Arrays

Lecture 11
CGS 3416 Spring 2017

March 6, 2017

Arrays

Definition:

An **array** is an indexed collection of data elements of the same type.

- **Indexed** means that the array elements are numbered (starting at 0).
- The restriction of the **same type** is an important one, because arrays are stored in consecutive memory cells. Every cell must be the same type (and therefore, the same size).

Creating Arrays

Two Steps:

- Declare an array variable (a reference to the array)
- Create the array

Formatting the Array Variable

```
type[] arrayName;        // preferred

type arrayName[];       // alternate form
```

Format for creating the array (with the operator new)

```
arrayName = new type[size];
     //type should match the type of "arrayName"
```

Examples:

- `int[] list = new int[30];`
- `char[] name = new char[20];`
- `double[] nums = new double[x];`

The array's size is stored in *arrayName.length*.

For example, the size of the "list" array above is: `list.length`

Initializing and Using Arrays

When array of a built-in type is created, elements are automatically initialized.

- numeric types - array elements initialized to 0.
- `char` type - array elements initialized to '\u0000' (unicode 0, corresponds to ASCII 0, or '\0').
- `boolean` type - array elements initialized to `false`.

Once an array is created, the notation `arrayName[index]` refers to a specific array slot – the one at the given index position.

- Arrays are indexed from 0 through (array.length - 1).
- The index must be an integer or integer expression.

Some Examples

```
int x;
int[] list = new int[5];      // create array
double[] nums = new double[10];      // create array

list[3] = 6; // assign value 6 to array item with
                    index 3
System.out.print(nums[2]); // output array item with
                    index 2
list[x] = list[x+1];

// set values in list array to {1, 3, 5, 7, 9}
for (int i = 0; i < list.length; i++)
     list[i] = i * 2 + 1;
```

Note that this last example, the for-loop, illustrates a good way to set all values in an array. Arrays and for-loops go great together!

A Shortcut

The following format can be used to declare, create, and initialize an array in one line. (Note that this also bypasses the new operation, which is implicit):

```
type[] arrayName = { initializer list };
```

The initializer list is a comma-separated list of array values. From this, the compiler can figure out the number of elements when creating the array. Examples:

```
int[] list ={2, 4, 6, 8, 10, 12, 14, 16}; //has size 8
double[] grades = {96.5, 88.4, 90.3, 70}; //has size 4
char[] vowels = {'a', 'e', 'i', 'o', 'u'}; //size 5
```

Passing arrays to methods

- Array variables can be used as method parameters.
- Arrays are not passed by value.
  - Recall that regular parameters are passed by value – i.e. the function parameter is a local copy of the original argument passed in.
  - When passing an array into a function, only the array reference variable is passed in. A copy of the array is not made. The parameter variable refers back to the original array.
  - If the method makes changes to the array, it will affect the original array that was passed in.

Copying Arrays

If we have these two arrays, how do we copy the contents of list2 to list1?

```
int[] list1 = new int[5];
int[] list2 = 3, 5, 7, 9, 11;
```

With variables, we use the assignment statement, so this would be the natural tendency – but it is wrong!

```
list1 = list2; //does NOT copy array contents
```

We must copy between arrays element by element. A for loop makes this easy, however:

```
for (int i = 0; i < list2.length; i++)
     list1[i] = list2[i];
```

Copying Arrays

There is also a static method called arraycopy in the java.lang.System class. It's format is:

```
arraycopy(srcArray, src_pos, tarArray, tar_pos, len);
```

`src_pos` and `tar_pos` indicate the starting positions to use in the copy process. `len` is the number of elements to be copied. Sample call:

```
//this is equivalent to the for-loop example above
System.arraycopy(list2, 0, list1, 0, list2.length);
```

Multi-dimensional Arrays

Use separate set of index brackets [ ] for each dimension. Examples:

```
 // a 5 x 3 table of integers
int[][] table = new int[5][3];

// a 4 x 3 matrix of short integers
short[][] matrix = { {3, 4, 5},
          {1, 2, 3},
          {0, 5, 9},
          {8, 1, -2} };
```

Multi Dimensional Arrays

create the single reference nums

```
int [][] nums;
```

create the array of references

```
nums = new int[5][];
```

this create the second level of arrays

```
for (int i=0; i < 5 ; i++)
    nums[i] = new int[4]; // create arrays of integers
```

you must always specify the first dimension

```
nums = new int[][]; // ILLEGAL - NEEDS 1ST DIMENSION
```

you do not need to specify the second dimension

```
nums = new int[5][]; // OK
nums = new int[5][4]; // OK
```

Multi Dimensional Arrays

- For a 2 dimensional array, we usually think of the first size as rows, and the second as columns, but it really does not matter, as long as you are consistent!
- We could think of the first array above as a table with 5 rows and 3 columns, for example.
- When using a multi-dimensional array, make sure that the indices are always used in the same order:

```
table[3][1] = 5;
// assigns a 5 to "row 3, column 1", in the above
interpretation
```

Higher Dimensional Arrays

One can create arrays of higher dimension than 2.
For example, if we were measuring the temperature in a rectangular volume.

```
int temperature[][][] = new int[10][20][30];//
```

This creates an array of 10x20x30=6000 integers.

temperature is an array of array of arrays
All but the last dimension must be initially specified:

```
int temperature[][][] = new int[10][10][]; // OK
int temperature[][][] = new int[10][][]; // NOT OK
```

Enhanced for loops

The enhanced for statement allows us to iterate through any array or collection without using a counter, the traditional index through the array.

Format (for arrays):

```
for ( parameter :  arrayName)
     loop body
```

Consider a typical loop for adding up the elements of a numerical array:

```
int[] values = new int[10];
// other statements that load array with data...

int total = 0;
for (int i = 0; i < values.length; i++)
     total = total + values[i];
System.out.println("The total is " + total);
```

Enhanced for loop

The following code is equivalent, using an enhanced for loop: `int[]`

```
values = new int[10];
// other statements that load array with data...

int total = 0;
for (int number :  values)
     total = total + number;
System.out.println("The total is " + total);
```

Variable-Length Parameter Lists

- A method can have an unspecified number of arguments, of a specified type.
- To do this, follow the type name with an ellipsis (...)
  Format:
  `typeName... variableName`
- The use of the ellipsis can only occur once in a parameter list, and must be at the **end.**

```
void doThing(int x, double... values) //LEGAL

void doThing(double... values, int size) // ILLEGAL

void doThing(double... x, int... numbers)// ILLEGAL
```

Variable-Length Parameter Lists

In the doThing method above, a variable amount of double arguments may be passed in. The following are all legal calls:

```
double d1, d2, d3, d4;

doThing(5, d1, d2);
doThing(5, d1, d2, d3);
doThing(5, d1, d2, d3, d4);
doThing(5, d1, d2, d3, d4, 3.4, 5.9, 12.4);
```

Variable-Length Parameter Lists

Java treats a variable-length list of arguments like an array of items of the same type, so enhanced for-loops can be used on them.

```
void printStats(double...  values)
{
     for (double val :  values)
          System.out.print(val + " ");
}
```