6 Arrays: Searching and sorting

1 One-dimensional Arrays

Arrays are a fundamental component of programming. We will revise some aspects of them here, but you are expected to have a good understanding of the basics from course CA165.

We make a small example to refresh your memory. We want to count the frequency of word-lengths in a piece of text. An example of output is:

```
1-letter words: 3
2-letter words: 5
3-letter words: 9
4-letter words: 4
6-letter words: 1
```

```
class WordCount {

    public static void main(String[] args) {
       int[] count = new int[20]; // word lengths up to 19, say
       // zero counts
       int i = 0;
       while (i<count.length) {
          count[i] = 0; i++;
       }
       // Read words
       while (! Console.endOfFile()) {
          String word = Console.readToken();
          count[word.length()]++;
       }
```

```
        // Print result
        i = 1;
        while (i<count.length) {
           if (count[i]>0)
              System.out.println(i + "-letter words: " + count[i]);
           i++;
        }
     }
}
```

The program uses an array of 20 components of type integer. Component i records the number of words of length i (component 0 is not used). For example, count[4] is intialised to 0, because initially we have seen no words of length 4 (or any other length). Each word is read into string variable *word*, and hence the word's length is word.length(). If this is 4, say, then we must increment word[4]. In general, we update the relevant count by executing the command

```
count[word.length()]++;
```

Note that the length operator for strings has a pair of brackets, unlike that for arrays. Funny that!

For-each loops

For-each loops provide a convenient way to process all the constituent values in an array. For example, suppose a program contains the integer array `myArray`;

```
int[]  myArray = { 23, 34, 65, 65, 19, 29, 34, 11, 85};
```

The values in myArray are printed using the following for-each loop:

```
for (int k: myArray)
     System.out.println(k);
```

We could have also expressed this a for-loop as follows, although it is a little more cumbersome:

```
for (int i=0; i<myArray.length; i++)
     System.out.println(myArray[i]);
```

The for-loop is in turn just a shorthand for the following while-loop

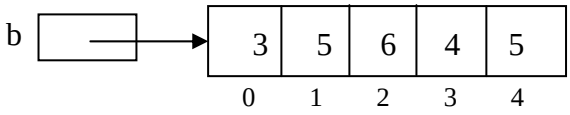
```
int i = 0;
while (i<myArray.length) {
     System.out.println(myArray[i]);
     i++;
}
```

While loops are completely general, for-loops are more restrictive but a little more convenient when, and for-each loops are again more convenient but are even more limited in their applicability. For example, it is not possible to use a for-each loop to assign 0 to every element of an integer array, or to print out the values in the first half of an array. None of the loops in the preceding program can be written using a for-each loop (in the case of the final loop, a for-each loop would not give access to the index `i`).

2 Arrays as objects

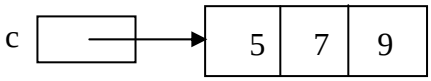
Java treats arrays as objects. Hence an array variable does not literally contain an array, but a reference to one. For example, the declaration below can be envisaged as shown:

```
int[] b = {3, 5, 6, 4, 5};
```



If we declare an array without initialising it, it is given the default value *null*. Java effects assignment of arrays by copying references, not by copying the array. For example, suppose we have if addition to the declaration of b above, array c declared as follows:

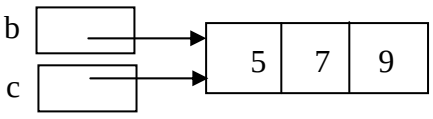
```
int[] c = {5, 7, 9};
```



Then the assignment

```
b = c;
```

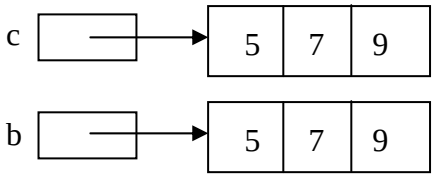
gives rise to the following state of affairs:



Any subsequent change to array c will be also in effect be a change to b. If you do not want this, then the preceding assignment should be replaced with an element-by-element copy:

```
b = new int[c.length];
for (int i=0; i<b.length; i++) b[i] = c[i];
```

Now b and c will refer to distinct arrays, and an assignment to one will not affect the other:



However, simple assignment of references is nearly always sufficient.

Because arrays are treated as objects, array parameters contain *references* to arrays. This is good news in that a method may change the contents of an array argument by assigning to the formal parameter. The following code illustrates this:

```
static void incAll(int[] b) {
        for (int i=0; i<b.length; i++) b[i]++;
}
.....
int[] c = {5, 7, 9};
incAll(c);
System.out.println(c[0] + " " + c[1] + " " + c[2]);
```

When the above program is executed it displays 6 8 10 (not 5 7 9). If you do not understand this, refer back to the section on object parameters.

