Chapter 6

Arrays

- Array Basics
- Arrays in Classes and Methods
- Programming with Arrays and Classes
- Sorting Arrays
- Multidimensional Arrays

Motivation

- How to organize 100 Student objects?
- 100 different Student object names?
- Organize data for efficient access
  - Class: different data types in one object
  - Array: multiple objects of the same type

Overview

- An array
  - a single name for a collection of data values
  - all of the same data type
  - subscript notation to identify one of the values
- A carryover from earlier programming languages
- More than a primitive type, less than an object
  - like objects when used as method parameters and return types
  - do not have or use inheritance
- Accessing each of the values in an array
  - Usually a for loop

Creating Arrays

- General syntax for declaring an array:
  ```java
  Base_Type[] Array_Name = new Base_Type[Length];
  ```
- Examples:
  - 80-element array with base type char:
    ```java
    char[] symbol = new char[80];
    ```
  - 100-element array of doubles:
    ```java
    double[] reading = new double[100];
    ```
  - 70-element array of Species:
    ```java
    Species[] specimen = new Species[70];
    ```

Three Ways to Use [ ] (Brackets) with an Array Name

1. Declaring an array: int[] pressure
   - creates a name of type "int array"
   - types int and int[] are different
     - int[] type of the array
     - int : type of the individual values
2. To create a new array, e.g. pressure = new int[100];
3. To refer to a specific element in the array
   - also called an indexed variable, e.g.
     ```java
     pressure[3] = keyboard.nextInt();
     System.out.println("You entered" + pressure[3]);
     ```

Some Array Terminology

- Array name
- Index - also called a subscript
  - must be an int
  - or an expression that evaluates to an int
- Indexed variable - also called an element or subscripted variable
- Value of the indexed variable - also called an element of the array

Note that "element" may refer to either a single indexed variable in the array or the value of a single indexed variable.
Array Length

- Specified by the number in brackets when created with `new`
  - `maximum` number of elements the array can hold
  - storage is allocated whether or not the elements are assigned values
- the attribute `length`
  ```java
  Species[] entry = new Species[20];
  System.out.println(entry.length);
  ```
- The `length` attribute is established in the declaration and cannot be changed unless the array is redeclared.

Subscript Range

- Array subscripts use zero-numbering
  - the first element has subscript 0
  - the second element has subscript 1
  - etc. - the `n`th element has subscript `n-1`
- The last element has subscript `length-1`
- For example: an int array with 4 elements

<table>
<thead>
<tr>
<th>Subscript</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>97</td>
<td>86</td>
<td>92</td>
<td>71</td>
</tr>
</tbody>
</table>

Subscript out of Range Error

- Using a subscript larger than `length-1` causes a `run time` (not a compiler) error
  - an `ArrayOutOfBoundsException` is thrown
    - you do not need to catch it
    - you need to fix the problem and recompile your code
- Other programming languages, e.g. C and C++, do not even cause a run time error!
  - one of the most dangerous characteristics of these languages is that they allow out of bounds array indices.

Array Length Specified at Run-time

// array length specified at compile-time
int[] array1 = new int[10];

// array length specified at run-time
// calculate size...
int size = ...;
int[] array2 = new int[size];

Programming Tip: Use Singular Array Names

- Using singular rather than plural names for arrays improves readability
- Although the array contains many elements the most common use of the name will be with a subscript, which references a `single` value.
- It is easier to read:
  - `score[3]` than
  - `scores[3]`

Initializing an Array's Values in Its Declaration

- can be initialized by putting a comma-separated list in braces
- Uninitialized elements will be assigned some default value, e.g. 0 for int arrays (explicit initialization is recommended)
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:
  ```java
  double[] reading = {5.1, 3.02, 9.65};
  System.out.println(reading.length);
  ```
  - displays 3, the length of the array `reading`
Initializing Array Elements in a Loop

• A for loop is commonly used to initialize array elements

  For example:

  ```
  int i;//loop counter/array index
  int[] a = new int[10];
  for(i = 0; i < a.length; i++)
  a[i] = 0;
  ```

  – note that the loop counter/array index goes from 0 to length - 1
  – it counts through length = 10 iterations/elements using the zero-numbering of the array index

Programming Tip:
Do not count on default initial values for array elements
  – explicitly initialize elements in the declaration or in a loop

