JAVA PROGRAMMING 1

Summer 2018 - Christian Hur

# Unit 9 Lecture - Advanced Array Concepts

Reading: Chapter 9

# Objectives:

* Sorting arrays using
  + Bubble sort algorithm
  + Insertion sort algorithm
* Two-dimensional and multi-dimensional arrays
* Ragged arrays
* Arrays class
* ArrayList class
* Enumerations

# Sorting and Swapping

Sorting is the process of arranging a series of objects or elements in logical order.

Consider the following two values:

int n1 = 5;

int n2 = 3;

If you want to sort these two values in an ascending order or if you just want to swap their values, what’s the best algorithm to achieve this? In computer programming/computer science a step-by-step process used to solve a particular problem is called **algorithm**. In a scenario such as this, you can’t just swap their values because you will lose the other.

In other words, doing this will not work:  
  
n1=n2;

The problem is that both n1 and n2 hold the value of “5” and the value “3” is lost forever. The best algorithm is to have a third variable (a temp variable) to hold one of the values before the swap:

int temp = n1;

Now, swapping them will work:

n1 = n2;

n2 = temp;

The two variables are not sorted in the correcta acending order without losing their original data. This is a very common feature in computer science and it’s often called the **swap algorithm**.

## Sorting Arrays

Arrays are collections of data of identical element types. Often it’s important to have the array sorted in certain order for a specific purpose. Sorting is the process of arranging a series of objects or elements in logical order. Typically, arrays or a list of item can be sorted in a logical order such as ascending order (small to large) or descending order (large to small).

When is array sorted?

An array is sorted when the following is true:

For an array An, where A is the array, n = the size of the array

The array An is sorted iff (if and only if):

Examples: All the following arrays are sorted (ascending order)

[a,B,c,g,z] //case insensitive

[A,D,b,c] //case sensitive

[1,3,6,88]

Default: Empty arrays and arrays with a single element (aka **singleton** arrays) are always sorted.

Examples:

//Empty arrays

int[] numbers = new int[0];

String[] words = { };

String[] text;

int t = ReadFrom Input

text = new String[t];

//Singleton arrays

int[] keys = {123};

boolean[] truths = {true};

## Sorting Algorithms

Computer scientists have developed several algorithms to sort arrays. Below is an complete list of some known sorting algorithms:

* Bubble sort
* Insertion sort
* Selection sort
* Merge sort
* Quick sort
* Heap sort
* Counting sort

Some of these algorithms are more effecient than others (in terms of running time), but may not be the best candidate for some situations based on some conditions such as *the size of the array*, *how dispersed the elements are in the list*, *whether or not the array is already sorted*, etc. We wil explore two popular sorting algorithms: **bubble sort** and **insertion sort**.

## 

## 

## Bubble Sort

Bubble sort is one of the simplest sorting algorithms to learn.

Consider this array:

int[] a = { 101, 201, 213, 108, 566, 12 };

The length of the array is 6: *n = a.length*

**Complexity of Bubble Sort in Big-O Notation: *O*(*n2*)**

For each element in the array, Bubble Sort does ***n-1*** comparisons. Without going further how the complexity of Bubble Sort is formulated, the complexity of Bubble sort in Big-O notation (growth rates) summarizes that Bubble sort performs *O*(*n*) comparisons (when the list is sorted) to a running time of *O*(*n2*) comparisons (worst case).

If interested, you can read more about it here: <https://brilliant.org/wiki/bubble-sort/>

The bubble sort algorithm in ascending order:

int[] a = { 101, 201, 213, 108, 566, 12 };

1. Comparing two adjacent elements a[0] and a[1]. If a[0] > a[1], then swap them otherwise they’re sorted so do nothing.
2. Move compare a[1] and a[2]. If a[1] > a[2], then swap them otherwise they’re sorted.
3. Repeat steps #1 and #2 for every pair of elements until the end of the array list is reached (***n-1*** times, where ***n*** = length of array).

As you do this, you will see that all the values “bubble” up to the top of the list and thus the term “bubble sort”.

