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 multi-dimensional arrays
4. To introduce collections

Materials:

1. Projectable version of “array of runners” version of steeple chase robot problem
2. MultiInputPaneDemo to demonstrate, plus handout with code from it and excerpts from MultiInputPane
3. ArrayParametersDemo.java, .class + handout
4. AddressBook implementation using a Hashtable: Handout of code side by side with Wu’s array implementation + transparency to discuss

I. Introduction to Arrays

A. Consider the following problem: we want to read in a list of 10 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 10 variables - e.g.

   ```java
   int n1 = input.getInteger("Number 1");
   int n2 = input.getInteger("Number 2");
   ...
   int n10 = input.getInteger("Number 10");
   output.print(n10 + " ");
   output.print(n9 + " ");
   ...
   output.println(n1);
   ```

   However, needing to have 10 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.
3. 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:

```java
int [ ] number;
number = new int[10];
for (int i = 0; i < number.length; i ++)
    number[i] = input.getInteger("Number " + (i+1));
for (int i = number.length - 1; i > 0; i --)
    output.print(number[i] + " ");
output.printLine(number[0]);
```

Note that the complete program is now shorter than just the 10 lines needed to input the values initially. 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.)

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 [10];

This can be combined with declaration

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

Example: int [] number = new int [10];

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)

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.

   ```java
   int [ ] startingAvenues =
   { -- starting avenues for each runner -- };
   SteepleChaseRobot [ ] runner =
   new SteepleChaseRobot[startingAvenues.length];
   for (int i = 0; i < startingAvenues.length; i ++)
   {
      if (i < startingAvenues.length - 1)
         runner[i] = new RelaySteepleChaseRobot(
            1, startingAvenues[i],
            Directions.EAST, 0);
      else
         runner[i] = new SteepleChaseRobot(
            1, startingAvenues[i],
            Directions.EAST, 0);
   }
   for (int i = 0; i < startingAvenues.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. Note that 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.

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

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

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

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

Suppose we have an array x of floats. 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

   ```java
   float 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

   ```java
   float 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 \( \text{max} \) to \( x[0] \).

3. 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 \texttt{Student} objects called \texttt{student}, each of which has a field called \texttt{name}, and we want to see if we have a \texttt{Student} object for a student named “Aardvark”. The following code will return the appropriate object if one exists, or null if it does not:

```java
int i = 0;
while (i < student.length && !student[i].name.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 \textit{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 \textit{before} we test the “have we found what we’re looking for case”. Why?

\textit{ASK}

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

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

\textit{ASK}
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:

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

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:

```java
return i;
```

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. 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:

```java
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.

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

1. We have made use of the showInputDialog() method of class JOptionPane, which allows a program to ask a user to enter a single string. What if we want to allow a user to enter multiple strings? While we could do this with a series of dialog boxes, one for each string, this can be painful to the user.

An earlier edition of our textbook included a package of classes written by the author, called the javabook package. One of these was a class called MultiInputBox, which allowed for a user to get multiple items of input with a single dialog box. Since this package was based on awt, not swing, it is not discussed in the book. However, it is possible to create a class that has similar capabilities using swing, and I have written one that you will actually use in a future project.

2. For now, we want to look at some excerpts from this class and a sample program that uses it.

EXAMPLE: MultiInputPaneDemo.java

HANDOUT: Source for the above + excerpts from MultiInputPane.java

DEMO

a) Note the method showMultiInputDialog, which comes in several versions.

(1) One takes two arrays as parameters - an array of prompts, and an array of initialValues.
(2) Another omits the array of initialValues - and utilizes the code for the first version, passing null as the initialValues parameter.

b) This method returns an array, containing the contents of the various fields after the user dismisses the dialog - or null if the user cancels the dialog.
(Go over code in handout)

3. As was true when passing an object as parameters or returning one as a value, 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():

Local variables of main
v

array1

array2

Parameters, local variable of foo
x

a1

a2

a3
Upon exit from foo(), we have the following state of memory:

Local variables of main:
- \( v \) = 42

Array1:
- Array: \( \text{array1} \)
- Elements: \( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \)

Array2:
- Array: \( \text{array2} \)
- Elements: \( 20, 18, 16, 14, 12, 10, 8, 6, 4, 2 \)

Parameters, local variable of foo:
- \( x \) = 43
- \( a1 \)
- \( a2 \)
- \( a3 \)
- Elements: \( 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 \)

III. 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:

```java
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:

```java
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:

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

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

```java
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.

IV. Collections

A. An array can be thought of as a particular 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:
   ```java
   ....
   Course [] coursesEnrolledIn;
   .
   ```

   b) in class Course
   ```java
   ...
   Student [] studentsEnrolled;
   ```
c) However, this solution suffers from a serious problem. What is it?

ASK

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.)

d) This problem could be managed by creating a new array of the appropriate size and copying the elements from the old array to the new

EXAMPLE: Create room for an additional course in the students coursesEnrolledIn array

```java
Course [ ] expandedArray =
   new Course[coursesEnrolledIn.length + 1];
for (int i=0; i < coursesEnrolledIn.length; i ++
   expandedArray[i] = coursesEnrolledIn[i];
coursesEnrolledIn = expandedArray;
```

However, this solution is cumbersome and very time consuming. A solution like this is used in some instances, though typically the array is grown by something like 50% of its current size, rather than a single element, to reduce the number of times the expansion needs to be done. (Further, this methodology is used internally by some of the standard java collection classes that do allow for expansion.)

4. Another problem is finding the appropriate object for a given student in a course, or vice versa. We could use code like the searching example we did earlier - but again, this is cumbersome, and can be quite inefficient if the list is long.

B. To deal with issues like this, the Java library includes a number of collection classes that provide various additional functionalities

1. Since the very first versions of the Java library, there have been three collection classes that are quite useful:
a) Class `java.util.Vector` is an extension of the basic notion of an array. In particular, it automatically expands itself when necessary to accommodate a new element.

b) Class `java.util.Stack` is a Last-in first-out (LIFO) sequence of objects. It allows an arbitrary number of objects, and supports removal of objects in the opposite of the order in which they were inserted.

c) Class `java.util.Hashtable` is a dictionary that maps keys to values. It stores pairs consisting of a key and a value, and allows looking up the value corresponding to a given key.

2. JDK 1.2 added a more extensive notion of collections, which incorporates these three classes and others.

a) The Collections framework distinguishes the following kinds of collections:

   (1) List: an ordered sequence of items
   (2) Set: an unordered collection of items - no duplicates allowed
   (3) Map: a collection of `<key, value>` pairs

b) The Collections framework provides pre-written code for performing various common operations on a collection - including, among others:

   (1) Iterators that allow “visiting” all the elements in a collection
   (2) Searching
   (3) Sorting

c) For now, we will not say more about the collections framework (though it really is quite nice!). We discuss it more extensively in CS211. We will say a bit more, though, about the collection classes that have been part of Java since the beginning.

C. An example of Java built in collections - using a Hashtable as compared with the example code in Wu.

**PROJECT, HANDOUT:** Excerpts from hashtable implementation of AddressBook side by side with Wu’s implementation using arrays

Discuss implementation of add (using put), search (using get), and delete (using remove).