Java Chapter 09 – Advanced Array Concepts

**SORTING ARRAYS:**

* Arranging a series of objects in some logical order
* Simplest Sort 🡪 2 values out of order
* Must swap the 2 values
* Wrong Example:

valA = valB;

valB = valB;

* Right Example:

temp = valA;

valA = valB;

valB = temp;

* Making decisions when sorting:
* if(valA > valB)

temp = valA;

valA = valB;

valB = temp;

**BUBBLE SORT ALGORITHM:**

* Repeatedly compare values of items, swapping them if they’re out of order, eventually creating a sorted list
* Not the fastest or most efficient technique, BUT one of the simplest to comprehend
* Steps:

Int[] someNums = { 1, 2, 3, 4 };

* Place unsorted values in an array
* Compare the first 2 – if not in ascending order 🡪 swap
* Continue down the list – for each position x, if value at position x + 1 is smaller 🡪 swap
* \*The ‘Heaviest’ value sinks to the bottom, the ‘Lighter’ values bubble to the top\*
* Continue the sort 🡪 repeat the comparison-swap procedure until fully sorted

for(a = 0; a < someNums.length – 1; ++a)

for(a = 0; a < someNums.length – 1; ++a)

if(someNums[b] > someNums[b + 1])

{

temp = someNums[b];

someNums[b] = someNums[b + 1];

someNums[b + 1] = temp;

}

\*For a sort on any array size, the value of b must remain less than the array’s length – 1

* **IMPROVING BUBBLE SORT EFFICIENCY :**
* Use a new control variable to limit the repetitions of the inner for loop & reducing the value by 1 on each cycle through the array
* int comparisonsToMake = someNums.lenth – 1;

for(a = 0; a < someNums.length – 1; ++a)

{

for(b = 0; b < comparisonsToMake; ++b)

{

if(someNums[b] > someNums[b + 1])

{

temp = someNums[b];

someNums[b] = someNums[b + 1];

someNums[b + 1] = temp;

}

} –comparisonsToMake 🡪 decremented by 1 each pass through the list

}

**SORTING ARRAYS OF OBJECTS:**

* Similar to sorting arrays of primitives
* Difference: When making the comparison that determines the swap
* Sorting Primitive Array 🡪 compares values
* Sorting Object Array 🡪 compares object fields

public class Employee

{

private in empNum;

private String lastName;

private double salary;

public void **setEmpNum(int emp)** {

EmpNum = emp;

}

public int **getEmpNum()** {

Return empNum;

}

public void **setFirstName(String name)** {

firstName = name;

}

public string **getFirstName()** {

Return firstName;

}

public void **setLastName(String name)** {

lastName = name;

}

public string **getLastName()** {

Return lastName;

}

public void **setSalary(double sal)** {

Salary = sal;

}

public double **getSalary()** {

Return salary;

}

}

* Create Object Array 🡪 Employee[] = new Employee[5];
* Sort Employee Salaries in order:

Public static void bubbleSort(Employee[] array)

{

int a, b;

Employee temp;

int highSubscript = array.length- 1;

for(a = 0; a > highSubscript; ++a)

for(b = 0; b > highSubscript; ++b)

if(array[b].getSalary() > array[b + 1].getSalary())

{

temp = array[b];

array[b] = array[b + 1];

array[b + 1] = temp;

}

}

* 3 Differences when using bubbleSort() method for Object Arrays (VS. Primitive)
* The bubbleSort() method header shows that in receives an array of type Employee
* The temp variable created for swapping is type Employee, therefore the temp variable will hold an Employee object, not just one number or one field. \*Even though employee salaries are being compared, you do not want to swap the salaries – Instead, you swap each Employee object’s empNum & salary as a unit.
* The comparison for determining whether a swap should occur uses method calls to the getSalary() method to compare the returned salary for each Employee object in the array with the salary of the adjacent Employee object.

**INSERSTION SORT ALGORITHM:**

* Look at each list element one at a time. If an element is out of order relative to any of the items earlier in the list, you move each earlier item down one position & then insert the tested element
* Similar to the technique you would most likely use to sort a group of objects manually

int[] someNums = { 1, 2, 3, 4, 5 };

a =1;

while(a < someNums.length)

{

temp = someNums[a];

b = a – 1;

while(b >= 0 && someNums[b] > temp) {

someNums[b + 1] = someNums[b];

--b;

}

someNums[b + 1] = temp;

++a;

}

1. a is set to 1 🡪 while loop begins 🡪 temp is set to 85 & b is set to 0
2. because b is greater than or equal to 0 and someNums[b] (90) is greater than temp, the inner loop is to be entered. (if you were performing a descending sort, then you would ask whether someNums[b] was less than temp)
3. The value of someNums[1] becomes 90, and b is decremented, making it -1, so b is no longer greater than or equal to 0, and the inner loop ends
4. Then someNums[0] is set to temp, which is 85

After these steps, 90 was moved down one position & 85 was inserted in the first position, so the array values are in slightly better order than before. The values are as follows: 85, 90, 65, 95, 75

1. Now the outer loop becomes 2, the value of temp is set to 65, & b is set to 1
2. The value of b is greater than or equal to 0, and someNums[b] (90) is greater than temp – so the inner loop is entered
3. The value of someNums[2] becomes 90, and b is decremented, making it 0, so the loop executes again
4. The value of someNums[1] becomes 85, and b is decremented, making it -1, so the loops ends
5. Then someNums[0] becomes 65

After these steps, the array values are in even better order, because 65 and 85 now both come before 90. The values are: 65, 85, 90, 95, 75.