**Bubble sort example1: smallest first**

int[] a = { 101, 201, 213, 108, 566, 12 };

int len = a.length, temp;

for (int j = 0; j < len - 1; ++j) {

for (int i = 0; i < len - 1; i++) {

if (a[i] > a[i + 1]) {

temp = a[i];

a[i] = a[i + 1];

a[i + 1] = temp;

}

}

}

**FIRST ITERATION: j=0**

Curent array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 213 | 108 | 566 | 12 |

First comparison: a[0] and a[1] j=0, i=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 213 | 108 | 566 | 12 |

Second comparison: a[1] and a[2] j=0, i=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 213 | 108 | 566 | 12 |

Third comparison: a[2] and a[3] j=0, i=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 213 | 566 | 12 |

Fourth comparison: a[3] and a[4] j=0, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 213 | 566 | 12 |

Fifth comparison: a[4] and a[5] j=0, i=4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 213 | 12 | 566 |

**SECOND ITERATION: j=1**

Curent array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 213 | 12 | 566 |

First comparison: a[0] and a[1] j=1, i=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 213 | 12 | 566 |

Second comparison: a[1] and a[2] j=1, i=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 213 | 12 | 566 |

Third comparison: a[2] and a[3] j=1, i=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 213 | 12 | 566 |

Fourth comparison: a[3] and a[4] j=1, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 12 | 213 | 566 |

Fifth comparison: a[4] and a[5] j=1, i=4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 12 | 213 | 566 |

**THIRD ITERATION: j=2**

Curent array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 12 | 213 | 566 |

First comparison: a[0] and a[1] j=2, i=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 12 | 213 | 566 |

Second comparison: a[1] and a[2] j=2, i=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 201 | 12 | 213 | 566 |

Third comparison: a[2] and a[3] j=2, i=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

Fourth comparison: a[3] and a[4] j=2, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

Fifth comparison: a[4] and a[5] j=2, i=4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

**FOURTH ITERATION: j=3**

Curent array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

First comparison: a[0] and a[1] j=3, i=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

Second comparison: a[1] and a[2] j=3, i=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

Third comparison: a[2] and a[3] j=3, i=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

Fourth comparison: a[3] and a[4] j=3, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

Fifth comparison: a[4] and a[5] j=3, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

**FIFTH ITERATION: j=4 (Last iteration: n-1)**

Curent array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

First comparison: a[0] and a[1] j=4, i=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

Second comparison: a[1] and a[2] j=4, i=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

Third comparison: a[2] and a[3] j=4, i=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

Fourth comparison: a[3] and a[4] j=4, i=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

Fifth comparison: a[4] and a[5] j=4, i=4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

# DONE!

The sorted array a = [12,101,108,201,213,566]

**Bubble sort example2: largest first (more efficient)**

int[] a = { 101, 201, 213, 108, 566, 12 };

int len = a.length, temp;

for (int j = len; j > 0; --j) {

for (int i = 0; i < j - 1; i++) {

if (a[i] > a[i + 1]) {

int temp = a[i];

a[i] = a[i + 1];

a[i + 1] = temp;

}

}

}

Current Array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 213 | 108 | 566 | 12 |

First pass: j=6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 213 | 108 | 12 | 566 |

Second pass: j=5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 201 | 108 | 12 | 213 | 566 |

Third pass: j=4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 108 | 12 | 201 | 213 | 566 |

Fourth pass: j=3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101 | 12 | 108 | 201 | 213 | 566 |

Fifth pass: j=2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

Sixth pass: j=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 12 | 101 | 108 | 201 | 213 | 566 |

## Insertion Sort

The insertion sort algorithm builds a sorted array one element at a time. In Big-O notation, Insertion Sort also has a worst-case running time of ***O***(***n2***) just like Bubble Sort.

Algorithm is simple: Select each element in the array, compare it to the elements before it and insert it in the order position.

