CS112 Lecture: Arrays and Collections

Last revised 3/25/08

Objectives:

1. To introduce the use of arrays in Java
2. To examine some typical operations on arrays and introduce the appropriate patterns
3. To introduce sorted arrays
4. To introduce multi-dimensional arrays
5. To introduce collections

Materials:

1. Projectable and demonstrable version of non-array and array versions of "reverse list of numbers" program, plus improved version allowing arbitrary size.
2. Projectable version of excerpt from “array of runners” version of steeple chase robot problem
3. Projectable version of code excerpts illustrating array operations: Figure 14.5 in book, sum, maximum, searching, sorting, expansion
4. ArrayParametersDemo.java, .class + handout
5. Projectable version of ordered array of runners example from text - Figures 14.26, 14.27, 14.28 plus code for RaceStatistics class
6. Excerpts from AddressBook class given to students for project 3
7. Non recursive solution to word frequency problem from recursion lecture
8. Projectable version of YearlyCalendar example from text

I. Introduction to Arrays

A. Consider the following problem: we want to read in a list of 5 numbers and print them out in reverse order.

1. Clearly, we need to read all of the numbers before we can print any of them out. This means we have to store all the numbers in variables.

2. One solution would be to use 5 variables:

   PROJECT BadReverse.java

   DEMO

3. However, needing to have 5 distinct variables is cumbersome, and an approach like this would become essentially impossible if we were working, say, with 100 numbers, or 1000, or 10,000 !

4. To deal with situations like this, Java - like most programming languages - provides a built in data structure called an *array*. In Java, a variable is declared to be an array by following the type name with a pair of square brackets ([]), and individual elements in the array can be referenced by following the name of the variable with a subscript enclosed in square brackets. In particular, our example could be handled as follows:

   PROJECT GoodReverse.java

   DEMO

   Note that the complete program is now shorter than the original program - and would be much shorter if we compared programs for a larger number of values. Further, it could easily be modified to work with *any* number of numbers by changing the initial declaration of the size of the array. Every Java array has a field called length with specifies the number of elements specified when the array was created. (Note that, for arrays, this is a *field*, not a method, so no () are used.)

5. In fact, it would be easy to create a variant of this program which allows the user to specify the number of numbers when the program is run.

   PROJECT EvenBetterReverse.java

   DEMO

B. Recall that earlier we say that Java has two basic kinds of data types: primitive types and reference types. The latter category has two subcategories - objects and arrays. Arrays in Java can be thought of as a special kind of object); however, the formal definition of the language distinguishes them because of slight differences in the way they are used. (For example, arrays have no methods).

C. To use an array in Java, you must:

1. Declare an array variable - two alternative, but equivalent syntaxes:

   < type > [] < variable name > (preferred)

   Example: `int [] number;`

   or

   < type > < variable name > [] ("C" style declaration)

   Example: `int number [];`

2. Allocate storage for the array, using new

   < variable name > = new < type > [< size >]

   (*Note:* the type used here must be the same as the type used when declaring the variable; and the size must be known at the time the array is created - it can either be an integer constant, or an integer variable or expression; in the latter case, the value of any variables at the time the array is created are what is used.)

   Example: `number = new int [5];`

   This can be combined with declaration

   < type > < variable name > [] = new < type > [< size >]

   Example: `int [] number = new int [5];`

3. You can now

   a) Refer to the array as a whole by using its name

   b) Refer to the individual elements of the array by using

      < variable name > [< subscript >]

      where < subscript > is an integer in the range 0 .. size - 1

      Examples:

      ```
      number[3]
      number[2*i+1]
      ```

      (*Note:* Java uses zero-origin indexing. An array declared with size n has elements 0 .. n - 1). So, the first element is called [0], the second [1] ...

      Note the distinction between the variable name all by itself - which stands for the *entire* array, and the variable name plus subscript, which stands for an individual *element* of the array. Operations such as arithmetic, input, and output are done on the individual elements.

      Example: If a given building is a single family home, you can address mail directly to it. If it is an apartment building, you must specify a particular apartment by giving an apartment number as well. You can

refer to the whole building for certain purposes - such as tax assessment - but most of the time you will need to refer to a specific apartment by number.

c) Refer to the number of elements in the array by

< variable name > . length

Example: `number.length`

D. One important characteristic of an array is that all of the elements of the array have the same type. The type of the elements of an array, however, can be any valid Java type.

