COSC 1P03

- Audience
  - planning to major in COSC (prereq. 1P02 60%)
- Web
  - COSC 1P03 Brightspace site
  - COSC: http://www.cosc.brocku.ca/
- Course Outline
  - Textbook
  - Software
  - Marking scheme
  - **Homework** submission/return
- Coordinator: Yathusan: yt19tk@brocku.ca (All course-related questions)
- Succeeding in COSC 1P03
  - labs/homework

Chapter 1
Arrays

Objectives

- Explain the difference between one-dimensional, two-dimensional and higher dimensional arrays.
- Describe the two standard traversal patterns for two-dimensional arrays—row-major and column-major.
- Explain how arrays may be parameters to and results of methods.
- Choose and apply arrays for representing information when appropriate.
- Choose and apply the appropriate technique for storing information in arrays—right-sized and variable-sized.
- Apply appropriate array traversal in solving a problem.

Arrays

- Collection
  - like `Pictures` & `Sounds`
  - difference from Collections: Not defined as a class but built-in
- Elements (components)
  - any type (primitive or object)
- Non-sequential processing
  - not just single pass in order of occurrence
  - selective (random) processing

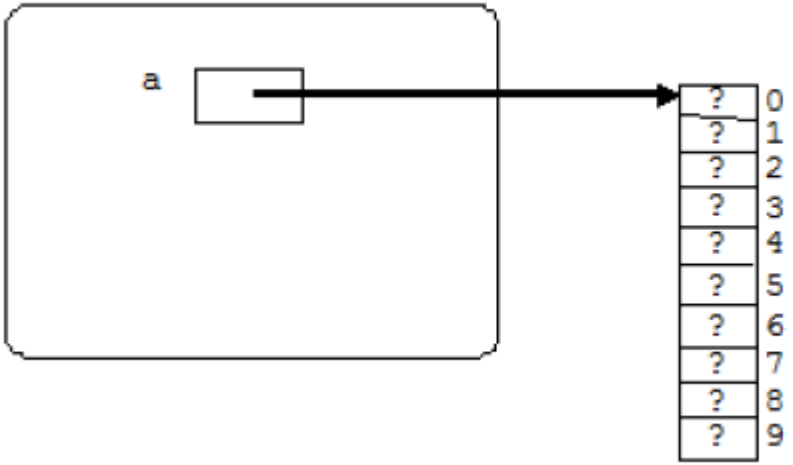
Arrays in Java

- An array is a collection of items all of the same type.
- The individual items are called **elements** and their type is the **element type.**
- Declared using:
  - `type[]..name;`
  - e.g., `int [] a; //one dimensional array of ints called a`
- Arrays are reference types (variable is a reference to the actual array)
- Array is *created* with:
  - `name= new type[expression]`
  - e.g., `a=new int[10];`
- **Deceptively similar to object creation!**
- May be one-dimensional, two-dimensional or have more dimensions. The # of pairs of brackets after the type is the number of dimensions.

Memory Model

```
a=new int[10];
```

- Storage capable of storing 10 `int` values is allocated
- Then, its reference is stored in `a`.
- The value of each individual element (`int`s) is not (yet) specified.



Flexibility of Arrays (On Assigning)

- Assignment compatibility for arrays requires that the array being assigned (right-hand side) have the same number of dimensions and the same element type as the variable (left-hand side)
- Example 1:
  - `a = new int[10];  b = new int[9]`
  - `a = b`

  Allowed?
- Example 2:
  - `a = new int[10]; b = new String[10]`
  - `a = b`

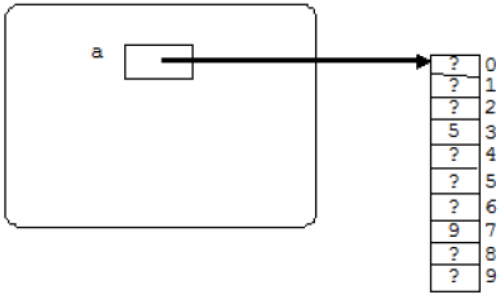
  Allowed?
- Example 2:
  - `a = new int[10];`

    `b =  new int[10] [10];`
  - `a = b`

  Allowed?