Example[1]:

int[] a = {8, 2, 4, 9, 3, 6}

Pseudocode:

for i = 1 to length(a)  
 x = a[i]  
 j = i - 1  
 while j >= 0 and a[j] > x  
 a[j+1] = a[j]  
 j = j - 1  
 end while  
 a[j+1] = x  
 end for

### Insertion Sort Implementation:

**int**[] a = {8, 2, 4, 9, 3, 6};

**int** len = a.length;

**int** currentElement;

**int** checkIndex;

**for**(**int** i = 1; i < len; i++){

currentElement = a[i]; //value of a[i] --> 2

checkIndex = i-1; //index of one element the left of temp

//Compare every element a[j] with current element

//If greater than currentElement, then swap them

//Otherwise check next one (if any)

**while**(checkIndex >= 0 && a[checkIndex] > currentElement){

a[checkIndex+1] = a[checkIndex];

checkIndex--; //decrement j and check the previous element

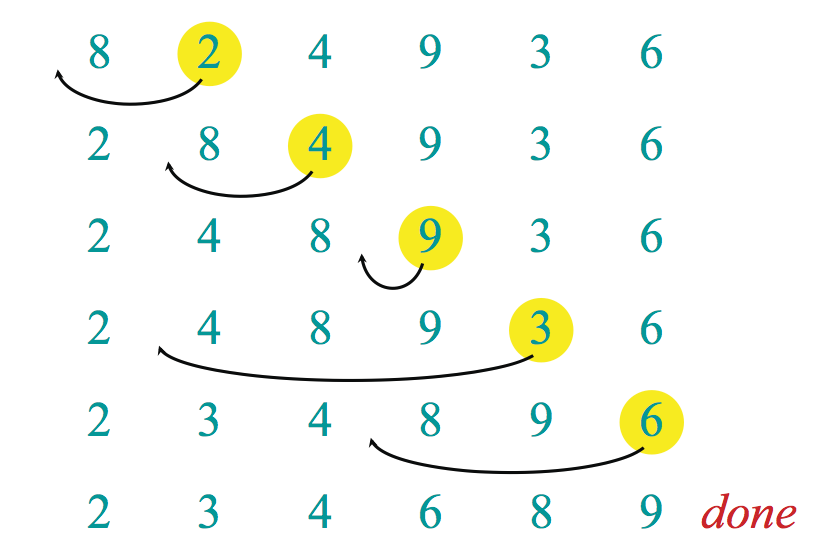
}

a[checkIndex+1] = currentElement;

