CSCE 155 - Java

Lab 07 - Arrays & Dynamic Memory

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review the following free textbook resources:
   - Oracle's Java Tutorial on arrays: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
   - Information on Java's Garbage Collection: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140228.html
   - Optionally read up on Perspectives on Garbage Collection: http://www.oracle.com/technetwork/articles/java/garbagecollection-488837.html

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Understand dynamic memory management and how the Java garbage collector operates
- Declare and fill an array with values
- Access an array's values individually and iterate over the entire array
- Pass an array as a parameter to a function

2 Background

Dynamic memory management involves allocating memory when needed and freeing up after it is no longer needed. Poorly written programs that do not handle memory properly can crash or cause memory leaks. A memory leak occurs when all references to a chunk of memory are lost, but the memory was not properly cleaned up. This can lead to wasted resources and a substantial slow-down in performance.

Java, like many modern programming languages offers automatic garbage collection. In the case of Java, the Java Virtual Machine manages memory for you: once an object is no longer being referenced by another object, it is available to be garbage collected and its memory freed up for further use by the JVM or by the operating system. In general, you have no control over when and how garbage collection is performed; the JVM decides when it is best and how best to go about it.

A typical example in Java:

```
1 String message = new String("Hello World!");
2 message = null;
```

The code snippet above allocates new memory space to hold the `String` object containing the `"Hello World!"` message. The variable message is holding a reference to that `String` object; it points to the memory location where the string is being stored.

In the second line when we reassign what the message variable is pointing to (to null) then all references to that original memory location are lost. In many programming languages the string would persist in memory until the program ended. In Java, it becomes available to be garbage collected and may, at some point, be cleaned up by the JVM.

Arrays hold collections of variable values of a certain type (`int` or `double` for example). In Java, arrays can dynamically allocated using the keyword `new`. A few examples:

```
int a[] = new int[100];
int n = 10;
double vector[] = new int[n];
double matrix[][] = new int[n][n];
```

The syntax requires that an integer size be indicated when the array is dynamically created. Each array is filled with the same default values as the type of variables it holds. Moreover, as a convenience, Java keeps track of the size of arrays and gives you access to the size through the `length` property:

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

3 Activities

For this lab, we'll be using several command line tools to observe memory usage. Therefore, we'll be building and running from the command line. You can still use Eclipse to clone the project and edit source files but it is suggested that you work from the command line for this lab. Clone the repository from GitHub from the command line using the following:

```
git clone https://github.com/cbourke/CSCE155-Java-Lab07
```

3.1 Observing Garbage Collection

In this exercise, we'll observe a memory leak in action and fix it.

Instructions

1. Open the `MemoryLeakA.java` source file and familiarize yourself with what it does.
2. Move to the project's root directory, `CSCE155-Java-Lab07`
3. Compile and build the project using the Apache Ant build script (`build.xml`) by

executing the following on the command line: `ant compile`

4. Now run the program by executing the following from the command line:
   `java -cp build/classes unl.cse.memory.MemoryLeakA 10`
5. Reopen the source file and comment out the line that prints the i-th order statistic; recompile with the same ant command
6. Now run the program by executing the following from the command line (this will log information on when Java's garbage collector is executing and how it affects memory):
```
java -verbose:gc -XX:+PrintGCDetails -cp build/classes
        unl.cse.memory.MemoryLeakA 1000000
```
7. Observe the output for a couple of minutes, but ultimately you may have to kill your program (control-c)
8. Answer the questions in your handout.

3.2 Breaking Garbage Collection

Java's garbage collection relieves the user from having to worry about cleaning up memory, but it does not mean that we can ignore dynamic memory issues altogether in Java. Java can still have "memory leaks" if poorly written code holds on to references even though it no longer needs them. If objects and data continue to have valid references, they are not eligible for garbage collection and remain resident in memory. In this exercise we'll observe such a situation.

Instructions

1. Open the `MemoryLeakB.java` source file and compare that to the other demo file–what are the key differences?
2. Execute this program in the background from the command line via the following (the ampersand runs the process in the background):
   `java -cp build/classes unl.cse.memory.MemoryLeakB 1000000 &`
3. To monitor how much memory your program is using, start the top program: `top -u login` where `login` is replaced with your cse login. Your Java process should be the top process.
4. Observe the performance of your program for a couple of minutes and then kill it: quit top by typing 'q' then type the command `kill 1234` where `1234` is the ID of your Java process (first column in top).
5. Answer the questions on your worksheet.

3.3 Using Arrays

In Java, arrays can be used as parameters/arguments in methods and returned as values by methods. The syntax for doing this is straightforward:

```
1 public static int[] someMethod(int a, double arr[]) {
2   //code
3 }
```

This defines a method that returns an integer array and takes, as its second argument, an array of `double` variables. In this exercise, you will write and use several functions that utilize arrays.

Instructions

1. Open `Statistics.java` in the editor of your choice. There are three incomplete methods: `findMax`, `findMin`, and `findMean`.
2. Fill in the missing parameters for each method. Your goal for each function is to find their respective statistic of a list of numbers (i.e., you'll need to find the mean of an array of integers in `findMean`). That means you'll have to pass in an array and the size of the array.
3. The calls to the methods in the `main` method are also incomplete. Pass each method the appropriate variables.
4. Fill in the method bodies to find the correct statistic. The minimum and maximum are the smallest and largest numbers in the list, and the mean is the sum of the list divided by its size.
5. Compile and test your code by using Ant as before; note you can use Ant to run your program by using `ant run-stats`
6. Complete the questions on your worksheet

4 Advanced Activities (Optional)

4.1 Activity 1

The Java Collections library offers alternatives to primitive arrays called lists which are object-oriented ordered collections that allow the user to add, retrieve, and remove elements from it without having to worry about the details of how elements are held. An example of its usage:

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add(20);
System.out.println("second element: "+list.get(1));
```

Read Oracle's Tutorial on the Collections framework: http://docs.oracle.com/javase/tutorial/collections/index.html and modify the exercises in this lab to utilize a list rather than arrays.

4.2 Activity 2

So far we've only been working with single dimension arrays, but often times it is important to represent something like a table or matrix in code. One way to accomplish this is to use a multidimensional array. You can declare a statically allocated 2D array with the statement

```
int matrix[100][50];
```

You can think of 100 as the number of rows in the table and 50 as the number of columns. Begin by writing a program that will initialize a matrix with some random numbers and find the average of each row (you'll want to use nested for loops). Once you've gotten the hang of accessing the elements in a 2D matrix, write a program that will find the product of two (square) matrices. To assist you, write a function that will automatically create a randomly populated two dimensional array and return a reference to it.