Array Operations

- Arrays may be compared for equality using == and != using **reference equality**:
  - "Do the expressions reference the same array?"
    - a = new int[10];
    - b = new int[10];
    - //a = b;
    - a == b //true or false?



- Length attribute: `array.length`
  - `a.length //10`
- `Accessing/setting array elements:`
  - `name[expression]`
  - `name` must be correct variable reference; `NullPointerException`
  - `expression` should not exceed length of the corresponding dimension of array minus 1;`ArrayIndexOutOfBoundsException`
  - expression is the index
  - `name[expression] = element`
  - e.g., `a[0]=1.`

When to use arrays

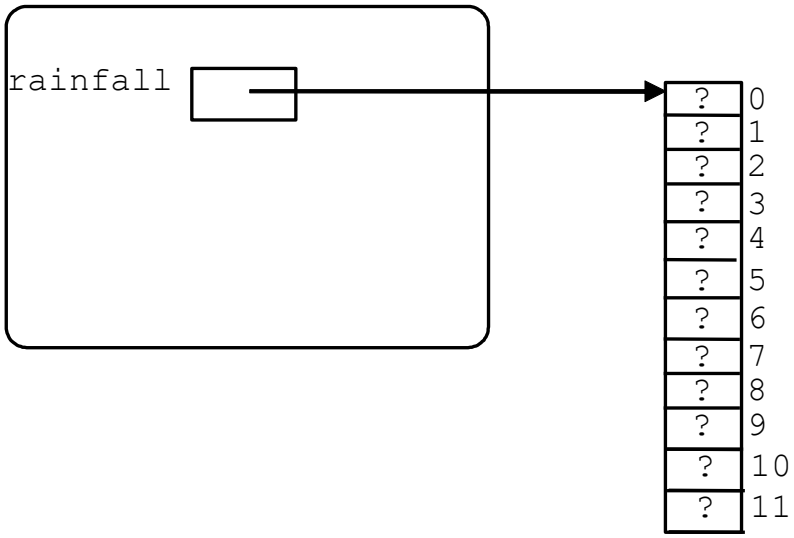
- Arrays can be used whenever we have a collection of related items of the same type such as marks on a test or students in a course.
- They are necessary whenever the collection must be processed in **non-sequential order**:
  - all the processing required cannot be done when the item is first encountered
- Array processing involves:
  - (1) the declaration of an array variable, e.g., `int [] a`
  - (2) the creation of the array, e.g., `a=new int[10]`
    - **"right-sized" arrays** (when the size is known)
    - **"variable-sized"** arrays (size is not known)
  - (3) initialization of some or all elements, e.g., `a[0]=1`
  - (4) processing of some or all elements. `a[1]=a[0]+1`
    - may involve going through each element, traversal – can be done with for loop (in Java indices start from 0).

Right-sized arrays example: Above-average rainfall

- Months with rainfall above monthly average for year
  - 12 data values Jan - Dec
- Non-sequential processing
  - must process all data to compute average
  - then process the data to list those above average
- Array
  - array of 12 `double` values
  - declare then create
  - memory model
  - indexing
    - like accessing a `Sample` of a `Sound`

"An algorithm that involves two or more processes, whether those processes are concurrent or parallel at times, is said to be *non-sequential*."

```
double[]   rainfall;

rainfall = new double[12];
```




```
totRain = totRain + rainfall[i];
```

Code

- Computing average
  - algorithm
  - code
- List those above average
  - algorithm
  - code

```
repeat for each data value
    read value into next element
    add value to total
compute average
```

```
repeat for each element
    if value > average
        display value
```