1. A primitive type (boolean, char, int, etc.) - as in the example above

2. An object type. In this case, it is necessary not only to create the array, but also to create the individual elements of the array - since they are objects.

   EXAMPLE: Consider the robot relay race problem, again. We could extend the program to handle any number of robots, as follows.

   PROJECT Code excerpt

```
int [ ] startAves = { 1, 5, 7, 13 };

SteepleChaseRobot [ ] runner = new
      SteepleChaseRobot[startAves.length];

for (int i=0; i < startAves.length; i ++)
{
     if (i < startAves.length - 1)
           runner[i] = new RelaySteepleChaseRobot(
             1, startAves[i], Directions.EAST, 0);
     else
           runner[i] = new SteepleChaseRobot(
             1, startAves[i], Directions.EAST, 0);
}

for (int i = 0; i < startAves.length; i ++)
      robot[i].runRace();
```

3. Another array type - yielding an array of arrays, or a *multidimensional array*. (We'll talk more about this later.)

E. Of course, it is possible to have an array of type char. How does this differ from a String?

1. In some programming languages (e.g. C) there is no distinction - strings in C are just arrays of characters.

2. In Java, type type String is a class that *uses* an array of char internally to store the characters, which the various methods access. It is not, however, possible to manipulate the array of characters comprising a String directly.

3. Interestingly, the C++ language supports *both* representations for strings - arrays of char (so-called "C strings") and its own string class. The latter, however, has many advantages because one is not constrained to a fixed size - and continuing use of the former turns out to be the reason for one of the most common vulnerabilities exploited by Internet worms - the so-called "buffer-overflow" problem.

F. Array initializers

1. Ordinarily, when an array is created, its elements are initialized to the default initial value for the type involved - e.g. zero for numbers, '\000' for characters, false for booleans, or `null` for reference types.

   (`null` is a Java reserved word For any reference type, `null` is the value that means the variable does not (yet) refer to anything. It is always an error to try to execute any method of a variable that is `null`)

2. It is possible, however, to specify the initial value for an array when it is declared - in which case an abbreviated notation is used that combines declaration, creation, and initialization.

   < type > [ ] < variable name > = { < expression > , < expression > ... }

   EXAMPLES

   a) An array containing of all the prime integers between 1 and 20:

      int [ ] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

   b) An array of strings containing the names of the people in the first row of the room

      String [] names = { --- whatever --- };

   c) Typically, when we initialize an array this way, we use *constants* as the initializers. Actually, though, it is possible to use an Java expression whose value can be calculated at the point the array is declared - but we won't pursue this further.

II. Operations on Arrays

A. One typical thing to do with an array is to perform some operation on each element of the array. This is most often done with a for loop. We'll look at several examples:

1. The book developed a Triangle class where a Triangle was represented by an array of three lines. Various operations on the Triangle could be implemented by performing the same operation on each of the lines.

   PROJECT: Figure 14.5 from the book

2. Calculating the sum of all the elements in an array.

   a) Suppose we have an array x of doubles. To store their sum in a variable called sum, we could proceed as follows:

   PROJECT

```
double sum = 0.0;
for (int i = 0; i < x.length; i ++)
    sum += x[i];
```

   b) Alternately, if we want to work with all the elements of the array - as is the case here - we could use an alternate form of the for loop (called the enhanced for loop, and only available beginning with JDK 1.5)

   PROJECT

```
double sum = 0.0;
for (double item : x)
    sum += item;
```

   (The meaning is that item is a double which should successively taken on the value of each element of the array x)

3. Finding the maximum (or minimum) valued element in an array.

   Suppose we have an array x of doubles. We want to store the value of the largest element in x in a variable called max.

   a) The following is a first attempt - though it has a problem

   PROJECT

```
double max = // See discussion below
for (int i = 0; i < x.length; i ++)
    if (x[i] > max)
        max = x[i];
```

The obvious problem with this solution is we do not know what initial value to give to max. How can we solve this? *ASK*

b) The following is a solution that solves our problem

PROJECT

```
double max = x[0];
for (int i = 1; i < x.length; i ++)
  if (x[i] > max)
      max = x[i];
```

Note how we start examining array elements at x[1], since we initialized max to x[0].

c) The enhanced for loop is less applicable here, since our loop explicitly begins with element [1]; however, the correct result could still be produced by using an enhanced for, given that no harm is done comparing an item to itself.