Arrays and Array Elements as Method Arguments

• Arrays and array elements can be
  – used with classes and methods just like other objects
  – be an argument in a method
  – returned by methods

Arrays, Classes, and Methods

An array of a class can be declared and the class’s methods applied to the elements of the array:

```java
public void getFigures()
{
  System.out.println("Enter number of sales associates:");
  numberOfAssociates = SavitchIn.readLineInt();
  SalesAssociate[] record =
  new SalesAssociate[numberOfAssociates];
  for (int i = 0; i < numberOfAssociates; i++)
  {
    record[i] = new SalesAssociate();
    System.out.println("Enter data for associate " + (i + 1));
    record[i].readInput();
  }
}
```
Passing Array Elements

```java
int[] grade = new int[10];
obj.method(grade[i]); // grade[i] cannot be changed

Person[] roster = new Person[10];
obj.method(roster[i]); // roster[i] can be changed
```

### Array Names as Method Arguments

- Use just the array name and no brackets
- Pass by reference
  - the method has access to the original array and can change the value of the elements
- The length of the array passed can be different for each call
  - when you define the method you do not need to know the length of the array that will be passed
  - use the `length` attribute inside the method to avoid `ArrayIndexOutOfBoundsException`

#### Example: An Array as an Argument in a Method Call

```java
public static void showArray(char[] a)
{
    int i;
    for(i = 0; i < a.length; i++)
        System.out.println(a[i]);
}
```

```java
char[] grades = new char[45];
MyClass.showArray(grades);
```

### Arguments for the Method `main`

- The heading for the `main` method shows a parameter that is an array of Strings:
  ```java
  public static void main(String[] arg)
  ```
- When you run a program from the command line, all words after the class name will be passed to the main method in the arg array:
  ```java
  java TestProgram Josephine Student
  ```
- The following `main` method in the class `TestProgram` will print out the first two arguments it receives:
  ```java
  public static void main(String[] arg)
  {
      System.out.println("Hello " + arg[0] + " " + arg[1]);
  }
  ```
- The output from the command line above will be:
  ```java
  Hello Josephine Student
  ```

#### Using = with Array Names:

Remember They Are Reference Types

```java
int[] a = new int[3];
int[] b = new int[3];
for(int i=0; i < a.length; i++)
    a[i] = i;
b = a;
System.out.println(a[2] + " " + b[2]);
```

This does not create a copy of array `a`; it makes `b` another name for array `a`.

```java
System.out.println(a[2] + " " + b[2]);
```

A value changed in `a` is the same value obtained with `b`.

#### Using == with array names:

Remember they are reference types

```java
int i;
int[] a = new int[3];
int[] b = new int[3];
for(i=0; i < a.length; i++)
    a[i] = i;
for(i=0; i < b.length; i++)
    b[i] = i;
if(b == a)
    System.out.println("a equals b");
else
    System.out.println("a does not equal b");
```

`a` and `b` are both 3-element arrays of `ints`.

All elements of `a` and `b` are assigned the value `i`.

Tests if the addresses of `a` and `b` are equal, not if the array values are equal.

The output for this code will be "a does not equal b" because the addresses of the arrays are not equal.
Behavior of Three Operations

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Class Type</th>
<th>Entire Array</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment ((=))</td>
<td>Copy content</td>
<td>Copy address</td>
<td>Copy address</td>
</tr>
<tr>
<td>Equality (==)</td>
<td>Compare content</td>
<td>Compare address</td>
<td>Compare address</td>
</tr>
<tr>
<td>Parameter Passing</td>
<td>Pass by value (content)</td>
<td>Pass by reference (address)</td>
<td>Pass by reference (address)</td>
</tr>
</tbody>
</table>

Testing Two Arrays for Equality

- To test two arrays for equality you need to define an `equals` method that returns true if and only if the arrays have the same length and all corresponding values are equal.

```java
public static boolean equals(int[] a, int[] b) {
    boolean match = false;
    if (a.length == b.length) {
        match = true; // tentatively
        int i = 0;
        while (match && (i < a.length)) {
            if (a[i] != b[i])
                match = false;
            i++;
        }
    }
    return match;
}
```