Methods may return arrays, although it is not a very common occurrence in programming. Of course an array-valued method returns a reference to an array, rather than the array itself. The following is a trivial example of an array-valued method:

```
static int[] get3() {
        int[] b = new int[3];
        for (int i=0; i<3; i++)
                b[i] = Console.readInt();
        return b;
}
```

```
.....
int[] c; // note: no "= new int[..]"!
c = get3();
System.out.println(c[0] + " " + c[1] + " " + c[2]);
```

If the input consists of the integers 5 6 7, say, then the program displays 5 6 7. Note that c is not initialised in its declaration, but is assigned a reference to an array object by an invocation of get3().

3 Command line arguments

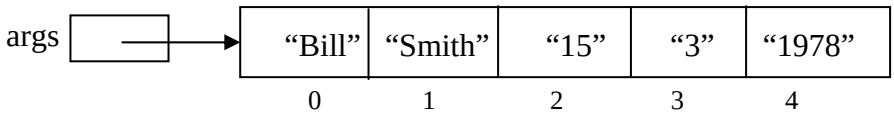
You can pass arguments to a program from the command line. The arguments are formed into an array of strings and passed to the program. For example, we may issue the command

```
java MyProgram Bill Smith 15 3 1978
```

where class MyProgram has method main() with header:

```
public static void main(String[] args)
```

This has the effect of initialising args thus



after which the code in main() is executed. Note that even integers are passed in String form. If you want to process an argument passed in args[] as an integer, you will have to convert it using Integer.parseInt(), and similarly for booleans, doubles, etc. We illustrate with a program which sums a list of integers supplied at the command line. The following are two typical invocations of the program:

```
java AddInts 5  3  6
java AddInts 7  13  -4  10  13
```

In the first case, the program should produce the output 14.

```
class AddInts {
   public static void main(String[] args) {
      int sum = 0;
      for (int i=0; i<args.length; i++) {
         sum = sum + Integer.parseInt(args[i]);
      }
      System.out.println(sum);
   }
}
```

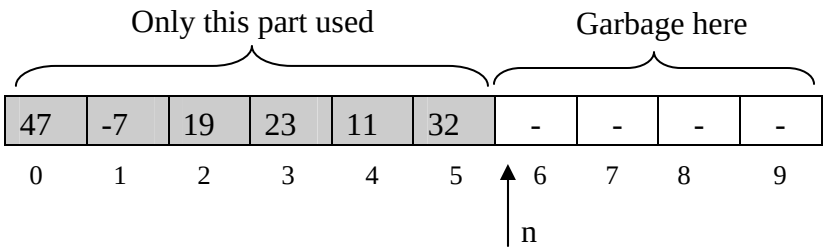
Remember that command line arguments are supplied as arguments to an array parameter in main() (the parameter usually called args), and so we do not employ input statements such as Console.readString() to acquire them. Input statements are only used when the input is supplied after the program has been started.

4 Partially filled arrays

Consider the following programming problem: Suppose we want to write a program which reads a list of integers and displays it in reverse order. Obviously we will save the integers in an array until we are ready to print them, but how big should we make the array? The problem is that don't know the ideal array size until after we have read in all the integers! This is a frequently occurring difficulty in programming, and the solution is important. (i) We choose some maximum size for the array. (ii) We keep the array partially unused at the right-hand end. (iii) If the array turns out to be too small, we typically abort the program. (iv) We employ an ancillary "count variable" to keep track of the number of elements in the array. The following is a typical declaration of an array for storing a list of integers of unknown size, together with the count variable:

```
int[] b = new int[10]; // allow for a max of 10 integers
int n = 0;
```

The count variable n is initially zero because b is empty. After 6 values have been read, say, b and n can be pictured thus:



Variable n has value 6, the values read are in the shaded part of b, and the rightmost 4 components of b contain garbage. Observe that the next available component in b is b[n], and that n must be incremented as each successive value is inserted.

The program to reverse the input follows. We have imposed an arbitrary upper limit of 100 on the number of values in the input list. In the code, we make use of the statement `System.exit(0)` which causes the program to terminate (we could alternatively invoke `return`).

```
class RevInts {
   public static void main(String[] args) {
      int[] b = new int[100]; // allow for a max of 100 integers
      int n = 0; // first n elements of b significant
      //Read list of numbers
      while (!Console.endOfFile()) {
         if (n==b.length) {
            System.out.println("Too many values");
            System.exit(0); //abort
         }
         b[n] = Console.readInt();
         n++;
      }
      // Print in reverse order
      int i = n; // i = no. of values remaining to be printed
      while (i!=0) {
         i--;
         System.out.print(b[i] + " ");
      }
   }
}
```

Observe that we write b.length even though we know its value is 100. The advantage is that if we subsequently decide to change the size of b, only the declaration of b needs to be changed.