Iterative For-Loop vs For-Each Loop

Iterative:

```
for ( int index=0 ; index<name.length ; index++ ) {
        :
    process name[index]
        :
};
```

For Each:

```
for ( type elt : name ) {
        :
    process elt
        :
};
```

Variable-sized Arrays: Class Statistics

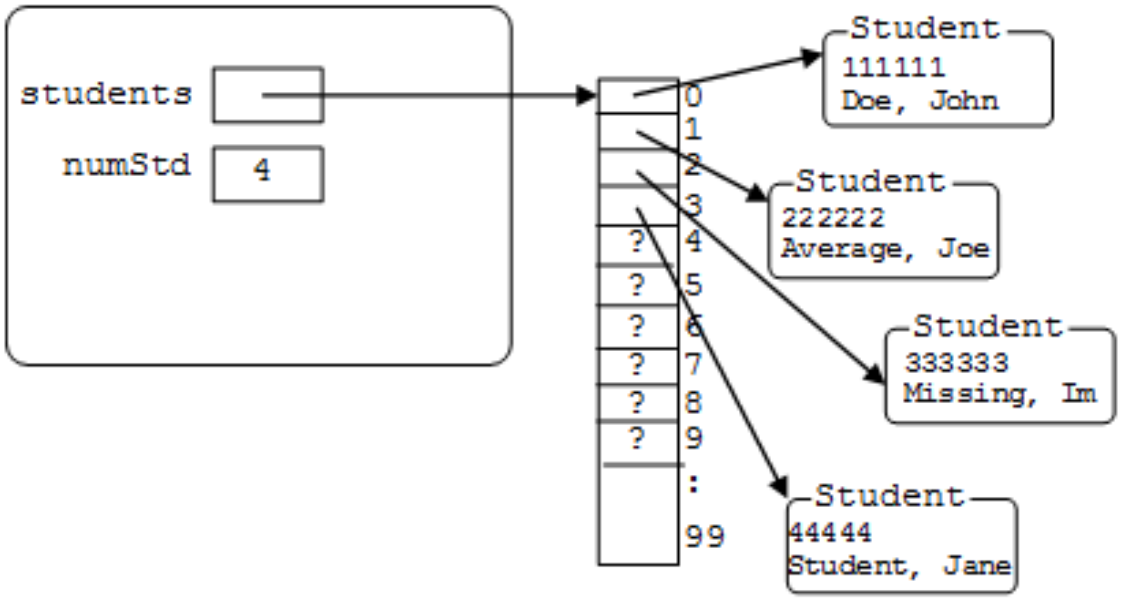
- Report with average and standard deviation of student marks
  - data file
- Standard Deviation formula
  - Non sequential (2 passes)
- Array of `Student` objects
  - array size not known
    - choose suitable size (constant)
    - keep count
  - array organization
    - not all elements used

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

How many passes are needed?

```
private static final int MAX_STD = 100;
        :
Student[]  students;
int        numStd;

        :

students = new Student[MAX_STD];
```

```
public class Student {
      :
    public Student ( ASCIIDataFile from ) {…
      :
    public String getStNum ( ) {…
      :
    public String getName ( ) {…
      :
    public double getTestMark ( ) {…
      :
}  // Student
```


students
numStd
4
0
1
2
3
?
4
?
5
?
6
?
7
?
8
?
9
:
99
Student
111111
Doe, John
Student
222222
Average, Joe
Student
333333
Missing, Im
Student
44444
Student, Jane

Code

- Reading data
  - termination
    - no more data (`EOF`)
    - no more space (`MAX_STD`)
      - error message?
  - accessing student data
- Computing standard deviation
  - array as parameter
    - like `Sound` as a parameter
    - accessing same array
  - also pass `nStd` as parameter
    - so method knows how many elements
  - second traversal

Multidimensional Arrays

- Often data is presented in tabular form (e.g., rainfall by month AND year, enrollment statistics across different universities and departments)
-