Methods that Return an Array

- the address of the array is passed
- The local array name within the method is just another name for the original array

```
public class returnArrayDemo {
    public static void main(String arg[]) {
        char[] c = vowels();
        for(int i = 0; i < c.length; i++)
            System.out.println(c[i]);
    }

    public static char[] vowels() {
        char[] newArray = new char[5];
        newArray[0] = 'a';
        newArray[1] = 'e';
        newArray[2] = 'i';
        newArray[3] = 'o';
        newArray[4] = 'u';
        return newArray;
    }
}
```

Wrapper Classes for Arrays

- Arrays can be made into objects by creating a wrapper class
  - similar to wrapper classes for primitive types
- In the wrapper class:
  - make an array an attribute
  - define constructors
  - define accessor methods to read and write element values and parameters
- The text shows an example of creating a wrapper class for an array of objects of type `OneWayNoRepeatsList`
  - the wrapper class defines two constructors plus the following methods: `addItem`, `full`, `empty`, `entryAt`, `atLastEntry`, `onList`, `maximumNumberOfEntries`, `numberOfEntries`, and `eraseList`

Partially Filled Arrays

- Sometimes only part of an array has been filled with data
- Array elements always contain something
  - elements which have not been written to
  - contain unknown (garbage) data so you should avoid reading them
- There is no automatic mechanism to detect how many elements have been filled
  - you, the programmer need to keep track!
- An example: the instance variable `countOfEntries` in the class `OneWayNoRepeatsList` is incremented every time `addItem` is called (see the text)

```
public class OneWayNoRepeatsList {
    private char[] entry;
    private int countOfEntries;
    // constructor, methods...
    void addItem(char item) {
        if (countOfEntries < maximumNumberOfEntries) { // check if full
            entry = resize(entry); // resize if necessary
            entry[countOfEntries] = item;
            countOfEntries++;
        }
    }
}
```

Example of a Partially Filled Array

```
entry[0] Buy milk.
entry[1] Call home.
entry[3] 
entry[4] 
```

- `countOfEntries` has a value of 3.
- `entry.length` has a value of 5.
Searching an Array

- There are many techniques for searching an array for a particular value
  - **Sequential search**
    - start at the beginning of the array and proceed in sequence until either the value is found or the end of the array is reached*
    - if the array is only partially filled, the search stops when the last meaningful value has been checked
    - it is not the most efficient way
    - but it works and is easy to program
  - *Or, just as easy, start at the end and work backwards toward the beginning

Gotcha: Returning an Array Attribute (Instance Variable)

- Access methods that return references to array instance variables cause problems for information hiding.
  - Example: class –
    ```java
    { 
    private String[] entry;
    public String[] getEntryArray()
     { return entry; }
    }
    ```
  - Even though `entry` is declared private, a method outside the class can get full access to it by using `getEntryArray`.
  - In most cases this type of method is not necessary anyway.
  - If it is necessary, make the method return a copy of the array instead of returning a reference to the actual array.

Sorting an Array

- Sorting a list of elements is another very common problem (along with searching a list)
  - sort numbers in ascending order
  - sort numbers in descending order
  - sort strings in alphabetic order
  - etc.
  - There are many ways to sort a list, just as there are many ways to search a list
  - **Selection sort**
    - one of the easiest
    - not the most efficient, but easy to understand and program

Selection Sort Algorithm for an Array of Integers

To sort an array on integers in ascending order:
1. Find the smallest number and record its index
2. swap (interchange) the smallest number with the first element of the array
   - the sorted part of the array is now the first element
   - the unsorted part of the array is the remaining elements
3. repeat Steps 2 and 3 until all elements have been placed
   - each iteration increases the length of the sorted part by one

Selection Sort Example

key:
- yellow: smallest remaining value
- blue: sorted elements

Problem: sort this 10-element array of integers in ascending order:
```
40 81 11 17 5 19 30 14
```
1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]
```
3 6 11 17 40 5 19 30 14
```
2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6]
```
3 5 6 11 17 7 15 19 30 14
```
How many iterations are needed?
**Example: Selection Sort**

- Notice the precondition: every array element has a value
- may have duplicate values
- broken down into smaller tasks
  - "find the index of the smallest value"
  - "interchange two elements"
  - **private** because they are helper methods (users are not expected to call them directly)

```java
// Precondition:
* Every indexed variable of the array a has a value.
* Action: Sorts the array a so that
* a[0] <= a[1] <= ... <= a[a.length - 1].

public static void sort(int[] a) {
    int index, indexOfNextSmallest;
    for (index = 0; index < a.length - 1; index++) {
        // Place the correct value in a[index]:
        indexOfNextSmallest = indexOfSmallest(index, a);
        interchange(index, indexOfNextSmallest, a);
        // a[0] <= a[1] <= ... <= a[index] and these are
        // the smallest of the original array elements.
        // The remaining positions contain the rest of
        // the original array elements.
    }
}
```