When an array parameter is allowed to be partially-filled, its count variable must also be a parameter – otherwise the method will not know which elements are significant and which are garbage. We illustrate this with an alternative coding of the preceding program:

```
class RevInts {

   static void putRev(int[] w, int k) { // print first k elements of w in reverse
      int i = k; // i = no. of values remaining to be printed
      while (i!=0) {
         i--;
         System.out.print(w[i] + " ");
      }
   }

   public static void main(String[] args) {
      int[] b = new int[100]; // allow for a max of 100 integers
      int n = 0; // first n elements of b significant
      //Read list of numbers
         .... as previously ...
      // Print in reverse order
      putRev(b, n); // print first n elements of b in reverse order
   }
}
```

The no-count-variable trap

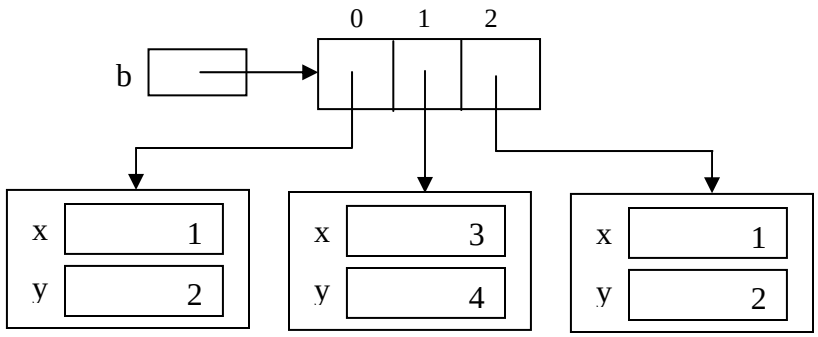
Let variable b be an array of integers that may be partially filled. *You must use an associated count variable*. If you don't, you're writing a program that can't work. Let n be the count variable associated with b. You should be careful to distinguish between the capacity of b, which is b.length, and the current size of b, which is n (n varies of course as the computation progresses). Always, the first n elements of b are significant (i.e. we care about their values), and the remaining b.length-n elements are garbage. Many beginners fail to understand that the garbage values at the end of the array are just that – garbage. We do not care about them, we do not inspect or print them, and they play no part in the computation. Some beginners pointlessly (and misleadingly)

initialise all the elements of the array (typically to 0 or –1, although all values are equally without merit). Although initialising the array elements to 0 is merely pointless, it is downright erroneous to believe that you can dispense with the count variable by testing array elements for 0 (or –1, or whatever initial value was used). It is erroneous, because significant values may well be 0. You *must* use a count variable. If you write a method which processes a partially-filled array supplied as a parameter, then the parameters must also include an associated count variable.

5 Arrays of objects

Arrays of objects are allowed (and are common). They are illustrated in the following code:

```
class Point {
    int x, y;

    Point(int u, int v) {  // constructor
            x = u; y = v;
    }

}
...
Point[] b = new Point[3]; // create the array
b[0] = new Point(1, 2);  b[1] = new Point(3, 4);  b[2] = new Point(1,2);
```



It is easy to forget that you have to create both the array (as in `new Point[3]`), and each constituent object (as in `new Point(1,2)`). The syntax of `new Point[3]` and `new Point(1,2)` are confusingly similar. Alternatively, we may create and initialise b in one go:

```
Point[] b = {new Point(1, 2), new Point(3, 4), new Point(1, 2)};
```

In this example, b[1] (say) references a Point object whose components are regular variables with names b[1].x (= 3) and b[1].y (= 4), respectively.

Example: dating again

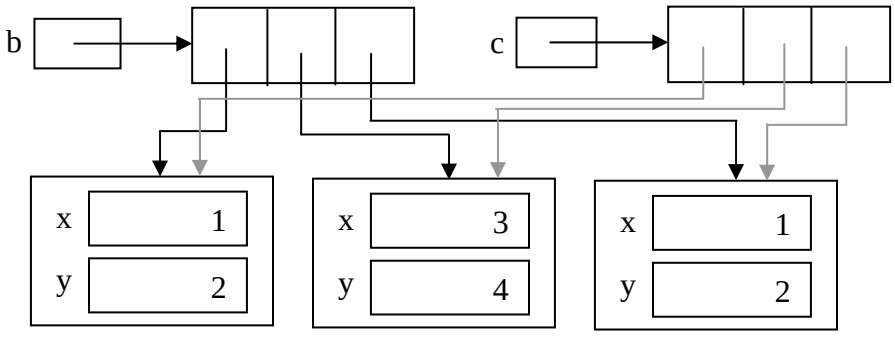
Recall the dating program in which we looked for the most compatible partner for a person. We modify it so that it outputs a list of *all* compatible partners. Refer back to the original solution. We have no need to change classes Date or Person (although method isPreferredTo() is no longer needed). Only main() needs revision:

```
class Dating {
   public static void main(String[] args) {
      Person[] pals = new Person[100]; // max 100 partners catered for
      int n = 0; // number of significant values in pals
      Person subject = new Person(); // subject person
      subject.get();
      while (!Console.endOfFile()) {
         Person p = new Person(); // candidate
         p.get();
         if (p.isCompatible(subject)) {
            if (n<pals.length) {
               pals[n] = p;
               n++;
            }
         }
      }
      if (n==0) System.out.println("Sorry - no partners!");
      else {
         subject.put(); System.out.println(" is partnered with the following:");
         int i;
         for (i=0; i<n; i++) {
            pals[i].put();  System.out.println();
         }
      }
   }

}
```

We have catered for a maximum of 100 partners. If we find more than this, we do not abort the program but simply ignore any new partners. Observe an example of referencing an object via an array component: `pals[i].put()`.