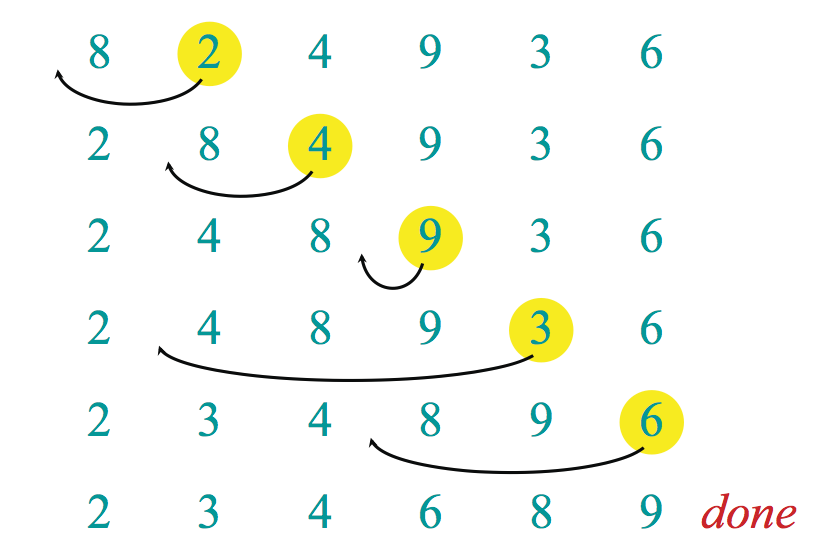
}

### INSERTION SORT ITERATIONS

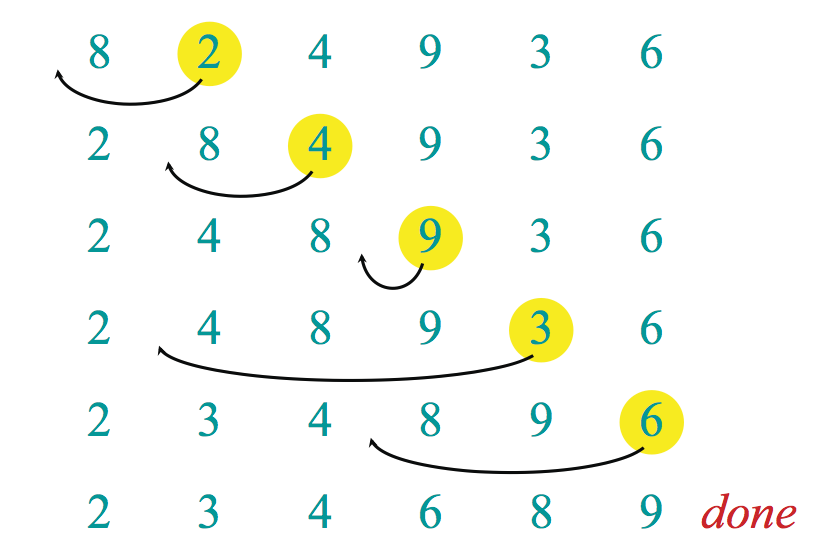
**1st Iteration: i = 1, currentElement = 2, checkIndex = 0**



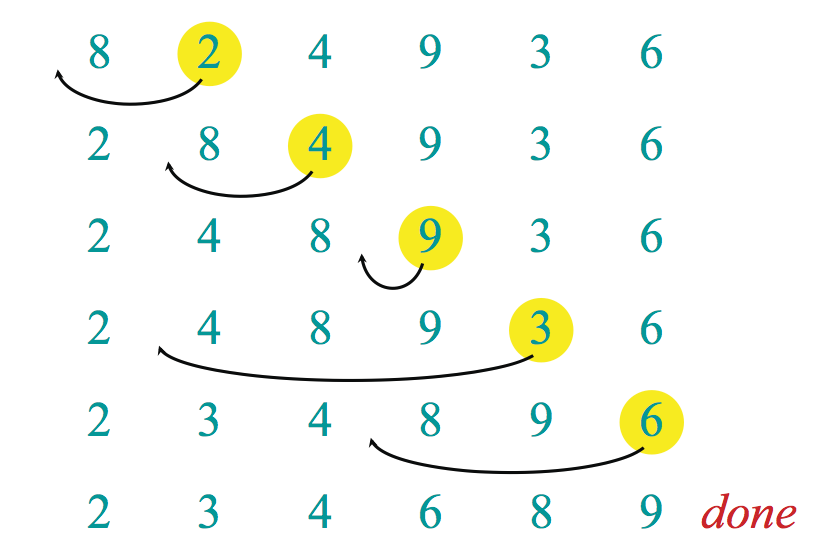
**2nd Iteration: i = 2, currentElement = 4, checkIndex = 1**



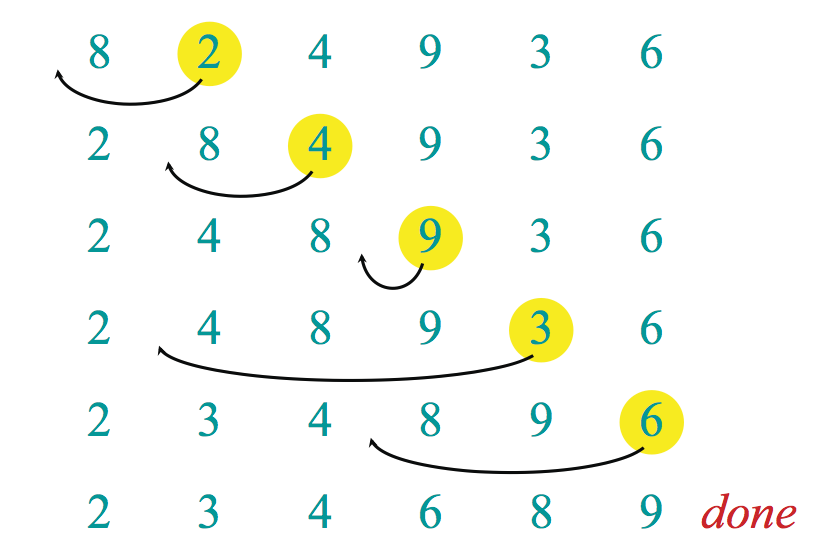
**3rd Iteration: i = 3, currentElement = 9, checkIndex = 2**