	Column 1	Column 2	Column 3	Column 4
Row 1	`x[0][0]`	`x[0][1]`	`x[0][2]`	`x[0][3]`
Row 2	`x[1][0]`	`x[1][1]`	`x[1][2]`	`x[1][3]`
Row 3	`x[2][0]`	`x[2][1]`	`x[2][2]`	`x[2][3]`

- Arrays of higher dimension can be declared, created and indexed by including additional dimensions as additional sets of brackets

University Statistics

- University enrolments at universities in a number of departments
  - produce summary table
- Data
  - Multi-dimensional
    - university
    - department
- Multi-dimensional array
  - like `Picture`
  - declaration
  - rows are departments
  - columns are universities

Enrolment Report
Jan 10, 2017

	Acadia	Brock	McMaster	Laurentian	Total
Math	162	15	237	35	449
Business	836	182	987	243	2,248
Comp. Sci.	263	91	321	110	785
Biology	743	432	648	0	1,823
French	42	59	117	57	275
Total	2,046	779	2,310	445	5,580

```
int       nUniv;
int       nDept;
int[][]   enrol;

      :

enrol = new int[nDept][nUniv];
```

	0	1	2	3
0	162	15	237	35
1	836	182	987	243
2	263	91	321	110
3	743	432	648	0
4	42	59	117	57

Right-sized vs Variable Multidimensional Arrays

```
int[][]  enrol;
      :
enrol = new int[5][4];
      :
… enroll[i][j]…
```

- Multidimensional arrays can be processed as right-sized arrays, or variable-sized arrays.
- When a right-sized two-dimensional array is used (like `enrol` above), the number of rows in the array is given by the length attribute (`enrol.length`).
- For any row, the number of columns in the row is given by the length attribute for that row (`enrol[2].length`).
- If a two-dimensional array is to be variable-sized, we would fill elements in the first rows and the first columns of each row—the top-left corner of the array.

Code

- Reading data
  - university & department names
  - enrolment data
    - like processing all `Pixels` of a `Picture`
- Compute department sums
  - method can return an array
    - like returning a `Picture`
  - processing by row (row-order)
- Computing university sums
  - processing by column (column-order)
- Compute grand total
- Produce summary table

Consolidation

Single-dimensional Arrays

- Collection like a `Sound`
- Use
  - declaration
  - creation
  - subscripting (indexing)
    - zero-based
    - `NullPointerException`
    - `ArrayIndexOutOfBoundsException`
  - `length` attribute
- Memory model
- Reference type

declaration

type [ ] *name*;

```
double[]   rainfall;
Student[]  students;
```

creation/construction

new *type* [*expr*]

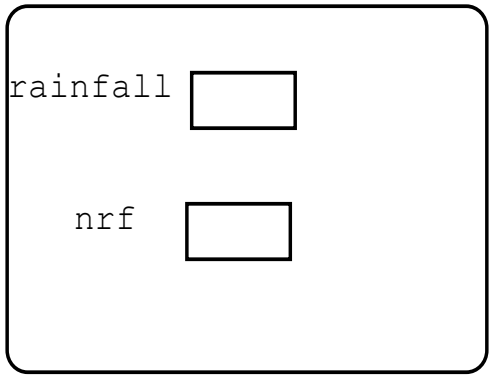
```
rainfall = new double[12];
students = new Student[MAX_STD];
```

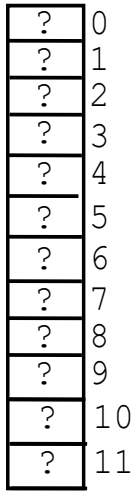
initialization

name [*expr*]

```
rainfall[i] = in.readDouble();
totRain = totRain + rainfall[i];

students[i] = aStudent;
sum = sum + students[i].getTestMark();
```

```
double[] rainfall;

double[] nrf;

rainfall = new double[12];

rainfall[5] = 5;

nrf = rainfall;

nrf[7] = rainfall[5];
```