Copying

If you wish to make a fresh copy c of an array b of *object*s it does not suffice to copy each component of b into the corresponding element of c, because that would be to copy references only:



Instead, it is necessary to create a copy of each of b's constituent objects and place a reference to it in the corresponding component in c. However, it is not often that we need to copy an array of objects.

6 Linear search

A common problem in programming is to determine whether a given item occurs among a given collection of items. For example, the core of a spell-checking program is determining whether a given word occurs among a collection of English words. "Linear search" is an algorithm for searching a collection of items when the collection is stored as array. The linear search algorithm is simple: begin at the beginning, and step through each element of the array one by one until the sought-for element is found or the array is exhausted. As an example, we write a program which reads a sequence of words and for each one determines whether it's the name of a day, month, or season.

```
class LinearSearch {

    public static void main(String[] args) {
```

```
        String[] ws =
            {"sunday", "monday", "tuesday", "wednesday",
            "thursday", "friday", "saturday", "january", "february",
            "march", "april", "may", "june", "july", "august", "september",
            "october", "november", "december", "spring", "summer", "autumn",
            "winter"};

        System.out.println("Enter a blank line to terminate.");
        System.out.print("Enter a word: ");
        String s = Console.readString();
        while (s.length()>0) {
           s = s.toLowerCase();
```

```
           // This is a linear search
           int i = 0;
           while (i<ws.length && !s.equals(ws[i])) {
              i++;
           }
           if (i<ws.length) // found
              System.out.println("That's a day, month, or season.");
           else // not found
              System.out.println("That's not a day, month, or season.");
```

```
           System.out.print("Enter a word: ");
           s = Console.readString();
        }
     }
}
```

Observe the use of a common coding trick to handle the case of input words that use a mixture of upper and lower case letters. The words in the table are in lower case, and each word read is converted to lower case before searching for it.

More generally, linear search can be used to find the first element in an array satisfying some property. For example, we may wish to locate the first prime number in an array of integers, or the first four-letter word in an array of strings. The following method uses linear search to determine

the first perfect square (i.e. one of 0, 1, 4, 9, ...) in an array of integers, returning -1 if there are none.

```
static int posSearch(int[] w) {
   int i = 0;
   while (i<w.length && !isSquare(w[i])) {
      i++;
   }
   if (i<ws.length) return w[i];
   else return -1;
}

static boolean isSquare(int n) { // is n square?
   int i = 0;
   while (i*i<n) i++;
   return (i*i==n);
}
```

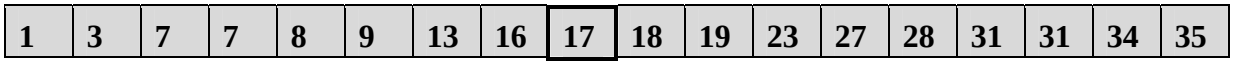
7 Binary search

"Binary search" is also used to determine whether a particular value occurs in an array, but it does so more efficiently than linear search. It is particularly fast when the array being searched is very big. *For binary search, the elements in the array must be in a sorted order*. For an array of integers, this usually means ascending order, and for an array of strings this usually means alphabetic order.

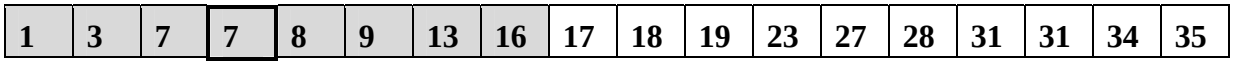
The binary search algorithm works as follows. Suppose we're determining whether x occurs in array w (remember w is sorted). Examine the middle element of w, say w[mid] (for example, if w has 20 (or 21) elements then mid=10). If this is x, we're done. Otherwise, if x is less than the middle element of w, then x occurs to the left of mid if it occurs at all (recall that w is sorted) and so we search the elements with index less than mid. If x exceeds the middle element, we search the elements to the right of mid. Note that in one step, we have either found the element or eliminated one half of the array. We are now left with half the array to search, and we proceed as before: look at the middle element of the half-array -- if it's x we're done, and otherwise, if x is less than ....

The general pattern is that we focus our search on a segment of the array for which we know that if the search element x occurs in the array, then x must be in that segment. In the following

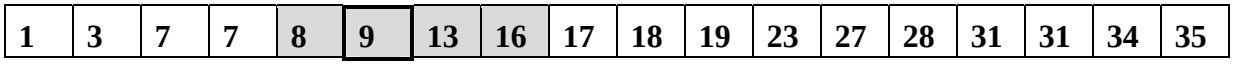
sequence of pictures, we use 13 as the search value, we indicate the search segment by shading, and we indicate the middle cell of the search segment by darker outlining