**Selection Sort Code**

**Insertion Sort**

- Basic Idea:
  - Keeping expanding the sorted portion by one
  - Insert the next element into the right position in the sorted portion
- Algorithm:
  1. Start with one element [is it sorted?] – sorted portion
  2. While the sorted portion is not the entire array
     1. Find the right position in the sorted portion for the next element
     2. Insert the element
     3. If necessary, move the other elements down
     4. Expand the sorted portion by one

**Insertion Sort: An example**

- First iteration:
  - Before: [5, 3, 4, 9, 2]
  - After: [3, 5, 4, 9, 2]
- Second iteration:
  - Before: [3, 5, 4, 9, 2]
  - After: [3, 4, 5, 9, 2]
- Third iteration:
  - Before: [3, 4, 5, 9, 2]
  - After: [3, 4, 3, 5, 9]
- Fourth iteration:
  - Before: [3, 4, 5, 9, 2]
  - After: [2, 3, 4, 5, 9]

**Bubble Sort**

- Basic Idea:
  - Expand the sorted portion one by one
  - “Sink” the largest element to the bottom after comparing adjacent elements
  - The smaller items “bubble” up
- Algorithm:
  - While the unsorted portion has more than one element
    - Compare adjacent elements
    - Swap elements if out of order
    - Largest element at the bottom, reduce the unsorted portion by one

**Bubble Sort: An example**

- First iteration:
  - [5, 3, 4, 9, 2] → [3, 5, 4, 9, 2]
  - [3, 5, 4, 9, 2] → [3, 4, 5, 9, 2]
  - [3, 4, 5, 9, 2] → [3, 4, 5, 2, 9]
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
- Second iteration:
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
- Third iteration:
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
- Fourth iteration:
  - [3, 4, 5, 2, 9] → [3, 4, 5, 2, 9]
How to Compare Algorithms in Efficiency (speed)

- Empirical Analysis
  - Wall-clock time
  - CPU time
  - Can you predict performance before implementing the algorithm?

- Theoretical Analysis
  - Approximation by counting important operations
  - Mathematical functions based on input size \( N \)

### How Fast/Slow Can It Get?
(10G Hz, assume 10\(^{10}\) operations/sec)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \log_2 N )</th>
<th>( N^2 )</th>
<th>( 2^N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>33</td>
<td>100</td>
<td>1,024</td>
</tr>
<tr>
<td>10(^6)</td>
<td>664</td>
<td>10,000</td>
<td>1.3 x 10(^{10}) (4 x10(^{12}) years)</td>
</tr>
<tr>
<td>1,000</td>
<td>9,966</td>
<td>1,000,000</td>
<td>Forever??</td>
</tr>
<tr>
<td>10,000</td>
<td>132,877</td>
<td>100,000,000</td>
<td>Eternity??</td>
</tr>
</tbody>
</table>

### Theoretical Analysis (Sorting)

- Counting important operations
  - Comparisons (array elements)
    - \( \gg, \ll, \ldots \)
  - Swaps/moves (array elements)
    - 1 swap has 3 moves
  - Comparison is the more important operation—could be expensive
  - Size of input \( \langle N \rangle \) = Number of array elements
- Three cases for analysis
  - Worst case (interesting, popular analysis)
  - Best case (not so interesting)
  - Average case (discussed in another course)

### Selection Sort

- Comparisons
  - \( N \) – 1 iterations
    - First iteration: how many comparisons?
    - Second iteration: how many comparisons?
    - \( (N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2 = (N^2 - N)/2 \)
  - Moves (worst case: every element is in the wrong location)
    - \( N \) – 1 iterations
    - First iteration: how many swaps/moves?
    - Second iteration: how many swaps/moves?
    - \( (N - 1) \times 3 = 3N - 3 \)