ASK CLASS FOR ENHANCED FOR VERSION - THEN PROJECT

```
double max = x[0];
for (int i = 1; i < x.length; i ++)
  if (x[i] > max)
      max = x[i];
```

4. Searching an array to see if a given value is present in it. We will devote more time to this in a later lecture, but we'll look at one method now.

Suppose we have an array of Student objects called student, each of which has a method called getName(), and we want to see if we have a Student object for a student named "Aardvark". The following code will return the appropriate object if one exists, or null if it does not:

PROJECT

```
int i = 0;
while (i < student.length &&
       ! student[i].getName().equals("Aardvark"))
    i ++;
if (i < student.length)
    return student[i];
else
    return null;
```

a) Notice a pattern that is characteristic of searches: the test for the loop contains *two* conditions to be tested on each iteration, which can be paraphrase as "while there is still hope of finding what we're looking for and we haven't yet found it yet ...". This relates to the fact that there are always two possible outcomes of a search: we may find what we are looking for, or we may conclude it doesn't exist.

b) Note, too, that we test the "there is still hope of finding it" case *before* we test the "have we found what we're looking for case".
Why?

ASK

The test student[i].name.equals("Aardvark") would not be legal if i were not < student.length.

c) One last point: the code we have written returns the actual *object* that matched. We could, instead, return the *index* of the object that matched. (In which case return student[i]; would become return i;). One question arises in this case, though - what should be return if no match is found?

ASK

(1) Clearly, the value returned must be one that cannot possibly be a legitimate index of an array element. One possibility is -1, in which case the if statement at the end becomes:

```
if (i < student.length)
  return i;
else
  return -1;
```

(2) Alternately, we could return a value equal to the length of the array, which is clearly not a possible element since subscripts range from 0 to length -1. In this case, the final if simplifies to a single statement:

```
return i;
```

(3) Which alternative is better?

ASK

The first is better, since it does not require that the caller of the search code know the length of the array - which could, in any case,

vary if we make provision for expanding the array if we need more room. (The simplicity of the code in the latter case is more than made up for by the additional complexity of the work done by the user of this search routine.)

(4) This is not a case where the enhanced for loop could be used. Why?

ASK

In a search, we generally don't need to consider all the elements of the array.

5. Sorting all of the elements in the array based on their value. We will devote more time to this in a later lecture, but we'll look at one method now - a method called *bubble sort.*

Suppose we have an array of Strings called name that we want to sort into ascending alphabetical order. The following would do the job:

PROJECT

```
for (int i = 1; i < name.length;  i ++)
    for (int j = 0; j < name.length -  i; j ++)
        if (name[j].compareTo(name[j+1]) > 0)
        {   // switch name[j] with name[j+1]
            String temp = name[j];
            name[j] = name[j+1];
            name[j+1] = temp;
        }
```

Discussion:

a) The outer loop iterates length - 1 times

b) Each time through the outer loop, we guarantee that the *largest* element of name[0..length - i] is placed into slot length - i - so after length - 1 iterations slots 1 .. length-1 are guaranteed to contain the correct values, which means that slot 0 does too.

c) There are various improvements possible, which we will not discuss now.

6. Expanding an array to accomodate growth over time

a) One problem one faces in using an array is deciding how big to make it - especially if it is being used for a problem where the number of elements in the array can grow over time.

This is an issue because the size of the array must be specified when it is constructed.

(1) One approach is to specify a size that is so large that it is hard to conceive that any real problem will exceed it.

Example: If we were using an array to record all of a person's children, a size of 30 is probably safe! [But 10 or even 20 is not]

(2) A problem with doing this is that a lot of space tends to be wasted - e.g. the average American family has about 2.3 children!

(3) The book uses this approach in the example if develops regarding recording the times of runners - it takes the size of the largest team and multiplies this by the number of teams to get a safe maximum - unless the program is used without modification in a different town!

b) An alternative approach is to allow for growth by - when necesssary - creating a new, larger array and then copying the existing elements to it.