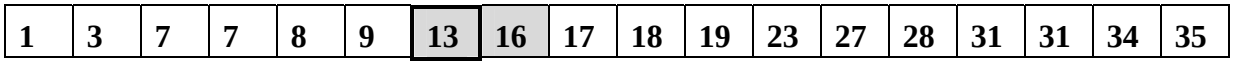
13<17



13>7

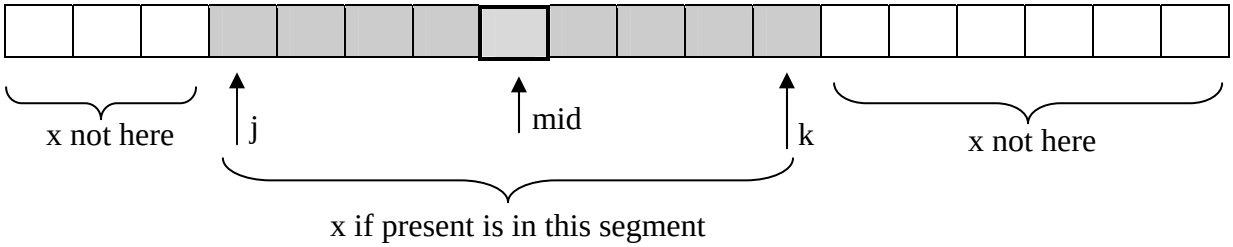


13>9

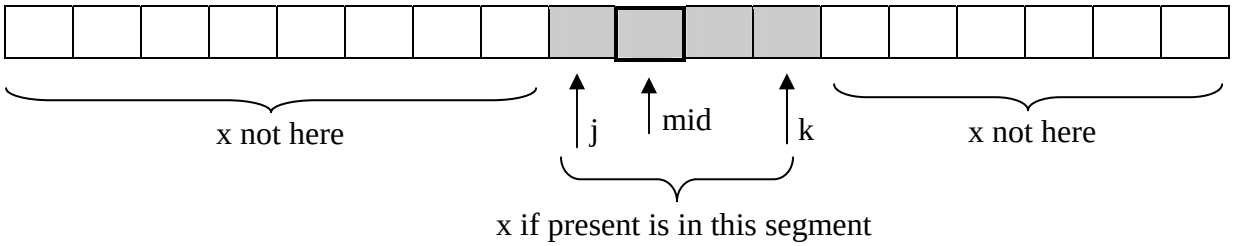


13=13

When coding the binary search algorithm, we use variables – j and k, say – to delimit the current search segment, and a variable – mid, say, – to reference the middle cell in the segment:



If x is greater, say, than the element in the middle cell, then at the next step the picture becomes:



As an example, we re-do the previous program, except that this time we use binary search.

```
class BinarySearch {

    public static void main(String[] args) {
```

```
        String[] ws = {"april", "august", "autumn", "december",
        "february", "friday", "january", "july", "june",
        "march", "may", "monday", "november", "october",
        "saturday", "september", "spring", "summer",  "sunday",
        "thursday", "tuesday", "wednesday", "winter"};
        // Note alphabetical order in array!
        System.out.println("Enter a blank line to terminate.");
        System.out.print("Enter a word: ");
        String s = Console.readString();
        while (s.length()>0) {
           s = s.toLowerCase();
```

```
           // This is a binary search
           int j = 0; int k = ws.length-1; int mid = (j+k)/2;
           // search segment is from index j to k, incl.
           while (j<=k && !s.equals(ws[mid])) {
              if (s.compareTo(ws[mid])<0) // s < middle word
                 k = mid-1; // eliminate top of segment
              else // s > middle word
                 j = mid+1; // eliminate bottom of segment
              mid = (j+k)/2; // ready for next time around
           }
           if (j<=k) // found
              System.out.println("That's a day, month, or season.");
           else  // not found
              System.out.println("That's not a day, month, or season.");
           // end of binary search
```

```
           System.out.print("Enter a word: ");
           s = Console.readString();
        }
     }
}
```

Relative efficiency of binary search

How efficient is binary search compared with linear search? Much more efficient! Suppose we have to search an array of 100,000 elements, and that the search is done 1000 times during the program (say we have to look up a dictionary 1000 times).

Using linear search, we have to examine up to 100,000 elements, say about 50,000 on everage per search. For 1000 searches, we have to examine 50 million elements in total. A machine would check an element in about one millionth of a second, giving a total search time of 50 seconds. Using binary search, the search segments for a single search have lengths (roughly) 100K, 50K, 25K, 12K, 6K, 3K, 1500, 750, 375, 187, 93, 46, 23, 12, 6, 3, 2, 1, 0. Each length corresponds to one comparison, and so there are about 18 comparisons at most per search, and about 18,000 in total, giving a worst case total search time of 18,000 microseconds, or less than one 50th of a second.

Moral: for all but small arrays, prefer binary search if there is a choice. For small arrays, the difference is small and we may prefer to use the easier-to-code linear search. Of course, if the elements cannot conveniently be sorted than we have no choice but to use linear search.

8 Selection sort

To use binary search on an array, the elements must be in sorted order. If not, then we can first re-arrange them; this is known as "sorting" the array. Sorting is probably the most important problem in computing – just about every industrial program will engage in some sorting either for efficiency purposes (to speed up searching, say) or because outputs are expected in sorted order (for example, we expect student class lists in alphabetic order).

