Arrays

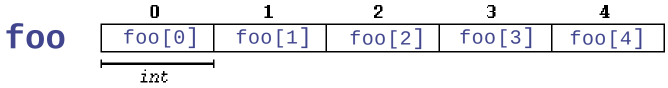
List (or Sequence)

- A **List** is an **ordered collection** of items of some element type E.
- Note that this doesn't mean that the objects are in sorted order, it just means that each object has a position in the List, (starting with position zero).

Array

- An array is a ***finite ordered list*** of ***homogeneous*** elements which occupy contiguous memory and each element is referenced by an ***array name*** and ***index***.
- In Java
  - An ***array*** is a group of homogeneous data elements that share the same name and are ordered sequentially from zero to one less than the number of data elements in the array.

```
int[] foo = new int[5];
```



  - The number of data elements that can be stored in the array is called the array's ***length***.

Arrays (cont'd)

- In Java, arrays are ***first-class objects***.
  - For example, you can assign one array of integers to another, just as you can assign one integer variable to another.
- Once you create an array, you cannot change its size, though you can modify individual components of the array.
- If you want to dynamically change the size of an array during program execution, use **`java.util.{ArrayList, Vector}`** object instead.

Three Steps to Creating Arrays

Step 1: **Declaring Arrays**

Step 2: **Creating Arrays**

Step 3: **Initializing Arrays**

Declaring Arrays

- Like all other variables in Java, an array must be declared. Declaring arrays merely says what type of values the array will hold.

```
int[]    arrayOfInt;    // array of ints
String[] arrayOfString; // array of Strings
```

Note: Be ware that, unlike C, *no dimension (or length) of an array is specified.* For example, "`int[10] arrayOfInt`" is illegal in Java.

Declaring Arrays (cont'd)

- Alternative forms:

```
int     arrayOfInt[];    // don't specify dim.
String  arrayOfString[]; // here, either

//int[]     arrayOfInt[];      // array of ints
//String[]  arrayOfString[]; // array of Strings
```

Note: "`int[] a, b;`" is the same as "`int[] a; int[] b;`".

- *Q*: What is the type of **b** in the following case?

```
int   a[], b;     // is b an integer or
                  // an array of integers?
```

Creating Arrays

- In Java, arrays are created (i.e. memory is allocated) using **new** operator.

```
arrayOfInt     = new int[100];
arrayOfFloat   = new float[200];
arrayOfDouble = new double[300];
```

Note: Every object in Java is created using **new** operator and arrays are also objects in Java.

- The numbers in the brackets(**[]**) specify the length of the array. Therefore **`arrayOfInt = new int[100]`** creates an array of 100 integers.

Creating Arrays (Cont'd)

- Array components are indexed from **`0`** to **`length-1`** as in C. That is, **`arrayOfInt[0]`** is the first component, **`arrayOfInt[1]`** is the second component, and **`arrayOfInt[99]`** is the last component of the array.
- *Q*: What will happen if you try to access **`arrayOfInt[100]`**?
  - **`ArrayIndexOutOfBoundsException`** is raised. Java performs a runtime range checking for array component access.

Initializing Arrays

- Once array created, you need to ***initialize*** the components of the array.

```
double[] squares;           // declaration
squares = new double[100]; // creation

for (int i = 0; i < squares.length; i++) {
      squares[i] = i*i; // type promotion
}
```

Note that the length of an array can be obtained by referring to the ***length*** field of the array object like **`squares.length`**.