EXAMPLE: Figure 14.20 modified to grow the array when needed

PROJECT

```
public void addRacer( int bib, String time ) {
    if ( racerCount >= racer.length ) {
        // No more room in the array - grow it
        RacerInfo [] newracer =
            new RacerInfo[ 2 * racer.length];
        // Copy existing values into the new array
        for (int i = 0; i < racer.length; i ++)
            newracer[i] = racer[i];
        // Replace the array with the new, bigger array
        racer = newracer;
    }
    racer[ racerCount ] = new RacerInfo( bib, time);
    racerCount++;
}
```

B. Passing an entire array as a parameter to a method, or returning an array as the value of a method.

1. The book shows an example of returning an array as the result of a method:

   PROJECT Figure 14.6

2. It is also possible for an array to be a parameter to a method. The fact that arrays are reference types has some interesting implications.

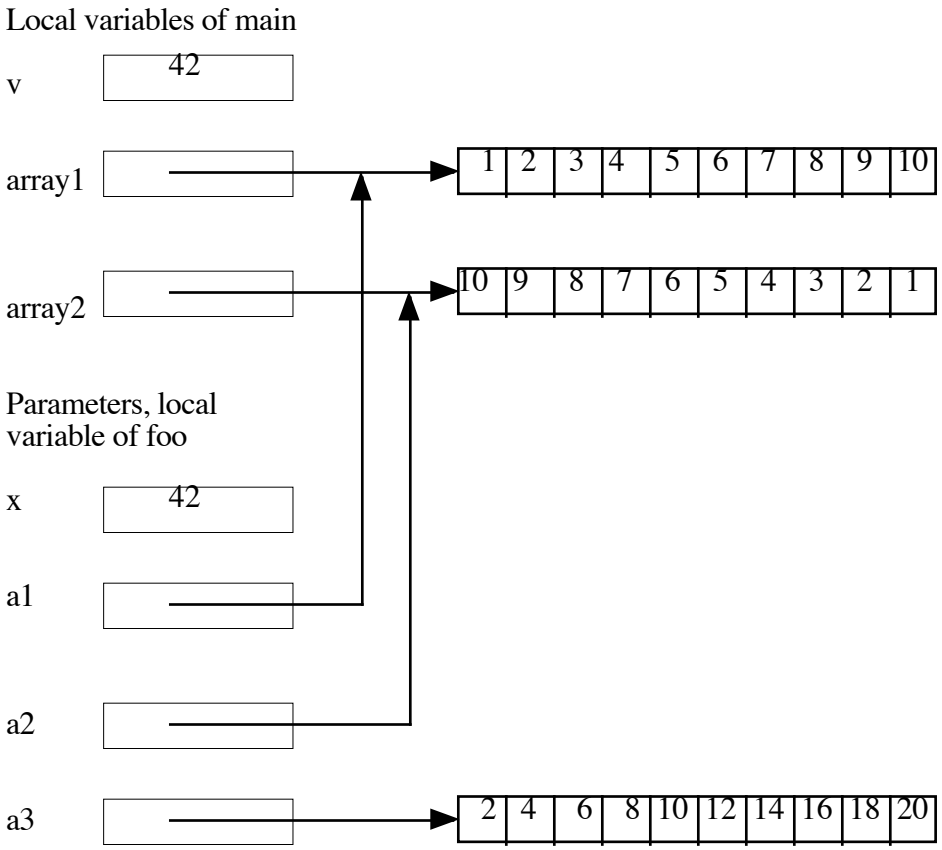
   HANDOUT: ArrayParametersDemo.java

   What will the output of this program be?