There are many sorting algorithms. For the moment, we will study an algorithm known as "selection sort". This is an "in-situ" sort, meaning that we sort by swapping elements in the array, rather than creating a new array. The essential idea is as follows:

1. Find the minimum value in the array, and swap it with the first element. That's the first element of the final sorted array taken care of.
2. Now look for the smallest element in the rest of the array (i.e. from the second element on), and swap it with the second element. That's the second element of the final sorted array taken care of.
3. Now look for the smallest element in the rest of the array (i.e. from the third element on), and

swap it with the third element. That's the third element of the final sorted array taken care of.

4. Now look for the smallest element in the rest of the array (i.e. from the fourth element on), and .....

We illustrate with a small list of numbers (explanation follows):

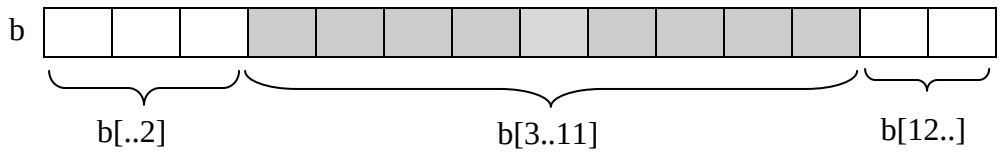
At each step, the segment of the sequence still to be sorted is underlined, and the minimum in the underlined sequence is highlighted. Observe that at each stage the array can be viewed as consisting of a left segment which is sorted, and a right segment (underlined) all of whose values are at least as big as the values in the left segment. Hence, at each step it only remains to sort the right segment.

We use the algorithm in a solution to the problem we solved earlier: determining whether a word belongs to a fixed collection. We will use binary search as before, but this time we will save ourselves the trouble of manually creating the table of words in sorted order. We will let the program do the sorting for us. The outline shape of the program is:

```
class Sorter {

    public static void main(String[] args) {

       String[] ws = {"sunday", "monday", "tuesday", ....., "winter"};
                          // Note not sorted!!!

       // Sort array ws using selection sort
       .....

       ..... " as previously, using binary search" .....
    }
}
```

The code for sorting the array is given below. In explaining the code, we make use of some extra notation for arrays. For b any array, and i and j any integers such that $0 \le i \le j < b.length$,

b[i..j] denotes the segment of b from index i up to index j, inclusive ($0 \le i \le j < b.length$).
b[i..] denotes b[i..b.length-1]
b[..j] denotes b[0..j]
b[i..j] denotes the empty segment if i >j



Remember that this notation is not Java, but just notation we use in commentary.

```
// Sort array ws using selection sort
int lft = 0; // ws[lft..] to be sorted
while (lft<ws.length-1) {
   // find minimum in ws[lft..]
   int min  ...  // Let min index minimum
   .............
   // swap ws[lft] and ws[min]
   String temp = ws[lft]; ws[lft] = ws[min]; ws[min] = temp;
   // advance
   lft++;
}
```

It only remains to supply the code to find the minimum element:

```
// find minimum in ws[lft..]
int min = lft;  int i = lft+1;
// ws[min] is minimum in ws[lft..i-1]
while (i<ws.length) {
   if (ws[i].compareTo(ws[min])<0) {
      min = i;
   }
   i++;
}
// ws[min] is minimum in ws[lft..]
```

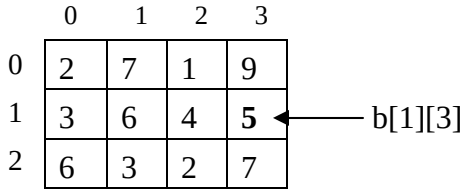
The complete code for sorting ws is:

```
// Sort array ws using selection sort
int lft = 0; // ws[lft..] to be sorted
while (lft<ws.length-1) {
   // find minimum in ws[lft..]
   int min = lft;  int i = lft+1;
```

```
        // ws[min] is minimum in ws[lft..i-1]
        while (i<ws.length) {
            if (ws[i].compareTo(ws[min])<0) {
                min = i;
            }
            i++;
        }
        // ws[min] is minimum in ws[lft..]
        // swap ws[lft] and ws[min]
        String temp = ws[lft]; ws[lft] = ws[min]; ws[min] = temp;
       // advance
        lft++;
    }
```

9 Multidimensional arrays

The arrays we have seen so far are "1-dimensional", by which is meant that a single index suffices to identify any component of the array. A 2-dimensional array can be visualised as a grid with so many rows and so many columns. A 2-dimensional integer array called b, say, with 4 columns and 3 rows can be pictured thus:



Each row and column is indexed from 0, and each component cell is identified by supplying two indices as indicated. We say that an array with m rows and n columns is an "m×n" array; for example, the array above is 3×4. A 2-dimensional array is also called a matrix. Arrays may have more than two dimensions.

