CMPS 4143 - Topics in Contemporary Programming Languages

L2-3- Java Array and Algorithms

Images and text taken from textbook unless otherwise noted.

Array as ADT

An array is an Abstract Data Type

- The array type has a set of *values*
- The array type has a set of *operations* that can be applied uniformly to each of these values
  - The only operation is *indexing*
  - Alternatively, this can be viewed as two operations:
    - *inspecting* an indexed location
    - *storing into* an indexed location
- It's *abstract* because the implementation is hidden: all access is via the defined operations

Array Data Structure

An array is an indexed sequence of components

- Typically, the array occupies sequential storage locations
- The length of the array is determined when the array is created, and cannot be changed
- Each component of the array has a fixed, unique index
  - Indices range from a lower bound to an upper bound
  - Any component of the array can be inspected or updated by using its index
    - This is an efficient operation: O(1) = constant time

Array Variation I

- The array indices may be integers (C++, Java) or other discrete data types (Pascal, Ada)
- The lower bound may be zero (C++, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)
- Bounds may be dynamic , that is set at run-time (Ada, C++, Java) or not (Pascal)
- Bounds may be undefined (Ada, Java)

Array Variation II

- Multidimensional (C++),
- Faked multidimensional (Java, Ada, Pascal)
- Arrays may be rectangular (C++, Ada) or nonrectangular (Java)
- Arrays may be accessed in “slices” (Java, Ada, Pascal) or not (C++)
- Array initializations are possible (Java, C++, Pascal, Ada)

Array Variation III

- In most languages, arrays are homogeneous (all components must have the same type);
- In some (Lisp, Prolog) the components may be heterogeneous (of varying types)

- In an object-oriented language, arrays may be objects (Java) or not objects (C++)

Array Variation IV

- Whole operations
  - Fortran: NO

  -Ada and Pascal: YES (assignment of entire arrays of same size and *type, not structure)*

  -C++: NO (arrays referenced by pointers)

  -Java: Yes (clone and equality test)

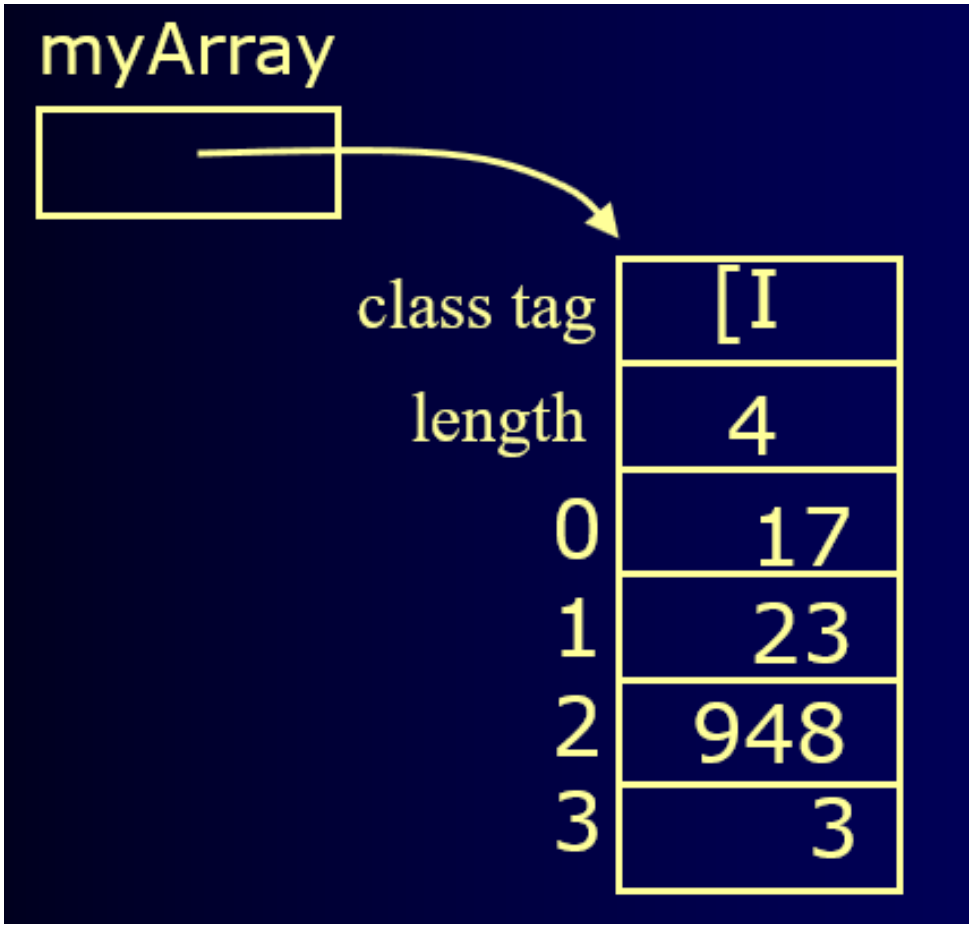
- Arrays may carry additional information about themselves, such as type and length (Java), or may consist *only* of their components (C, C++)
  - The terms reflective and non-reflective, respectively, refer to these two types of arrays
  - This is not standard terminology, but it is consistent with other uses of the terms