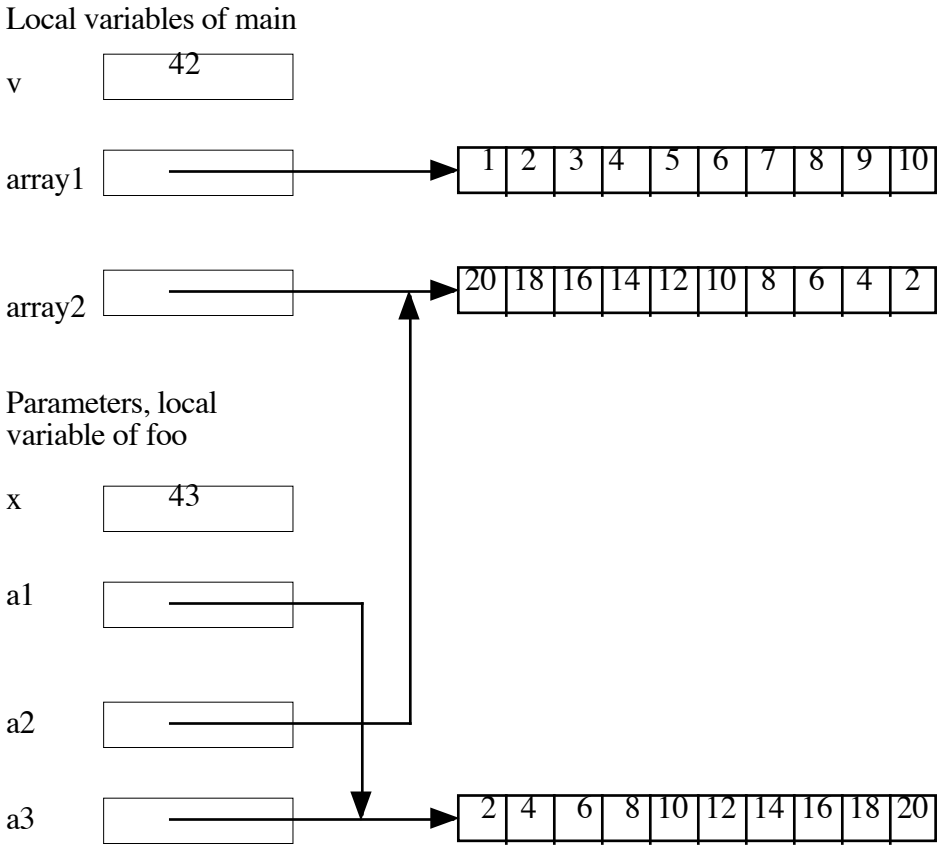
   ASK

   DEMO

   We can see why the program does what it does by considering a state of memory diagram showing the state that exists upon entry to foo():



Upon exit from foo(), we have the following state of memory



III. Ordered Arrays

A. Sometimes we have a problem in which we need to keep the elements of an array in some order. The book develops an extended example of this, in which an array is used to keep track of runners in a race, and is maintained in order of finish.

B. In a case like this, inserting a new element into the array requires shifting the existing elements that belong after it.

PROJECT Figures 14.26, 14.27, 14.28 showing how inserting a new racer with time 23:08 requires shifting 6 other finishes

PROJECT Code for addRacerAtPosition

C. A similar issue arises when removing an element, in which case elements after it must be shifted down.

PROJECT code for removeRacerAtPosition() method

IV. Collections

A. An array can be thought of as a very simple case of a data structure called a *collection*. Collections are useful whenever some one object must be related in some way to multiple other objects of the same type.

EXAMPLE: Suppose we were trying to model the college's registration system by using Student objects to model individual students and Course objects to model individual courses.

1. Each student object needs to be related to some number of courses in which the student is enrolled.

2. Each course object needs to be related to some number of students who are enrolled in it.

3. This could be managed by having an array field in each class - e.g.

   a) in class Student:
   ....
   Course [] coursesEnrolledIn;
   .

   b) in class Course
   ...
   Student [] studentsEnrolled;

   c) However, this solution suffers from two serious problems. What are they?
   ASK

   (1) When an array is created, we must give it a specific size, which cannot thereafter be changed. This poses a problem here, because students may add or drop courses (requiring a change in the size of their coursesEnrolledIn array, and also a change in the size of the studentsEnrolled array for the corresponding course.)

   (2) One refers to an element of an array by its *position* in the array - e.g. studentsEnrolled[0] is the first student on the list of students for a course. But in cases like this, one typically wants to be able to refer to an array element by its value - e.g. the id or name of the student.

d) These problems could be managed by

(1) Creating a new array of the appropriate size and copying the elements from the old array to the new whenever a student adds or drops a course, as we discussed earlier

(2) Using code like the searching example we did earlier to find a specific student in the list of students (or course in the list of courses).