In Java, a 3×4 integer array b is created by the following declaration:

```
int[][] b = new int[3][4];
```

As an example, we write a program to generate a magic square, i.e. a square whose rows, columns, and diagonals all have the same sum. An example of a 3×3 magic square is

8	1	6
3	5	7
4	9	2

Verify that each row, column, and diagonal adds up to 15. The following is an algorithm to generate a magic square of order n, i.e. an n×n magic square. The algorithm only works for odd n. Deposit 1, 2, 3, ... up to n^2 in cells of an n×n grid proceeding as follows. The first cell to receive a value is the middle of the top row. The successor cell in each case will be the cell immediately to the north-east (see how 6 follows 5 above). However, if the north-east cell does not exist, we wrap around the borders (see how 3 follows 2, and how 2 follows 1). After each group of n numbers have been deposited, the successor cell is immediately to the south rather than the north-east (see how 4 is to the south of 3). Test your understanding of this on the magic square above. For a further check, generate a magic square of order 5. You should get the following:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Note that the number in each cell must be printed "right-justified" so that each column of numbers is aligned on the right-hand side.

The magic squares program is given below. The command

```
java MagicSquare 5
```

will generate a magic square of order 5 (note that the size of the square is supplied at the command line, and is not read in during program execution).

```
class MagicSquare {
   public static void main(String[] args) {
   // Generate a magic square of order args[0], which
   // must be an odd positive integer
      int n = Integer.parseInt(args[0]);
```

```
      int[][] ms = new int[n][n]; // for the magic square
      int num = 1; // next number to be deposited
      int j = 0; int k = n/2; // next cell to be filled is ms[j][k]
      while (num<=n*n) {
         ms[j][k] = num; // fill in cell
         if (num%n==0) { // after each n steps go south
            j++;
         }
         else { // otherwise go north-east, with wrap around if necc.
            j--; if (j<0) j = n-1;
            k++; if (k==n) k = 0;
         }
         num++;
      }
      // print magic square
      for (j=0; j<n; j++) {
         for (k=0; k<n; k++) {
            // print  ms[j,k] right-justified in field of width 5
            System.out.printf("%5d", ms[j][k]);
         }
         System.out.println();
      }
   }
}
```

In Java, 2-dimensional arrays are treated as an array of arrays. For example, b[1] is a 1-dimensional array of integers in its own right – it has four elements b[1][0], b[1][1], b[1][2], and b[1][3] (in the picture above equal to 3, 6, 4, 5). It is allowable to supply b[1], say, as an argument to a method whose parameter is a 1-dimensional array of integers.

10 Large example: Good players

We present a program that is more complex than anything we have seen to date. As usual, the program is not of any importance per se. It is presented so that you can understand better how programs are composed as a collection of classes.

Players in a certain sport play a certain number of games in a season, and score a number of points in each game (up to a maximum of 500, say). Think of a batsman in cricket, or a basketball player, or a rugby player. We make a program which reads a text file of player data that looks like

```
Bill Smith, 23 56 47 67 88 83 44 67
Claude van Damme, 54 1 23 54 0 35
Fred Jones, 66 77 88 99
Jane Campbell, 44 55 66 77 88 99 123 145
Mike Murphy, 33 1 12 0 0 23 34
```

Each line consists of a player's name (at most 25 characters, say) terminated by a comma, followed by a list of the player's scores for the season. A player has at least one and at most 25 scores, say, for a season. A player is deemed to be a good player if he/she has played at least five games and has averaged at least 25 points over all games of the season. The program should produce a list of good players, typified by

```
Bill Smith              59.38       88  83  67  67  56
Claude van Damme        27.83       54  54  35  23   1
Jane Campbell           87.13      145 123  99  88  77
```

The output contains a line for each good player. Each line consists of the player's name, his/her average for the season rounded off to two decimal places, and his/her best five scores in descending order, all formatted in a neat columnar style as indicated. The program is to be designed under the assumption that the input comes from a text file by re-direction at the command line.

We construct the program in an object-oriented style. We begin by identifying the classes of objects that constitute the subject-matter of the problem. We might identify "name", "score", "collection of scores", "player", and "collection of good players".

- We can represent "score" as an integer and so it is hardly worth making a class of it.
- The same might be said for "name" which we can represent as a string, but nevertheless we shall make a class for it. A reason to do so is that reading a name is not so straight-forward – we shall have to recognise the comma that marks the end of the name.
- A "collection of scores" can be represented by an integer array, together with a count of the number of scores in the array (which may be anything from 1 up to 25).
- A player can be represented by a "name" and a "collection of scores".

- We probably won't need to represent a "collection of players" directly in the program, as it is likely that we can simply print each good player as we discover him or her while processing the input.

For each class, we need to decide the appropriate operations. Typically, some operations do not manifest themselves until the program is well into its development, but in this case the operations are pretty straightforward (at least to an experienced programmer).

- For the "name" class, we only need operations to input and output a name.
- For the "player" class, we need input and output operations, and a function to determine if a player is good.
- For the "scores" class, we again need input and output operations, as well as functions to yield the average of the scores, and to determine if a set of scores is "good" by the criteria given in the problem statement.