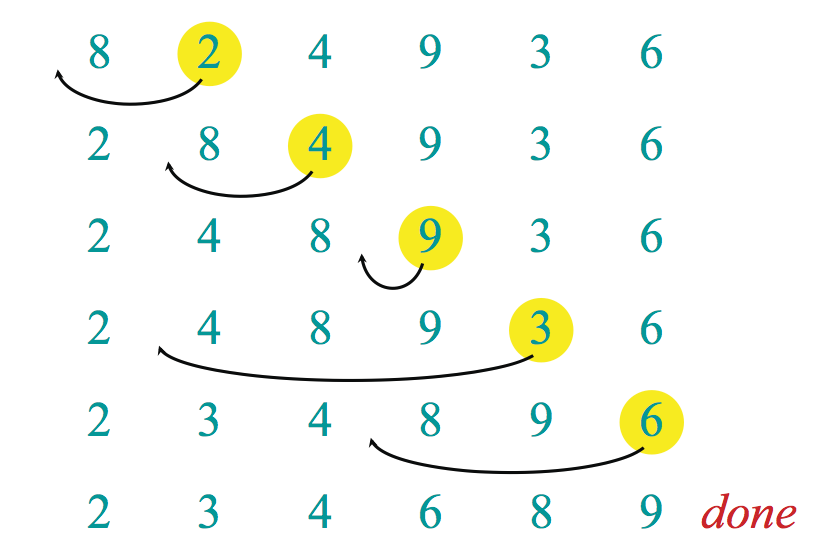
**4th Iteration: i = 4, currentElement = 4, checkIndex = 3**



**5th Iteration: i = 5, currentElement = 6, checkIndex = 4**



**DONE!**



## Multi-Dimensional Arrays

So far, we’ve used only a single-dimensional (1D) array. Java supports multi-dimensional arrays. The most common type of multi-dimensional array is a **two-dimensional** array, also called a **matrix,** a **table**, or a **spreadsheet**. A two-dimensional array is composed of a row and a column and uses two indices. A two-dimensional array is really an array of arrays -- this means that each row is an array instead of a single primitive data type.

A typical 3x3 table of text.

|  |  |  |
| --- | --- | --- |
| Apple | Ant | Ape |
| Banana | Beetle | Bird |
| Cherry | Charmelian | Cat |

## Two-Dimensional Arrays

Two-dimensional arrays (2D arrays) are indexed by two subscripts: first index for row, second index for column. Each element must be the same type - same rule for 1-dimensional array.

**Syntax**:

*datatype*[][] arraryName]; //a variable that references a 2D array of datatype

*datatype*[][] arraryName = new *dataType*[ ROW\_INDEX ][ COL\_INDEX ]; //n x m matrix

int[] array1 = {1,2,3}; //1D array

int[] array2 = {4,5,6}; //another 1D array

int[] array3 = {7,8,9}; //a third 1D array

int[][] numbers = { {array1}, {array2}, {array3} }; //3x3 matrix

Can also be written as:

int[][] numbers = { {1,2,3},

{4,5,6},

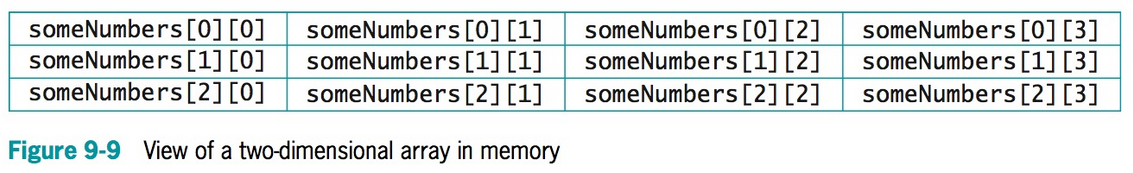
{7,8,9}

};