Initializing Arrays (Cont’d)

- You can declare and create an array at the same time:

```
double[] squares = new double[100];
String[] name = new String[10];
```

- You can even declare, create, and initialize an array at the same time:

```
int[] intArray = {1, 2, 3, 4, 5};
String[] name = {“Stacy”, “Tracy”, “Dorothy”};
```

Notice that you do not use a call to **new** when using this syntax.

```
int[] intArray = new int[] {1, 2, 3, 4, 5}; // works,too!
```

Copying Arrays

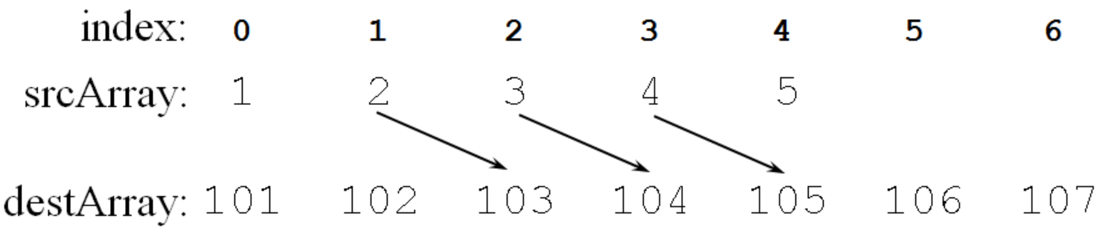
- Java has an extremely useful method in its **System** class for copying all or part of an array to another array.

```
System.arraycopy(srcArray, srcIndex, destArray,
                 destIndex, numberOfEntriesToCopy);

int[] srcArray  = {1,2,3,4,5};
int[] destArray = {101,102,103,104,105,106,107};
```

Copying Arrays (Cont'd)

```
System.arraycopy(srcArray,1,destArray,2,3);
for(int i=0; i < destArray.length; i++) {
    System.out.println(destArray[i]);
}
```



index:	0	1	2	3	4	5	6
srcArray:	1	2	3	4	5		
destArray:	101	102	103	104	105	106	107

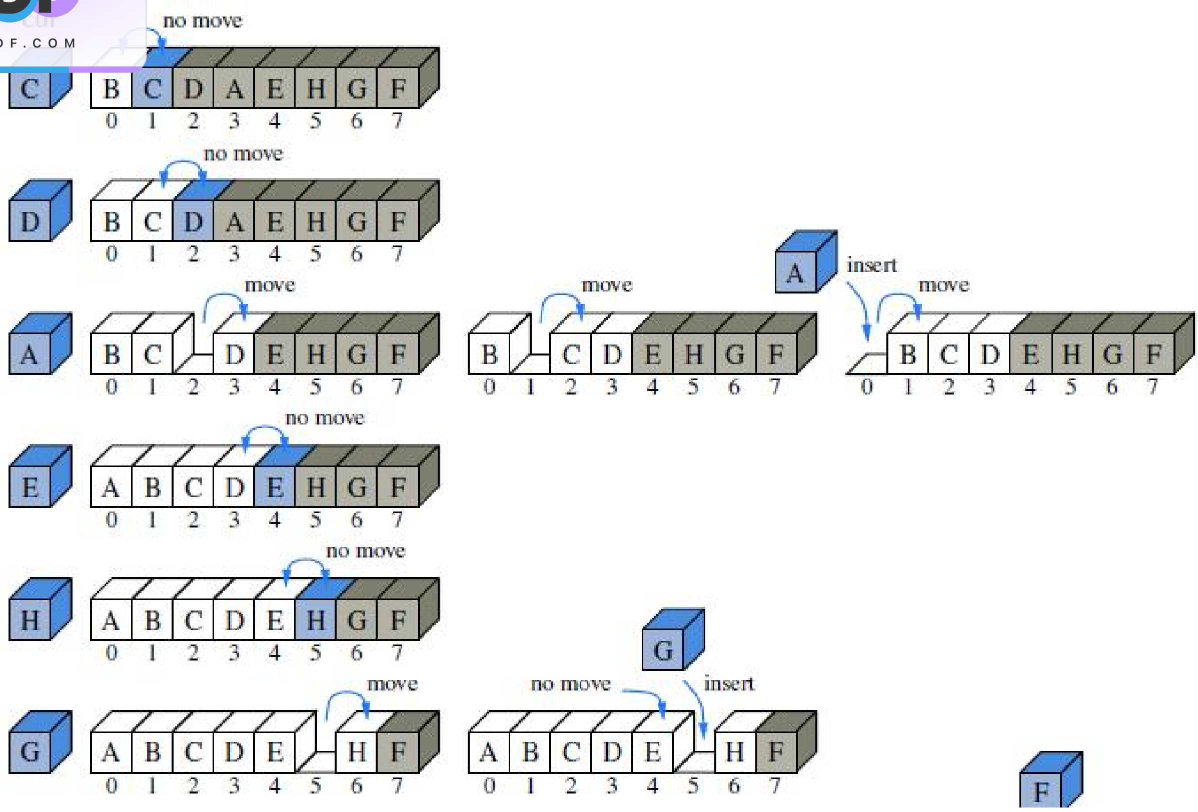
```
Algorithm InsertionSort(A):
    Input: An array A of n comparable elements
    Output: The array A with elements rearranged in nondecreasing order
    for k from 1 to n − 1 do
        Insert A[k] at its proper location within A[0], A[1], . . ., A[k].
```

