Arrays

- 1-Dimensional Arrays
- 2-Dimensional Arrays

=> Read section 1.4
How would you store 100 exam scores?

- For example, to compute an average and then compare each score to the average.

More generally, how would you store \( n \) exam scores, where the value of \( n \) is unknown or might change?

An array is a data structure for storing and managing multiple variables (or objects) of the same type.
Array properties:

- Contains many data items, all of the same *base type*
- Has a *single name* for the collection of data values
- Uses *subscript notation* to identify individual data items

Loops, especially *for* loops, are used extensively with arrays.
Declaring an Array

- An array must be both declared and allocated before using it.

- Declaring an array:

  ```
  int[] A;
  ```

  creates an array called A whose elements are each of type int (the base type of the array).
Allocating an array:

```java
A = new int[10];
```

Individual locations in an array are *indexed* from 0 to n-1.
Allocating an Array

- Individual locations in an array are indexed from 0 to n-1.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

- The index, or subscript, of a location is used to access it.

  A[0] - first location
  A[1] - second location
  ...
  A[9] - tenth location
Individual elements in an array can be treated just like any other variable of the base type of the array.

```

System.out.print("Enter value: ");
System.out.println("You entered: " + A[3]);


int i = 2;
A[i] = 25;
```

Despite the funny syntax, `A[i]` is just another `int` variable.
The base type can be almost any type.

```java
char[] myChars;
float[] someFloats;
String[] names;
boolean[] indicators;

myChars = new char[100];
names = new String[25];
indicators = new boolean[50];
System.out.print(“Enter array length:”);
n = kb.nextInt();
someFloats = new float[n];   // Note length is a variable
names = new int[10];        // What’s wrong with this?
```
An array can be declared and allocated in one statement:

```java
// 10-element array with base type int:
int[] myInts = new int[10];

// 80-element array with base type char:
char[] symbol = new char[80];

// 100-element array of double:
double[] reading = new double[100];

// 70-element array of Species (object type):
Species[] specimen = new Species[70];
```
The length of an array is determined when allocation takes place.

The length of an array can be accessed programmatically using the attribute `length`:

```java
double[] entry = new double[20];
System.out.println(entry.length);
```

The `length` attribute cannot be changed.

The `length` attribute is frequently used in loops that process arrays.
Using a subscript larger than \texttt{length-1}, or less than 0, causes a \textit{run time} (not a compiler) error.

Other programming languages, e.g. C and C++, are not guaranteed to cause a run time error!

Such errors may turn up long after the code has been tested and released.
A for loop is commonly used to process array elements.

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

// Initialize all elements to random values; note the use of the variable i to index
for(int i = 0; i <= a.length-1; i++)
    a[i] = random.nextInt(50) + 1;  // Generates a random number in the range 1..50

// Output the contents of the array
for(int i = 0; i <= a.length-1; i++)
    System.out.println("Element " + i + " is: " + a[i]);