1. A becomes 3, the value of temp becomes 95, and b is set to 2
2. For the loop to execute, be must be greater than or equal to 0, which it is, and someNums[b] (90) must be greater than temp, which it is not. So the inner loop does not execute
3. Therefore, someNums[2] is set to 90, which it already was 🡪 So, NO CHANGES WERE MADE

The final sort is as follows:

1. A is increased to 4, the value of temp becomes 75, and b is set to 3
2. The value of b is greater than or equal to 0, and someNums[b] (95) is greater than temp, so the inner loop is entered
3. The value of someNums[4] becomes 95, and b is decremented, making it 2, so the loops executes again
4. The value of someNums[3] becomes 90, and b is decremented making it 1, so the inner loop executes again
5. The value of someNums[2] becomes 85, and b is decremented, making it 0; someNums[b] (65) is no longer greater than temp (75), so the inner loop ends. – values 85, 90, & 95 are each moved down one position, but 65 is left in place
6. Then someNums[1] becomes 75 🡪 values are now in ascending order – 65, 75, 85, 90, 95

**2-DIMENSIONAL & MULTIDIMENTIONAL ARRAYS:**

\*\*When declaring: int[] someNumbers = new int[3]; 🡪 can envision 3 declared integers as a column of numbers in memory 🡪 think of the array size as its height

\*\*\*Two-Dimensional arrays:

* \*Represent the HEIGHT & WIDTH of the array
* \*Picture it as having rows & columns
* \*Is an array of arrays (because each row is also and array)
* Declaration:
* Int [] [] someNumbers = new int [] [];
* Initialize:
* Int [] [] someNumbers = new int [3] [4]; 🡪 initialize individually later
* Int [] [] someNumbers = { {1, 2, 3, 4 },

{ 5, 6, 7, 8 },

{ 9, 10, 11, 12} }; 🡪 initialize at creation

* \*1st bracket = rows
* \*2nd bracket = columns
* Passing 2-dimensional Array to Method:
* Public static void displayScores(int [] [] scoresArray)
* \*NO need to insert numbers into brackets because each passed array name is a starting memory address 🡪 the way you manipulate subscripts w/in the method determines this
* Length Filed 2-dimensional Array:
* Length field hold the number of rows

someNumbers.length 🡪 3 ( 3 rows in the array)

* Each row then has a length field that holds the number of columns

someNumbers[0].length 🡪 4 (4 columns in the first row)

* Ragged Arrays:
* \*In Java you can declare a 2-dimensional array in which each row has a different length
* Declaring 🡪 define # of rows but not # of columns in rows:

Double [] [] sales = new double[4] [];

* Other Multidimensional Arrays:
* Arrays w/ more than one dimension
* In Java – can use as many dimensions as desired as long as
* You can keep track of the order of the variables needed as subscripts
* You don’t exhaust your computer memory

**ARRAYS CLASS:**

* Located in the java.util package 🡪 use import java.util.\*; to access it
* Contains methods you’ll often want to use to perform similar tasks w/ different arrays
* Methods are static
* **\*\*Useful Array Class Methods\*\* ...**

**ARRAYLIST CLASS:**

* Used to create containers and store lists of objects
* Dynamically resizable 🡪 advantage over the Arrays Class
* To use 🡪 import java.util.ArrayList; OR java.util.\*
* To declare 🡪 ArrayList<String> names = new ArrayList<String>(); - default constructor
* Data type in ‘< >’ causes Java to check that you’re assigning correct types to the list
* \*\* Optional \*\*
* The default constructor creates the ArrayList w/ a capacity of 10 items
* Specify capacity 🡪 ArrayList<String> names = new ArrayList<String>(20);
* **\*\*Useful ArrayList Class Methods\*\* …**

**CREATING ENUMERATIONS:**

* Programmer-Created data type with fixed set of values
* Enumerated Types:
* Is a class
* Identifier begins with an uppercase
* Enumerated constants appear in all CAPS
* Creating your own data types that have a finite set of legal values:
* enum Month { JAN, FEB, MAR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
* After creating an enumerated type, you can declare a variable of that type:
* Month birthMonth;
* Can assign any of the enum constants to the variable:
* BirthMonth = Month.FEB

\*\*You can declare an enumerated type in its own file, in which case the filename matches the type name & has a ‘.java’ extension. OR You can declare an enumerated type w/in a class (but not w/in a method)

\*Enumeration type like Month is a class, and its enum constants act like objects instantiated from the class, including having access to the methods of the class\*

* **\*\*BUILT-IN enum Methods (Static & Non-Static)\*\* …**
* In Java 7 & 8 – can use comparison operators w/ enumeration constants instead of the compareTo() method to return a number
* You can use enumerations to control a switch structure
* **Advantages of using enumeration types** – the Month enumeration improves the program:
* If you did not create an enumerated type for month values, you could use another type – for example ints or Strings. However, any value can be assigned to an int or a String variable, but only the 12 allowed values can be assigned to a Month
* Therefore, invalid behavior could be applied to the values – for example if integers used to represent months – you could add, subtract, multiply, or divide months which is not logical 🡪 enums make the values type-safe (only appropriate behaviors are allowed
* Enum constants provide a form of self-documentation
* As with other classes, can add methods & other fields to an enum type