An outline of the program is as follows:

```
class Name {
  private String name; // assume length < 25
  void get() { /* read a name */ ... }
  void put() { /* display a name */ ... }
}

class Scores {
  private int[] scores = new int[25]; // scores, max 25
  private int n = 0; // first n elements in scores significant
  void get() {  /* read  a line of scores */ ... }
  void put() { /* display best scores in descending order */ ... }
  double average() { /* average of scores */ ... }
  boolean goodScores() { /* is this set of scores "good"?  */... }
}

class Player {
  private Name playerName; // player's name
  private Scores playerScores; // player's scores
  void get() { /* read name and scores*/ ... }
  void put() { /* display details of good player */ ... }
  boolean goodPlayer() { /* is this player "good" */... }
```

```
}

class GoodPlayers {
   public static void main(String[] args) {
      while (! Console.endOfFile()) {
         Player p = new Player();
         p.get();
         if (p.goodPlayer())  p.put();
      }
  }

}
```

It is just coincidental here that most classes have both input and output methods; this is not usually the case. Class Scores uses a partially filled array. Class Player uses two nested objects (a player's name and scores).In main(), we repeatedly create new instances of Player. In this case we can reduce the computational cost by creating just one instance, and re-using it for each item read in, as follows:

```
   public static void main(String[] args) {
      Player p = new Player();
      while (! Console.endOfFile()) {
         p.get();
         if (p.goodPlayer())  p.put();
      }
   }
```

However, it is not always wise to be so economic, because re-using objects sometimes introduces a risk of inadvertent aliasing.

In writing the code, we shall use method

```
Console.hasMoreTokens()
```

which, you will recall, returns true or false according to whether or not there are more tokens on the current input line (this works best when input is redirected from a text file). Recall also that for integers j and k, j/k yields an integer with any fraction truncated. For real division, at least one of the arguments must be real. Finally, we shall need to align displayed items in neat columns. By

convention, names are aligned left-justified, and numbers are aligned right-justified. The complete program follows. Classes are followed by some further explanatory words.

```
class Name {

  private String name; // assume length < 25

  void get() {
    // Read comma-terminated name from current line
    // (comma discarded)
    name = Console.readToken();
    while (!name.endsWith(",")) {
      name = name + " " + Console.readToken();
    }
    // remove comma
    name = name.substring(0, name.length()-1);
  }

  void put() {
    System.out.printf("%-25s", name); // padding on right to length 25
  }

}
```

Class Name is self-explanatory. Class Scores is

```
class Scores {

  private int[] scores = new int[25]; // scores (partially filled)
  private int n = 0; // first n elements in scores significant

  void get() {
  // Read a line of scores
    while (Console.hasMoreTokens()) {
      scores[n] = Console.readInt();
      n++;
    }
  }
```

```
private void sort() {
// Sort scores[0..n-1] in descending order
      int lft = 0; // scores[lft..n-1] to be sorted
      while (lft<n-1) {
         // find maximum in scores[lft..n-1]
         int max = lft;  int i = lft+1;
         // always scores[max] is maximum in scores[lft..i-1]
         while (i<n) {
            if (scores[i]>scores[max])
               max = i;
            i++;
         }
         // scores[max] is minimum in scores[lft..n-1]
         // swap scores[lft] and scores[max]
         int temp = scores[lft]; scores[lft] = scores[max]; scores[max] = temp;
         // advance ...
              lft++;
      }
 }

double average() {
// average of scores (assumes scores non-empty)
  int total = 0;
  for (int i=0; i<n; i++)
     total = total+scores[i];
  return (double)total/n; // "(double)" so that division yields a real
}

boolean goodScores() {
//  Do scores meet criteria for goodness?
  final int Numscores = 5; // minimum number of scores to be "good"
  final double threshold = 25.0;
  return (n>=Numscores && average()>=threshold);
}

void put() {
```

```
  // Display best scores in descending order on current line
    sort();
    final int Numscores = 5; // minimum number of scores to print
    for (int i=0; i<Numscores; i++) {
       System.out.printf("%4d", scores[i]);  // print score[i] padded to length 4
    }
  }
}

class Player {

  private Name playerName = new Name(); // player's name
  private Scores playerScores = new Scores(); // player's scores

  void get() {
  // Get a line of player data
    playerName.get();
    playerScores.get();
  }

  void put() {
  // Display players name, average score, and best five scores in
  // descending order on a line, appropriately formatted
    playerName.put(); // name
    System.out.print("   "); // column separator
    System.out.printf("%7.2f", playerScores.average());
    System.out.print("        "); // column separator
    playerScores.put(); // scores
    System.out.println();
  }

  boolean goodPlayer() {
    return playerScores.goodScores();
  }
}
```

Note that method get() does not read the name and scores directly from the standard input, but invokes the appropriate methods in component objects.

```
class GoodPlayers {

  public static void main(String[] args) {
    while (! Console.endOfFile()) {
      Player p = new Player();
      p.get();
      if (p.goodPlayer())  p.put();
    }
  }
}
```