// Add 5 to every element of the array, in reverse order
for(int i = a.length-1; i >= 0; i--)
    a[i] = a[i] + 5;
```

Note that off-by-one errors are very common with arrays.
Array Example #1

Enter # of Grades: 4
Enter grade: 30
Enter grade: 10
Enter grade: 20
Enter grade: 40

The average is: 25.0
Student #1 is above average
Student #2 is below average
Student #3 is below average
Student #4 is above average
int[] grades;
int n, sum;
double avg;

// Input the number of grades from the user

// Allocate the array

// Input the grades into the array

// Add up all the grades

// Calculate and output the average

// Compare each grade to the average
int[] grades;
int n, sum;
double avg;

// Input the number of grades from the user
System.out.print("Enter # of Grades:");
n = kb.nextInt();

// Allocate the array

// Input the grades into the array

// Add up all the grades

// Calculate and output the average

// Compare each grade to the average
int[] grades;
int n, sum;
double avg;

// Input the number of grades from the user
System.out.print("Enter # of Grades:");
n = kb.nextInt();

// Allocate the array
grades = new int[n];

// Input the grades into the array

// Add up all the grades

// Calculate and output the average

// Compare each grade to the average
int[] grades;
int n, sum;
double avg;

// Input the number of grades from the user
System.out.print("Enter # of Grades:");
n = kb.nextInt();

// Allocate the array
grades = new int[n];

// Input the grades into the array
for (int i=0; i<=grades.length-1; i++) {
    System.out.print("Enter grade:");
    grades[i] = kb.nextInt();
}

// Add up all the grades

// Calculate and output the average

// Compare each grade to the average
int[] grades;
int n, sum;
double avg;

// Input the number of grades from the user
System.out.print("Enter # of Grades:");
n = kb.nextInt();

// Allocate the array
grades = new int[n];

// Input the grades into the array
for (int i=0; i<=grades.length-1; i++) {
    System.out.print("Enter grade:");
    grades[i] = kb.nextInt();
}

// Add up all the grades (could have been done in the above loop)
sum = 0;
for (int i=0; i<=grades.length-1; i++)
    sum = sum + grades[i];
// Calculate and output the average

// Compare each grade to the average
// Calculate and output the average
avg = (double)sum/n;
System.out.println();
System.out.println("The average is: " + avg);

// Compare each grade to the average
// Calculate and output the average
avg = (double)sum/n;
System.out.println();
System.out.println("The average is: " + avg);

// Compare each grade to the average
for (int i=0; i<=grades.length-1; i++)
    if (grades[i] > avg)
        System.out.println("Student #" + (i+1) + " is above average");
    else if (grades[i] < avg)
        System.out.println("Student #" + (i+1) + " is below average");
    else
        System.out.println("Student #" + (i+1) + " is average");
Enter # of students: 4
Enter name for student #1: smith
Enter exam #1 grade: 10
Enter exam #2 grade: 30
Enter name for student #2: jones
Enter exam #1 grade: 40
Enter exam #2 grade: 20
Enter name for student #3: brown
Enter exam #1 grade: 30
Enter exam #2 grade: 40
Enter name for student #4: carson
Enter exam #1 grade: 20
Enter exam #2 grade: 50

Exam #1 average: 25.0
Exam #2 average: 35.0

Student smith is below average on exam 1
Student smith is below average on exam 2
Student jones is above average on exam 1
Student jones is below average on exam 2
Student brown is above average on exam 1
Student brown is above average on exam 2
Student carson is below average on exam 1
Student carson is above average on exam 2

my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/parallelArrays.txt
Consider the following array declarations:

```java
int[] a, b, c;