A[0],A[1],…,A[k-1]

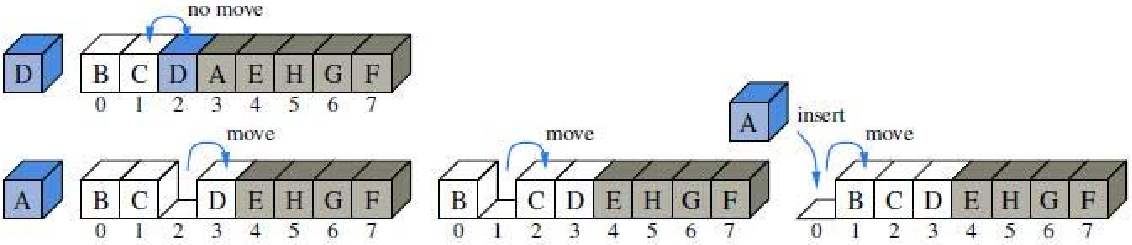
A[k],A[k+1],…A[n-1]

Sorted

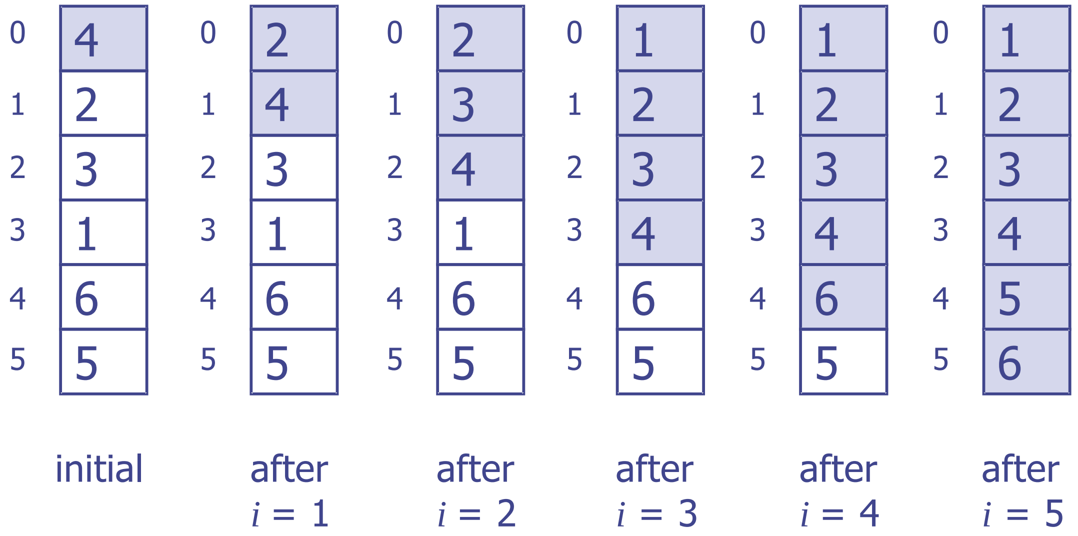
Unsorted



```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {              // begin with second character
    char cur = data[k];                      // time to insert cur=data[k]
    int j = k;                               // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {       // thus, data[j-1] must go after cur
      data[j] = data[j−1];                   // slide data[j-1] rightward
      j−−;                                   // and consider previous j for cur
    }
    data[j] = cur;                           // this is the proper place for cur
  }
}
```