### Insertion Sort

- Comparisons (worst case: correct order)
  - \( N \) – 1 iterations
    - First iteration: how many comparisons?
    - Second iteration: how many comparisons?
    - \( 1 + 2 + \ldots + (N - 2) + (N - 1) = N(N-1)/2 = (N^2 - N)/2 \)
  - Moves (worst case: reverse order)
    - \( N \) – 1 iterations
    - First iteration: how many moves?
    - Second iteration: how many moves?
    - \( 3 + 4 + \ldots + N + (N + 1) = (N + 4)(N + 1)/2 = (N^2 + 3N + 4)/2 \)

### Bubble Sort

- Comparisons
  - \( N \) – 1 iterations
    - First iteration: how many comparisons?
    - Second iteration: how many comparisons?
    - \( (N - 1) + (N - 2) + \ldots + 2 + 1 = N(N-1)/2 = (N^2 - N)/2 \)
  - Moves (worst case: reverse order)
    - \( N \) – 1 iterations
    - First iteration: how many swaps/moves?
    - Second iteration: how many swaps/moves?
    - \( (N - 1) \times 3 = 3N - 3 \)
Summary of Worst-case Analysis

<table>
<thead>
<tr>
<th></th>
<th>Comparisons</th>
<th>Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$(N^2 - N)/2$</td>
<td>$3N - 3$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$(N^2 - N)/2$</td>
<td>$(N^2 + 3N - 4)/2$</td>
</tr>
<tr>
<td>Bubble</td>
<td>$(N^2 - N)/2$</td>
<td>$(3N^2 - 3N)/2$</td>
</tr>
</tbody>
</table>

Sorting Algorithm Tradeoffs

- Easy to understand algorithms
  - not very efficient
  - less likely to have mistakes
  - require less time to code, test, and debug
  - Selection, Insertion, Bubble Sorting algorithms
  - Bubble Sort is the easiest to implement
- Complicated but more efficient
  - useful when performance is a major issue
  - programming project for Chapter 11 describes a more efficient sorting algorithm

"Getting the wrong result is always inefficient."

Multidimensional Arrays

- Arrays with more than one index
  - number of dimensions = number of indexes
- Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays
  - A 2-D array corresponds to a table or grid
    - one dimension is the row
    - the other dimension is the column
    - cell: an intersection of a row and column
    - an array element corresponds to a cell in the table

Table as a 2-Dimensional Array

- The table assumes a starting balance of $1000
- First dimension: row identifier - Year
- Second dimension: column identifier - percentage
- Cell contains balance for the year (row) and percentage (column)
- Balance for year 4, rate 7.00% = $1311

Balances for Various Interest Rates

<table>
<thead>
<tr>
<th>Year</th>
<th>5.00%</th>
<th>5.50%</th>
<th>6.00%</th>
<th>6.50%</th>
<th>7.00%</th>
<th>7.50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1050</td>
<td>$1055</td>
<td>$1060</td>
<td>$1065</td>
<td>$1070</td>
<td>$1075</td>
</tr>
<tr>
<td>2</td>
<td>$1103</td>
<td>$1113</td>
<td>$1124</td>
<td>$1134</td>
<td>$1145</td>
<td>$1156</td>
</tr>
<tr>
<td>3</td>
<td>$1158</td>
<td>$1174</td>
<td>$1191</td>
<td>$1208</td>
<td>$1225</td>
<td>$1242</td>
</tr>
<tr>
<td>4</td>
<td>$1216</td>
<td>$1239</td>
<td>$1262</td>
<td>$1286</td>
<td>$1311</td>
<td>$1335</td>
</tr>
<tr>
<td>5</td>
<td>$1276</td>
<td>$1307</td>
<td>$1338</td>
<td>$1370</td>
<td>$1403</td>
<td>$1436</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Java Code to Create a 2-D Array

- Syntax for 2-D arrays is similar to 1-D arrays
- Declare a 2-D array of ints named table
  - the table should have ten rows and six columns
  ```java
  int[][] table = new int[10][6];
  ```
Method to Calculate the Cell

- balance(starting, years, rate) = (starting) x (1 + rate)^years
- The repeated multiplication by (1 + rate) can be done in a for loop that repeats years times.

```java
public static int balance(double startBalance, int years, double rate) {
    double runningBalance = startBalance;
    int count;
    for (count = 1; count <= years; count++)
        runningBalance = runningBalance*(1 + rate/100);
    return (int) (Math.round(runningBalance));
}
```