// Suppose a and b (but not c) are allocated and loaded with integers.
```

Give a segment of Java code that allocates the array c, and stores in it the product of the corresponding elements from arrays a and b.

Questions:
- Will a and b have the same length?
- What if they don't?

Write the above code by hand first, then type it in, compile it, and run it.
Array Exercises

- Assume similar declarations and assumptions for each of the following

- Easy:
  1. Given two int arrays, construct a third that contains the maximum of the corresponding elements of the two given arrays.
  2. Given an int array, and a single integer, set all locations in the array to 0 that contain the given integer.
  3. Given two int arrays, output true if the two arrays are identical and false if they are not.

- Challenging:
  4. Given an int array, determine and output the largest value in the array.
  5. Same as the previous exercise, but also output a position where the maximum value was located.
  6. Given an int array, construct a second array that contains the contents of the first array in reverse order.
  7. Given an int array, reverse the contents of that array (how is this different from the previous exercise?).
**Difficult:**

8. Given an int array and an integer pos, where $0 <= pos <= a.length-1$, construct an array that is identical to given array, but with the value in position pos deleted (Note that the resulting array should have length 1 less than the given array).

9. Given an int array and two integers pos1 and pos2, where $0 <= pos1 <= pos2 <= a.length-1$, construct an array that is identical to given array, but with the values in position pos1 though pos2 deleted.

10. Given an int array, and a single integer, construct an array that is identical to the given array, but with all occurrences of the given integer deleted.

11. Given an int array, perform a “left-shift” of the array, i.e., each element in the array moves one position to the left. Note that the element in the first position “drops off” of the array, and the last element stays as is.

12. Modify your solution from the previous exercise to do a “circular-left-shift” of the array, i.e., each element of the array moves one position to the left, with the element in the first position moving to the last position.

13. Given an int array, plus two integers pos1 and pos2, where $0 <= pos1, pos2 <= a.length-1$ and pos1 <= pos2, perform a circular left shift all the values in the array between positions pos1 and pos2.

14. Given an int array, plus three integers pos1, pos2 and k, where $0 <= pos1, pos2 <= a.length-1$, pos1 <= pos2 and k >= 0, perform a circular left shift of all the values in the array k times, between positions pos1 and pos2.

15. Redo the last four exercises, but for a “right-shift.”

16. Given two int arrays intArray1 and intArray2, each containing no duplicates, construct another array that contains the set-difference of the two given arrays. In other words, all the integers that appear in intArray1 but not intArray2.

17. Same as the last one, but for set-intersection, and set-union.

*Note that for the exercises performing set operations (union, intersection, set-difference, and removing duplicates), determining the size of the resulting array is itself non-trivial, assuming the resulting array is to have length exactly equal to the number of elements in the result of the operation.*

➢ Redo any of the exercises, but for arrays of strings.
Partially Filled Arrays

- Sometimes only part of an array has been filled with data.

- Locations which have not been written to contain unknown, i.e., garbage data (not really, but this is a conservative assumption).

- There is no automatic mechanism to detect how many elements have been filled; the programmer needs to keep track!
**Example of a Partially Filled Array**

<table>
<thead>
<tr>
<th>entry[0]</th>
<th>Buy milk.</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry[1]</td>
<td>Call home.</td>
</tr>
</tbody>
</table>

- Garbage values are shown in entry[3] and entry[4].

- `countOfEntries` has a value of 3.
- `entry.length` has a value of 5.
Suppose that we want to input and process a list of student grades.

Suppose we don’t have the luxury of asking the user how many grades there will be, but we do know there will be at most 100.

Also suppose that we need to process the list several times:
  - Compute a class average
  - Compare each grade to the average
public static void main(String[] args) {
    int count, studentGrade;
    double sum, average;
    int[] grades = new int[100];

    // Input the data
    count = 0;
    System.out.println("Enter grade:");
    studentGrade = kb.nextInt();
    while (studentGrade >= 0) {
        grades[count] = studentGrade;
        count = count + 1;
        System.out.println("Enter grade:");
        studentGrade = kb.nextInt();
    }
    Enter grade: 95
    Enter grade: 70
    Enter grade: 82
    Enter grade: 64
    Enter grade: -1
    Average is: 77.75
    Grade 1 is >= average
    Grade 2 is < average
    Grade 3 is >= average
    Grade 4 is < average
// Compute the average
sum = 0;
for (int i=0; i<=count-1; i++) // Notice grades.length is NOT used here.
    sum = sum + grades[i];
average = (double)sum/count;

System.out.println("Average is:" + average);

// Compare each grade to the average
for (int i=0; i<=count-1; i++) // Notice that grades.length is NOT used here.
    if (grades[i] < average)
        System.out.println("Grade in position " + (i+1) + " is < than average");
    else
        System.out.println("Grade in position " + (i+1) + " is >= average");
}
2-Dimensional Arrays

- A 2-D array corresponds to a table or grid:
  - Commonly referred to as a *matrix* or a *table*
  - One dimension is referred to as the *row*
  - The other is referred to as the *column*

- Accessing a value in the matrix requires two values – a row value and a column value.
### Example – 2 Dimensional Array

<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><strong>$1311</strong></td>
<td><strong>$1335</strong></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>
</tbody>
</table>

- Assume a starting balance of $1000
- Row identifier (first dimension) - year
- Column identifier (second dimension) - percentage
Example – 2 Dimensional Array

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1050</td>
<td>1055</td>
<td>1060</td>
<td>1065</td>
<td>1070</td>
<td>1075</td>
</tr>
<tr>
<td>1</td>
<td>1103</td>
<td>1113</td>
<td>1124</td>
<td>1134</td>
<td>1145</td>
<td>1156</td>
</tr>
<tr>
<td>2</td>
<td>1158</td>
<td>1174</td>
<td>1191</td>
<td>1208</td>
<td>1225</td>
<td>1242</td>
</tr>
<tr>
<td>3</td>
<td>1216</td>
<td>1239</td>
<td>1262</td>
<td>1286</td>
<td>1311</td>
<td>1335</td>
</tr>
<tr>
<td>4</td>
<td>1276</td>
<td>1307</td>
<td>1338</td>
<td>1370</td>
<td>1403</td>
<td>1436</td>
</tr>
</tbody>
</table>

- Both rows and columns are numbered starting at 0
  - Balance[3][4] = 1311
  - Balance[3][4] corresponds to year 4 and 7% interest
2-D arrays are a straight-forward generalization of 1-D arrays.

A 2-D array of ints named `table` with 10 rows and 6 columns:

```java
int[][] table = new int[10][6];
```

The declaration and allocation can be separate.

```java
int[][] table;
table = new int[10][6];
```
Initializing a 2-D Array in Java

- An initial value can be specified in the declaration.

```java
int[][] table = {{10,5,3},
                 {7,21,13},
                 {85,90,2},
                 {12,9,4}};
```

- How many rows and columns does `table` have?
  - The number of rows and columns is implicit.

- Note the use of white-space in the declaration.
Initializing a 2-D Array in Java

- Doubly-nested loops are commonly used with 2-D arrays.

- Example – Initializing a 2-D array:

```java
int[][] table = new int[4][6];

// Initialize table to contain random numbers
for (int row=0; row<4; row++)
    for (int column=0; column<6; column++)
        table[row][column] = random.nextInt(10) + 1;
```
More examples:

```java
int[][] table = new int[4][6];

// Adding up all the values
int sum = 0;
for (int row=0; row<4; row++)
    for (int column=0; column<6; column++)
        sum = sum + table[row][column];

// Incrementing all the values by 1
for (int row=0; row<4; row++)
    for (int column=0; column<6; column++)
        table[row][column] = table[row][column] + 1;
```
And more examples:

// Outputing all the values
for (int row=0; row<4; row++)
    for (int column=0; column<6; column++)
        System.out.println(table[row][column]);

// Outputing all the values, one row per line
for (int row=0; row<4; row++) {
    for (int column=0; column<6; column++)
        System.out.print(table[row][column]);
    System.out.println();
}
All of the previous loops executed in what is called “row-major order.”

Any one of them could have been executed in “column-major order.”

```java
int[][] table = new int[4][6];

// Initialize table to contain random numbers
for (int column=0; column<6; column++)
    for (int row=0; row<4; row++)
        table[row][column] = random.nextInt(10) + 1;
```
// Adding up all the values
int sum = 0;
for (int column=0; column<6; column++)
    for (int row=0; row<4; row++)
        sum = sum + table[row][column];

// Incrementing all the values by 1
for (int column=0; column<6; column++)
    for (int row=0; row<4; row++)
        table[row][column] = table[row][column] + 1;

// Outputing all the values
for (int column=0; column<6; column++)
    for (int row=0; row<4; row++)
        System.out.println(table[row][column]);

// Outputing all the values, one column per line
for (int column=0; column<6; column++)
    for (int row=0; row<4; row++) {
        System.out.print(table[row][column]);
        System.out.println();
    }
In Java, a 2-dimensional array is implemented as an “array of arrays.”