Processing One-dimensional Arrays

- "Right-Sized"
  - size known *a priori* or computable
  - create with known size
  - all elements have values
  - traversal patterns
- "Variable-sized"
  - array size not known
    - choose arbitrary size (constant)
    - keep count in ancillary variable
  - not all elements have values
  - traversal pattern

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

```
for ( int i=0 ; i<rainfall.length ; i++ ) {
    totRain = totRain + rainfall[i];
};
```

```
for ( type e : a ) {
    process e
};
```

```
for ( int val : rainfall ) {
    totRain = totRain + val;
};
```

```
for ( int i=0 ; i<numberOfElements ; i++ ) {
    process a[i]
};
```

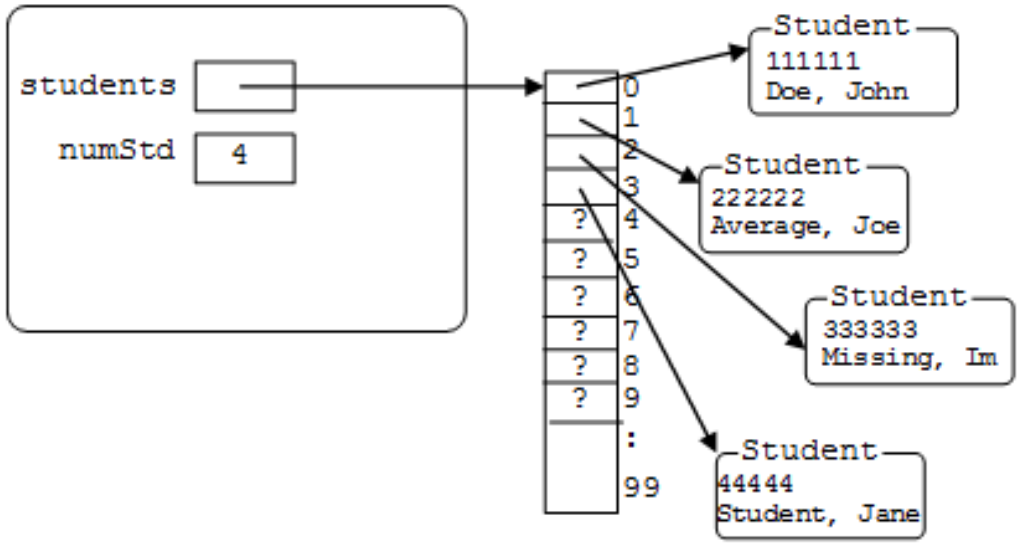
```
for ( int i=0 ; i<numStd ; i++ ) {
    total = total + students[i].getMark();
};
```

Arrays as Parameters

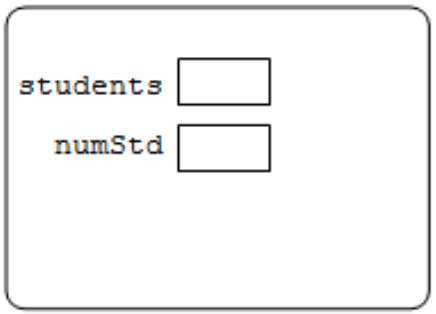
- Pass array as parameter
  - like passing a `Sound`
- Parameter is reference to same array as argument
- Change to element is change to element in argument

constructor

```
Student[] students;
int       numStd;
    :
std = computeStd(students,numStd,ave);
```




```
private double computeStd ( Student[] students, int numStd, double ave ) {
    :
    aMark = students[i].getTestMark();
```