Array in Java I

- Array indices are integers
  - The bracket notation `a[i]` is used (and not overloaded)
  - Bracket operator performs bounds checking
- An array of length `n` has bounds `0` and `n-1`
- Arrays are homogeneous
  - However, an array of an object type may contain objects of any subtype of that object
    - For example, an array of `Animal` may contain objects of type `Cat` and objects of type `Dog`
    - An array of `Object` may contain any type of object (but cannot contain primitives)

Array Visualization

Here's one way to visualize an array in Java:



Arrays are objects

Arrays are allocated by new, manipulated by reference, and garbage collected

```
int a[]= new int [100];
int[] a = new int [100];
```

Arrays can be initialized

```
int [ ] smallPrimes = {2,3,5,7,11,13};
```

Array declaration and initialization

dataType[] arrayName;

```
// declare an array
double[] data;

// allocate memory
data = new double[10];

double[] data = new double[10];

//declare and initialize and array
int[] age = {12, 4, 5, 2, 5};
```

```
// initialize array
age[0] = 12;
age[1] = 4;
age[2] = 5;
```

Printing an Array

```
class Main {
 public static void main(String[] args) {

    // create an array
    int[] age = {12, 4, 5};

    // loop through the array
    // using for loop
    System.out.println("Using for Loop:");
    for(int i = 0; i < age.length; i++) {
      System.out.println(age[i]);
    }
 }
}
```

Printing an Array with for-each loop

```
class Main {
 public static void main(String[] args) {

    // create an array
    int[] age = {12, 4, 5};

    // loop through the array
    // using for loop
    System.out.println("Using for-each Loop:");
    for(int a : age) {
      System.out.println(a);
    }
 }
}
```

Compute Sum and AVG of an Array

```
class Main {
 public static void main(String[] args) {

   int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
   int sum = 0;
   Double average;

   // access all elements using for each loop
   // add each element in sum
   for (int number: numbers) {
     sum += number;
   }

   // get the total number of elements
   int arrayLength = numbers.length;

   // calculate the average
   // convert the average from int to double
   average =  ((double)sum / (double)arrayLength);

   System.out.println("Sum = " + sum);
   System.out.println("Average = " + average);
 }
}
```

Accessing Array Elements

```
int[] array = new int[] { 3, 5, 2, 5, 14, 4 };

// access array elements
array[index]

int firstItem = array[0];

int lastItem = array[array.length - 1];

int anyValue = array[new
Random().nextInt(array.length)];
```

Append to an Array

```
int[] newArray = Arrays.copyOf(array, array.length + 1);

newArray[newArray.length - 1] = newItem;


int[] newArray = ArrayUtils.add(array, newItem);
```

Insert a value between two values

```
int[] largerArray = ArrayUtils.insert(2, array, 77);
```

Array Comparison

```
boolean areEqual = Arrays.equals(array1, array2);
```

Checking an Empty Array

```
// These are empty arrays
Integer[] array1 = {};
Integer[] array2 = null;
Integer[] array3 = new Integer[0];

// All these will NOT be considered empty
Integer[] array3 = { null, null, null };
Integer[][] array4 = { {}, {}, {} };
Integer[] array5 = new Integer[3];
```

```
boolean isEmpty = array == null || array.length == 0;

boolean isEmpty = ArrayUtils.isEmpty(array);
```