**More 2D Array Examples**:

**A 3X4 matrix:**

int[][] someNumbers = new int[3][4];



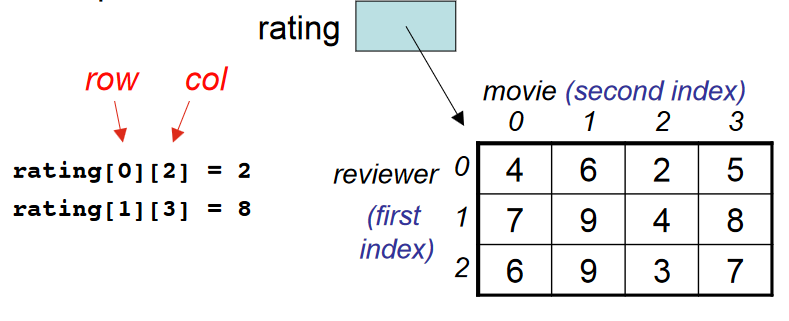
**A 3X4 matrix of ratings:**

int[][] rating = { {4,6,2,5},

{7,9,4,8},

{6,9,3,7}

};



## Size of a 2D Array

To get the length of the row and column in a 2D array, use the “length” property.

Number of rows: array.length; //Same way you’d use for 1D array

Number of columns: This gets a little tricky, but you must reference the row number then use the length property:

array[0].length; //returns the number of columns.

Typically, 2D arrays should have same equal number of columns, but sometimes they do not. This is the reason why it’s tricky is because of the possibilty of a **ragged array**, which is an array of unequal number of columns. We will discuss this in the next section below.

## Ragged Arrays

Ragged arrays are 2D arrays that do not have same number of columns. For example:

int[][] rating = { {1,2,3,4},

{3,5},

{6,8,9},

{5},

{}

};

This is a ragged array with 4 rows and variable column sizes. To get the number of rows is straight forward: rating.length;

To get the size of each column, you must reference its row index:

Index 0: rating[0].length; //4

Index 1: rating[1].length; //2

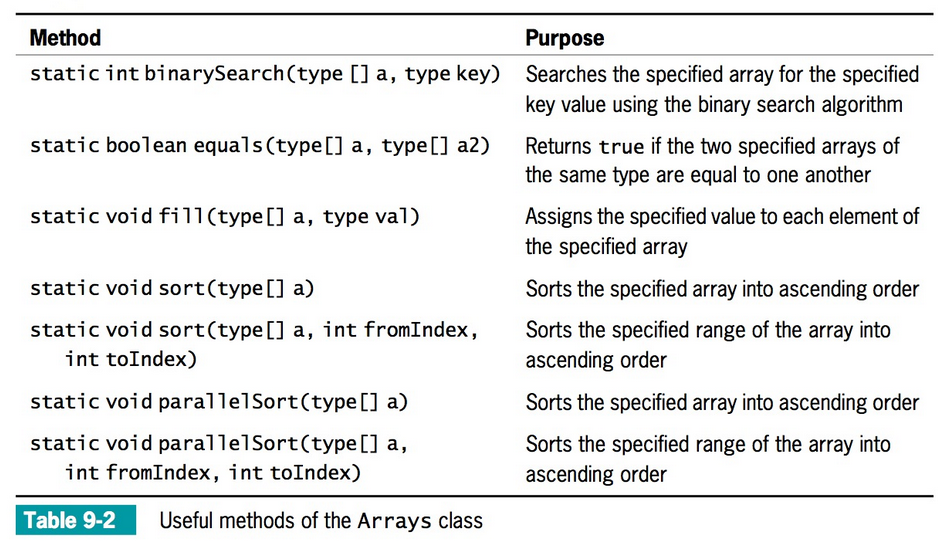
Index 2: rating[2].length; //3

Index 3: rating[3].length; //1

Index 4: rating[4].length; //0 (empty array)

## Arrays Class

Java provides an Arrays class which contains many useful helper methods for manipulating arrays quickly. The table 9-2 below lists some static methods of the Arrays class.



## Usage:

Arrays.*methodName()*;

Remember that all “void” methods cannot be on the right of “=”.