(3) However, these solutions are cumbersome, and can be quite inefficient if the list is long.

B. To deal with issues like this, the Java library includes a large number of collection classes that provide various additional functionalities by way of code that has been written by the authors of the library. We will discuss collections more thoroughly in CS211; for now, we just want to look at a couple kinds of collection that we either have seen or will see.

1. In project 3, the AddressBook class manages a collection of Persons. It is a subclass of a Java library class known as `java.util.Vector`.

PROJECT excerpts from AddressBook for project 3 given to students

a) Class `java.util.Vector`. is an extension of the basic notion of an array. In particular, it keeps track of the current number of elements and automatically expands itself when necessary to accommodate a new element.

b) However, elements in a vector are still accessed by explicitly specifying a position. (This is not an issue in the address book problem, because the edit an delete use cases

c) Since JDK 1.5, class `java.util.Vector` has been a generic class which allows one to explicitly specify the type of elements it contains, just as one does with an array. (Note form of extends)

d) GO OVER A SUBSET OF CODE

(1) constructor

(2) add()

(3) getPerson()

(4) removePerson()

(5) sortByName() - note how it makes use of a library method that actually does the sorting, based on a comparator specified via an anonymous class

(6) then skip to search(). Note how we explicitly specify a starting point (0 initially). elementAt(i) returns a Person object, and Person defines a method called contains() which succeeds if the value specified appears in one of the fields.

(7) Mention setupTests()

2. In the lecture on recursion, we looked at a non-recursive solution to the word frequency problem. This made use of a library collection called `java.util.TreeMap`

   PROJECT non recursive solution to AddressBook problem

   a) A map stores a collection of key, value pairs. In this case, the key is a word, and the value is the frequency of occurrence of the word. (Of course, if a word does not occur in the text, it does not occur in the map either).

   b) Since `java.util.TreeMap` is also generic, in the declaration and the constructor (both) we specify the key type (String) and value type (Integer).

   c) The put() method is used to store a key and associated value, or to change the value associated with a given key.

   d) The get() method returns the value associated with a given key - or null if the key does not occur.

   e) The keySet() method returns a collection consisting of just the keys, which is used for printing the collection.

   f) Note how the enhanced for can be used with any kind of collection - the set, in this case

V. Multidimensional Arrays

A. As noted earlier, the elements of an array can be of any type - including another array type. This leads to the possibility of *multidimensional* arrays.

EXAMPLE:

Suppose we were writing a computer chess game, and had created a class Piece to model individual chess pieces. Then a board could be represented as an 8 x 8 array of pieces - as follows:

```
Piece [ ] [ ] board;
board = new Piece [8] [ ];
for (int row = 0; row < 8; row ++)
    board[row] = new Piece [8];
```

We could refer to an individual piece - say the piece in row 2, column 3, by syntax like:

```
board[row][column]
```

(Note that the following syntax, used in some programming languages for accessing elements of a multi-dimensional array, is *not* legal in Java:

```
board[row, column]   // NO!!!
```

B. Actually, for initializing multi-dimensional arrays, a shorter but equivalent syntax is available

```
Piece [] [] board;
board = new Piece [8] [8];
```

This initializes board to 8 uninitialized arrays of pieces, then in turn initializes each array to be an array of 8 pieces, as desired.

C. The book develops an extended example of storing a calendar in a two-dimensional array

1. The main array has twelve elements, corresponding to the 12 months of the year.

2. The array for each month has 28 - 31 elements, corresponding to the days of that month

   PROJECT and discuss code for YearlyCalendar

3. A question for the class - can you think of a simpler way to do what getDays() does?

   ASK

   Use array with an explicit initializer, then fine tune the value for February

   Add the following to the constructor

```
int [] daysInMonth = {31,28,31,30,31,30,31,31,30,31,30,31};
if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
     daysInMonth[1] ++;
```

   Change `getDays(month)` to `daysInMonth[month]`,

D. Another important use of two-dimensional arrays is to store an image, with each element representing the brightness of a particular pixel in the image. The book discusses this, and project 4 will be based on this idea.