway object. If the surname of the object is greater than the one we are looking for then we can eliminate all those objects between the midway object and the end of the array. If there is a match of surname we return the matching object.
- Eventually we either find a match or i is bigger than or equal to j and then we know there is no student with the required surname.

The implementation of this method in Java looks like this:

```

28 public Student find(String surname) {
29     int i = 0, j = number, mid;
30
31     while (i < j) {
32         mid = (i+j)/2;    // NB: "/" takes integral part
33         int comparison =

```

```

34         classMembers[mid].surname.compareTo(surname);
35     if (comparison<0) {
36         i = ++mid;
37     } else if (comparison>0) {
38         j = mid;
39     } else {
40         return classMembers[mid];
41     }
42 }
43 return null;

```

As we have done previously we might ask ourselves how much effort is needed to find out whether a given surname has a matching student? It is certainly much less effort than the `find` method in `ClassOfStudents1`. For example, the new version of `find` will examine at most 10 objects if the original collection contains around 1000 objects. This second implementation is clearly much better than the first for the application we are considering. The cost of adding an element is quite high but searching is much less costly. This matches the likely pattern of use of the system much better than the implementation in `ClassOfStudents1`.

17.4 A simple test program

The code below implements a single, simple test for the `ClassOfStudents2` implementation. It is often worth implementing such simple tests, even for code which seems “obviously” correct. A more rigorous approach is to build up a larger *test suite* of multiple tests. The test suite can then be re-executed whenever the implementation is changed.

```

1  class Test2 {
2      public static void main(String[] args) {
3          Student paul = new Student();
4          Student david = new Student();
5          Student murray = new Student();
6          ClassOfStudents2 cs1 = new ClassOfStudents2();
7
8          paul.surname = "Jackson";    paul.forename = "Paul";
9          david.surname = "Aspinall";  david.forename = "David";
10         murray.surname = "Cole";     murray.forename = "Murray";
11
12         cs1.insert(paul);
13         cs1.insert(david);
14         cs1.insert(murray);
15
16         System.out.println("Forename: " +
17                             cs1.find("Jackson").forename);
18     }

```

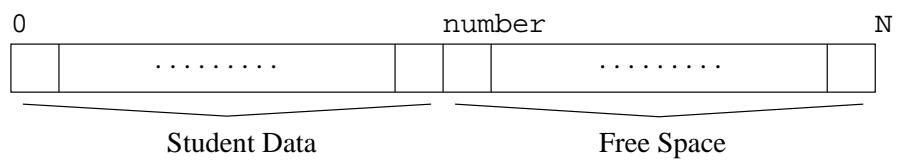
You might also find it instructive to try out the *BlueJ* Integrated Development Environment on this example. See the appendix to lecture note 3 for information on BlueJ.

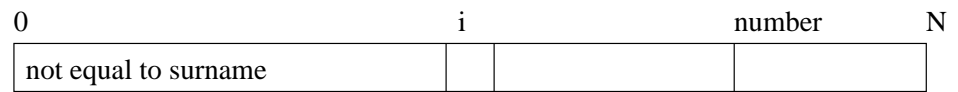
While testing a class implementation interactively with is well worthwhile, it is no substitute for automated tests that can be rerun repeatedly and incrementally extended throughout a software development project.

17.5 Summary

- We can use arrays in Java to implement collections of similar objects.
- We have choice over how to use arrays in that representation.
- Ordering items by the value we will search on can speed up searching.
- The choice of a particular implementation depends on the context in which the system will be used.

*Authored by Stuart Anderson and David Aspinall.
Paul Jackson, 19th November 2002.*







0	i	j	number	N
less than	possibly equal		greater than	