Arrays.fill() --- this a void type

int[] k = Arrays.fill(k,1); //this won’t work! -- INCORRECT

int[] k = new int[100];

Arrays.fill(k,1); **//fills with 1’s -- CORRECT Way!**

Arrays.sort() and Arrays.parrallelsort()

The **sort()** and **parrallelsort()** are identical, but parrelsort is a newer method introduced in Java 8 to handle large array of objects (especially in the millions) more efficiently.

Recall: Arrays are “references” so the array passed to them will be affected

int[] a = {8, 2, 4, 9, 3, 6};

Arrays.sort(a); //sort all elements

Result: {2,3,4,6,8,9}

Arrays.sort(a,2,4) //sort only from subscripts 2 through 4

Result: {8,2,3,4,9,6}

## ArrayList Class

The ArrayList class is and advanced feature in Java and we will only discuss it briefly in this course. ArrayList is used to create containers that store lists of objects. The main advantage of the ArrayList class is that its ***size is dynamically resizable***. Array sizes are usually fixed when they’re declared. If you want to change its sizes anytime during runtime, then you should use the ArrayList class instead. The array size expands and contracts automatically!

To use it, you must import its class library:

import java.util.\*;

Or

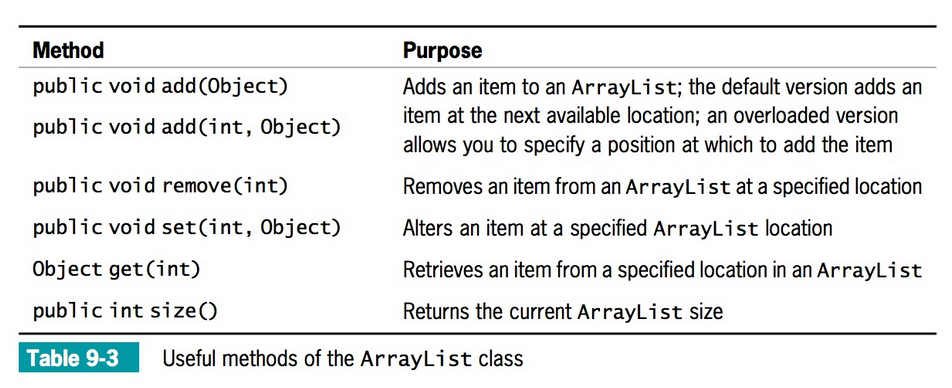
import java.util.ArrayList;

### ArrayList Declaration

ArrayList<datatype> arrayName = new ArrayList<datatype>();

Notice that no “length” is specified because size is dynamically resizable. By default, the ArrayList’s capacity is set to 10 items. You can specify the number of items or leave

ArrayList also comes with some userful methods.



## Enumerations

Enumeration special data type that enables for a variable to be a set of predefined constants. This features allows you to create your own custom data type with a set of finite values. Enumeration works similar to how a Class works. In fact, classes are derivitives of enumerations. Thus, enumeration is a special type of class in Java.

**Enumeration Declaration**

* Enumeration must be declared in the class space only.
* Use the keyword “enum” follow by a variable and a set of curly braces:  
    
  enum Variable{ };
* Each element in the enum is a constant

You should use enum types any time you need to represent a fixed set of constants such as directions, days of week, months, etc.

Instead of declaring these constants:  
  
final String JAN = “JAN”;

final String FEB = “FEB”;

final String MAR = “MAR”;

final String APR = “APR”;

final String MAY = “MAY”;

final String JUN = “JUN”;

final String JUL = “JUL”;

final String AUG = “AUG”;

final String SEP = “SEP”;

final String OCT = “OCT”;

final String NOV = “NOV”;

final String DEC = “DEC”;

Use enum to declare a custom data type called Month with some constants (identifiers).

enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

You can then use Month as a data type:

Month specialMonth; //Enumeration is a special type of a class

Now, you can assign any of the constants to the variable “specialMonth” as follows

specialMonth = Month.JAN;

Sources:

1. Devadas , S. 6.006 Introduction to Algorithms: Lecture 3, Fall 2011. Retrieved March, 24, 2016, from<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec03.pdf>