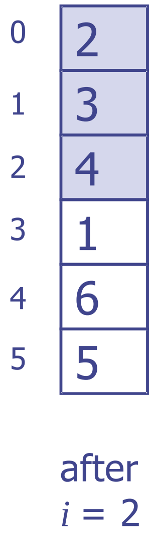
Insertion Sort -- Example



	initial	after i = 1	after i = 2	after i = 3	after i = 4	after i = 5
0	4	2	2	1	1	1
1	2	4	3	2	2	2
2	3	3	4	3	3	3
3	1	1	1	4	4	4
4	6	6	6	6	6	5
5	5	5	5	5	5	6

Insertion Sort (Another Solution)

```
for (int k = 1; k < n; k++)
{
  j = k;
  while (j > 0 && A[j-1] > A[j])
  {
     swap A[j-1] and A[j-1];
     j = j-1;
  }
}
```



	after i = 2
0	2
1	3
2	4
3	1
4	6
5	5

Java.util.Arrays

- **equals(A, B)**
  - Returns true iff the array **A** and the array **B** are equal.
- **fill(A, x)**
  - Store element **x** into every cell of **A**.
- **copyOf(A, n)**
  - Returns an array of size ***n*** such that the first ***n*** elements are copied from **A**. If ***n*** > ***A.length***, then remaining elements are padded with default value.
- **copyOfRange(A,s,t)**
  - Returns a subarray of **A** with length t-s from **A[s]** to **A[t-1]**.
- **sort(A)**
  - Sorts the array **A** based on natural ordering of elements. (Quick sort)
- **toString(A)**
  - Return a String representation of **A**.

Two Dimensional Arrays

- In Java, two dimensional arrays are implemented as ***arrays of arrays***. The following statement declares and creates an array of arrays of **doubles**. The first dimension is 2 and the second, 3 in this case:

```
double[][] M = new double[2][3];
```

- You can also use a shortcut to declare, create, and initialize two dimensional arrays at the same time.

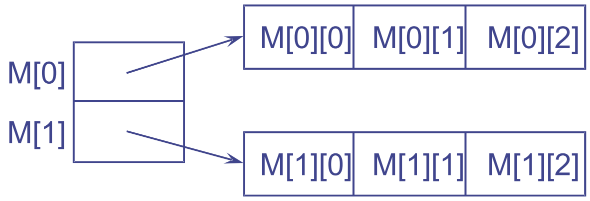
```
double[][] M = {{0,1,2}, {1,2,3}};
```

Two Dimensional Arrays (Cont'd)

- You can initialize this array like this:

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

M[0][0]	M[0][1]	M[0][2]
M[1][0]	M[1][1]	M[1][2]



Two Dimensional Arrays (Cont'd)

- When you create a two dimension array without initialization, the ***first dimension must be specified***, but the second dimension may be left unspecified, to be filled in later.