Processing a 2-D Array: for Loops Nested 2-Deep

- To process all elements of an n-D array nest n for loops
  - each loop has its own counter that corresponds to an index
- For example: calculate and enter balances in the interest table
  - inner loop repeats 6 times (six rates) for every outer loop iteration
  - the outer loop repeats 10 times (10 different values of years)
  - so the inner repeats 10 x 6 = 60 times = # cells in table

```java
int[][] table = new int[10][6];
int row, column;
for (row = 0; row < 10; row++)
    for (column = 0; column < 6; column++)
        table[row][column] = balance(1000.00, row + 1, (5 + 0.5*column));
```

Multidimensional Array Parameters and Returned Values

- Methods may have multi-D array parameters
- Methods may return a multi-D array as the value returned
- The situation is similar to 1-D arrays, but with more brackets
- Example: a 2-D int array as a method argument

```java
public static void showTable(int[][] displayArray) {
    int row, column;
    for (row = 0; row < displayArray.length; row++)
        for (column = 0; column < displayArray[row].length; column++)
            System.out.print($displayArray[row][column] + "  ");
    System.out.println();
}
```

Ragged Arrays

- Ragged arrays have rows of unequal length
  - each row has a different number of columns, or entries
- Ragged arrays are allowed in Java
- Example: create a 2-D int array named b with 5 elements in the first row, 7 in the second row, and 4 in the third row:

```java
int[][] b = new int[3][];
b[0] = new int[5];
b[1] = new int[7];
b[2] = new int[4];
```
Nested Loops with Multidimensional Arrays

```java
for (employeeNumber = 1; 
   employeeNumber <= numberOfEmployees; employeeNumber++) 
{  // Process one employee
    sum = 0;
    for (dayNumber = 0; dayNumber < 5; dayNumber++) 
      sum = sum + hours[dayNumber][employeeNumber - 1];
    weekHours[employeeNumber - 1] = sum;
}
```

- The method `computeWeekHours` uses nested for loops to compute the week's total hours for each employee.
- Each time through the outer loop body, the inner loop adds all the numbers in one column of the `hours` array to get the value for one element in the `weekHours` array.

Parallel Arrays

```java
public class Course
{
  private String   _name;
  private String[] _studentName;
  private int[]    _studentId;
  private float[]  _studentGrade;
  private String[] _assignmentName;  // parallel array?
  public Course(String name, int numOfStudents)
  {  _name = name;
      _studentName = new String[numOfStudents];
      _studentId = new int[numOfStudents];
      _studentGrade = new float[numOfStudents];
      for (int i = 0; i < numOfStudents; i++)
        _studentName[i] = "none";
        _studentId[i] = 0;
        _studentGrade[i] = 0.0;
  }
}
```

Summary

Part 1
- An array may be thought of as a collection of variables, all of the same type.
- An array is also may be thought of as a single object with a large composite value of all the elements of the array.
- Arrays are objects created with `new` in a manner similar to objects discussed previously.

Part 2
- Array indexes use zero-numbering:
  - They start at 0, so index `i` refers to the `(i+1)th` element;
  - The index of the last element is `(length-of-the-array - 1)`. Any index value outside the valid range of 0 to length-1 will cause an `array index out of bounds` error when the program runs.
- A method may return an array.
- A "partially filled array" is one in which values are stored in an initial segment of the array:
  - use an `int` variable to keep track of how many variables are stored.

Part 3
- An array element can be used as an argument to a method any place the base type is allowed:
  - if the base type is a primitive type, the method cannot change the array element;
  - if the base type is a class, the method can change the array element.
- When you want to store two or more different values (possibly of different data types) for each index of an array,
  - parallel arrays (multiple arrays of the same length)
  - use a class that have multiple types/values.
- An accessor method that returns an array corresponding to a private instance variable of an array type should be careful to return a copy of the array, and not return the private instance variable itself (like any object).
Summary
Part 3

- Sorting algorithms
  - Selection
  - Insertion
  - Bubble

- Analysis
  - Empirical
  - Theoretical
    - Comparisons: Quadratic-time ($N^2$) algorithms

Summary
Part 4

- Arrays can have more than one index.
- Each index is called a dimension.
- Hence, multidimensional arrays have multiple indexes.
  - e.g. an array with two indexes is a two-dimensional array.
- A two-dimensional array can be thought of as a grid or table with rows and columns:
  - one index is for the row, the other for the column.
- Multidimensional arrays in Java are implemented as arrays of arrays.
  - e.g. a two-dimensional array is a one-dimensional array of one-dimensional arrays.