Multidimensional Arrays

- 2 or more dimensions
  - e.g. table
  - like `Picture`
- Declaration
- Creation
- Subscripting
- `length`
- "Variable-sized"
  - fill upper left corner
  - one counter for each dimension

type [ ]… *name*;	int[][] enrol;
new *type* [*expr*]…	enrol = new int[nDept][nUniv];
name [*expr*]…	enrol[i][j] = … …enrol[i][j]…
name.length *name*[expr].length	…enrol.length… …enrol[i].length…

Processing 2-Dimensional Arrays

- Random access
  - `a[i][j]`
  - look-up table
- Sequential access
  - row-major order
    - lexicographic order
      - The order of access is: `a[0][0], a[0][1], a[0][2],… a[1][0], a[1][1], a[1][2],… a[2][0], a[2][1], a[2][2],….`
    - algorithm
  - column-major order
    - `a[0][0], a[1][0], a[2][0],… a[0][1], a[1][1], a[2][1],… a[0][2], a[1][2], a[2][2],….`
    - algorithm

row-major order

```
for ( int i=0 ; i<a.length ; i++ ) {
    preprocessing for row i
    for ( int j=0 ; j<a[i].length ; j++ ) {
        process a[i][j]
    };
    postprocessing for row i
```

```
for ( int i=0 ; i<stats.length ; i++ ) {
    sums[i] = 0;
    for ( int j=0 ; j<stats[i].length ; j++ ) {
        sums[i] = sums[i] + stats[i][j];
    };
};
```

column-major order

```
for ( int j=0 ; j<a[0].length ; j++ ) {
    preprocessing for column j
    for ( int i=0 ; i<a.length ; i++ ) {
        process a[i][j]
    };
    postprocessing for column j
};
```

```
for ( int j=0 ; j<stats[0].length ; j++ ) {
    sums[j] = 0;
    for ( int i=0 ; i<stats.length ; i++ ) {
        sums[j] = sums[j] + stats[i][j];
    };
};
```

Array Representation

- contiguous allocation
  - value/object/array is sequence of consecutive cells
- single dimensional
  - contiguous allocation
  - `address(a[i]) = address(a) + i × s`
    - where
      - `s` = size of element type
  - if not zero-based subscripting
    - `address(a[i]) = address(a) + (i-l) × s`
    - where
      - `l` = lower bound

- multi-dimensional
  - lexicographic (row-major) ordering
  - consider as array of rows
  - `address(a[i]) = address(a) + i × S`
    - where
      - `S` = size of row (`S=a[0].length × s`)
    - start of `i`th row
  - `address(a[i][j]) = address(a[i]) + j × s`
  - substitution gives
    - `address(a[i][j]) = address(a) + i × S + j × s`
  - for non-zero based
    - `address(a[i][j]) = address(a) + (i-`l_r`) × S + (j-`l_c`) × s`

Arrays of Arrays

- non-contiguous allocation
  - each row contiguous
- memory model
- addressing
  - `address(a[i][j]) = content(address(a)+i×4) + j × s`
- access
  - row-major order
- ragged array
  - creation

address of the i^{th} row

why 4?

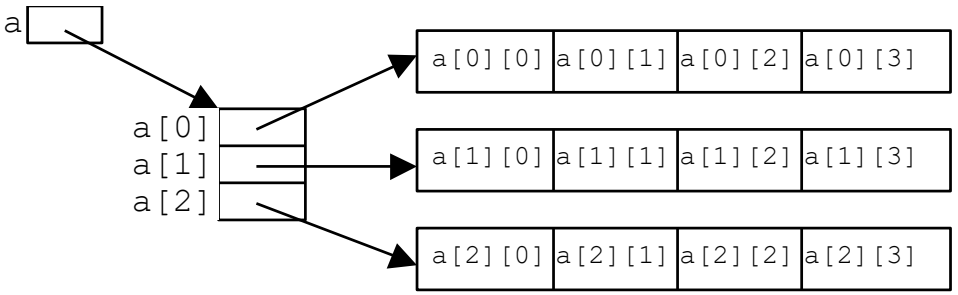


Figure 2.31 Array of array storage