```java
int[][] table = new int[3][4];
```

In the above declaration `table` is (technically) a 1-D array of length 3.

Each element `table[i]` is an array of length 4.
Consequently, given a 2D array:

```java
int[][] table = new int[3][4];
```

The number of rows (3) is given by:

```java
table.length
```

The number of columns (4) in row \(i\) is given by:

```java
table[i].length
```
These attributes can then be used when processing the array.

```java
// Outputing all the values
for (int row = 0; row < table.length; row++)
    for (int column = 0; column < table[row].length; column++)
        System.out.println(table[row][column]);
```
Row and Column Major Order

- Note that any of the preceding loops could have been written in either row or column major order.

```java
// Outputing all the values in row major format
for (int row=0; row<table.length; row++) {
    for (int column=0; column<table[row].length; column++)
        System.out.print(table[row][column]);
    System.out.println();
}

// Outputing all the values in column major format
for (int column=0; column<table[0].length; column++) {
    for (int row=0; row<table.length; row++)
        System.out.print(table[row][column]);
    System.out.println();
}
```
Row and Column Major Order

- Depending on the language, an array is stored in either row or column major order.

```java
char[][] myChars = new char[3][4];
```

```
<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>S</td>
<td>R</td>
<td>D</td>
</tr>
<tr>
<td>Q</td>
<td>P</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>
```

- In row major order, array elements are stored in memory as:

```
B C A X W S R D Q P D F
```

- In column major order, array elements are stored in memory as:

```
B W Q C S P A R D X D F
```

- This has a number of implications for efficiency (but for now, don’t worry about it).
More generally, a multi-dimensional array is an array with more than one index.

Number of dimensions is unbounded – 3D, 4D, etc., are all possible.

Processing multi-dimensional arrays requires correspondingly nested loops.
Consider the following array declaration:

```java
int[][] a;
```

// Suppose a is allocated and loaded with integers.

Give a segment of Java code that determines if the array is square and outputs an appropriate message.

Write the above code by hand first, then type it in, compile it, and run it.
Array Exercises

- Assume similar declarations for each of the following

- Easy:
  - Given one 2-D array of strings, plus a string str, a (valid) row number $i$, and a (valid) column number $j$, set the value in row $i$ and column $j$ of the array to the string str.
  - Given one 2-D array of strings, plus a string str, and a (valid) row number $i$, set all of the values in row $i$ of the array to the string str.
  - Same as the last exercise, but where the integer $i$ is a (valid) column number.

- Challenging:
  - Given two 2-D arrays of integers, determines if the arrays are identical (i.e., the same number of rows and columns, plus the same contents).
  - Given two 2-D arrays of integers that have the same number of rows and columns, constructs a third array that contains the maximum of the corresponding elements of the two given arrays.
  - Given one 2-D array of integers plus two (valid) row numbers $i$ and $j$, swap the contents of rows $i$ and $j$.
  - Same as the previous exercise, but where $i$ and $j$ are valid column numbers.
Difficult:

- Given one 2-D integer array, performs a vertical mirror-transformation of that array.
- Given one 2-D integer array, performs a horizontal mirror-transformation of that array.
- Given one square 2-D integer array, performs a top-left, bottom-right, diagonal-transformation of that array.
- Given one square 2-D integer array, performs a bottom-left, top-right, diagonal-transformation of that array.
- Given two 2-D integer arrays, determines if the first array is a sub-array of the second.
- Given one 2-D integer array, performs a circular, horizontal shift of the array, i.e., each row shifts one-position to the right, but where the element at the end of row $i$ shifts to the first location of row $i+1$; also, the last element of the last row shifts to the first position of the first row.
- Same as the last exercise, but do a circular, vertical shift of the array.