mandatory

```
double[][] M = new double[2][];
```

optional

```
for (int i = 0; i < M.length; i++) {
    M[i]= new double[3];
}

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

Sparse Matrix

- 2-dimensional matrix

$$\begin{array}{c|cccccc|}
 & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} \\
\text{row0} & \underline{15} & 0 & 0 & \underline{22} & 0 & \underline{-15} \\
\text{row1} & 0 & \underline{11} & \underline{3} & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & \underline{-6} & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & \underline{91} & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & \underline{28} & 0 & 0 & 0
\end{array}$$

Only 8 out of 36 elements (6 * 6) are nonzero → sparse matrix

Representation of Sparse Matrix

- Uniquely characterize any element within a matrix by using the triple ***(row, col, value)***.
- Then, use an array of triples to represent a sparse matrix.
- Store triples so that row indices are in an ascending order.
- All column indices for any row are stored in an ascending order.

Eg.) $$\begin{bmatrix} 0 & 0 & 0 & 0.3 \\ 0 & 1.1 & 0 & 0 \\ 0 & 0 & 2.2 & 2.3 \end{bmatrix}$$

⟶ ((0, 3, 0.3), (1, 1, 1.1), (2, 2, 2.2), (2, 3, 2.3))

Sparse Matrix Example

$$\begin{array}{c|cccccc} & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} \\ \hline \text{row0} & \underline{15} & 0 & 0 & \underline{22} & 0 & \underline{-15} \\ \text{row1} & 0 & \underline{11} & 3 & 0 & 0 & 0 \\ \text{row2} & 0 & 0 & 0 & \underline{-6} & 0 & 0 \\ \text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{row4} & \underline{91} & 0 & 0 & 0 & 0 & 0 \\ \text{row5} & 0 & 0 & \underline{28} & 0 & 0 & 0 \end{array}$$

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

a[0].row = # of rows,
a[0].col = # of columns,
a[0].value = # of nonzero entries

major order minor order

Sparse Matrix ADT

for all a, b ∊ Sparse_Matrix, x ∊ item, i,j,max_col, max_row ∊ index

Sparse_Matrix Create (max_row, max_col) **::=**
return a *Sparse_matrix* that can hold up to *max_items* = *max_row* * *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

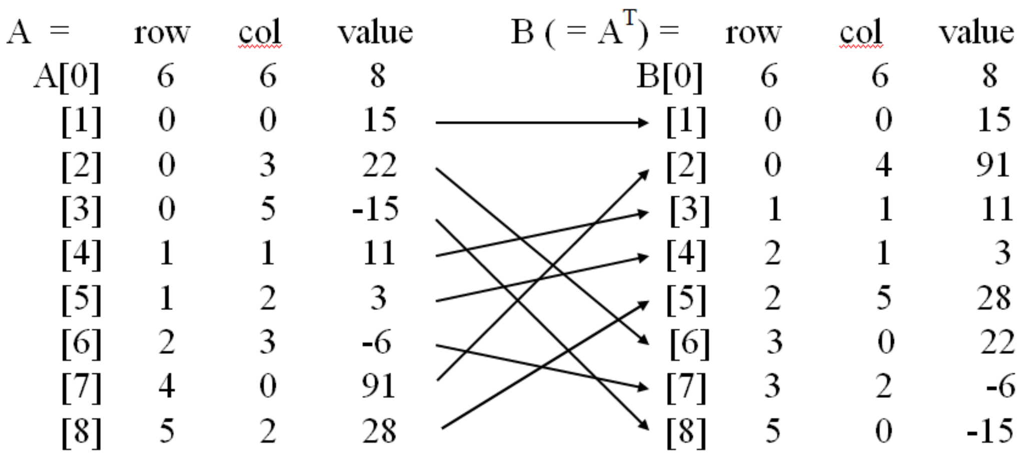
Sparse_Matrix Transpose(a) **::=**
return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(a, b) **::=** **if** the dimensions of *a* and *b* are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else return error

Sparse_Matrix Multiply(a, b) **::=** **if** number of columns in *a* equals number of rows in *b* **return** the matrix d produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum (a[i][k] \bullet b[k][j])$ where *d(i,j)* is the *(i,j)*th element **else return** error

Transposing a Sparse Matrix

```
for all elements in column j
  place element (i, j, value) in
  element <j, i, value>
```



A =	row	col	value
A[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

B (= A^T) =	row	col	value
B[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15