Shuffling an Array

```
ArrayUtils.shuffle(array);
```

Boxing and Unboxing an Array

```
Integer[] list = ArrayUtils.toObject(array);
```

```
Integer[] objectArray = { 3, 5, 2, 5, 14, 4 };
int[] array = ArrayUtils.toPrimitive(objectArray);
```

Remove duplicates from an Array

```
// Box
Integer[] list = ArrayUtils.toObject(array);

// Remove duplicates
Set<Integer> set = new
HashSet<Integer>(Arrays.asList(list));

// Create array and unbox
return ArrayUtils.toPrimitive(set.toArray(new Integer[set.size()]));
```

Arrays are reflective

a.length is the length of array a

```
for (int i= 0; i < a .length; i++)
        System.out.println (a[i]);
```

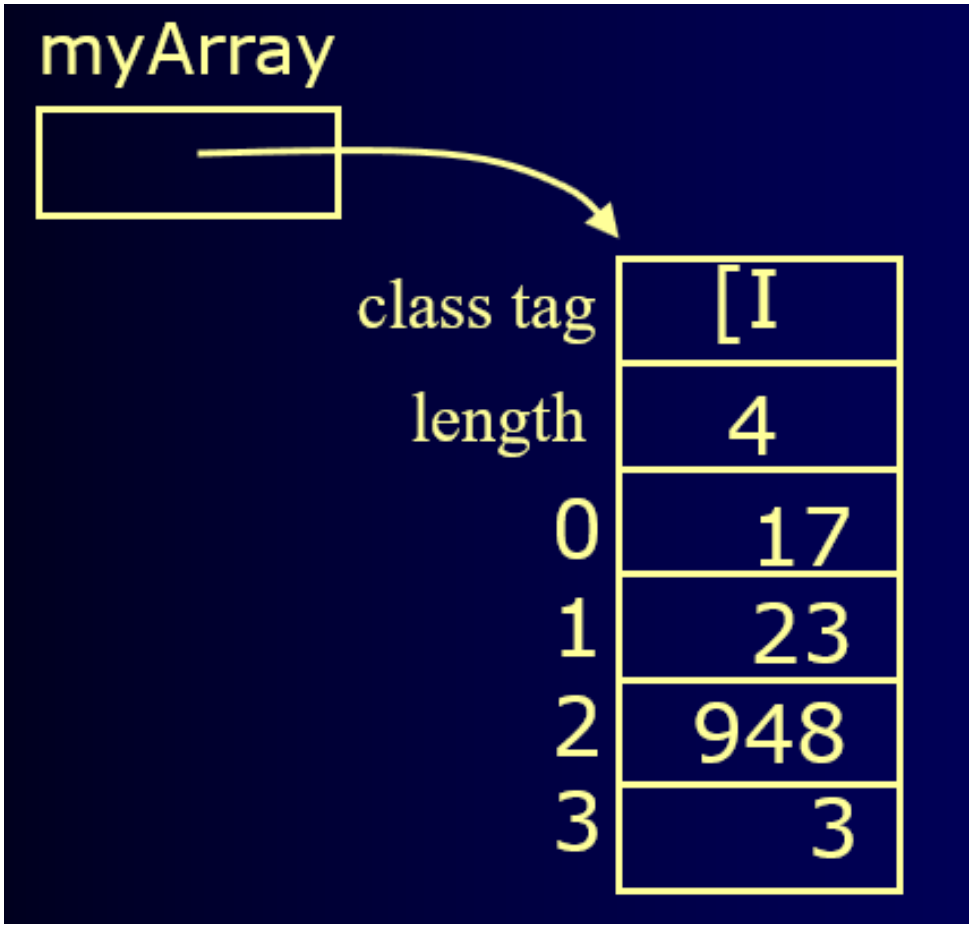
a.getClass() is the type of array a

An array of integers has type [I

An array of Strings has type [Ljava.lang.String;

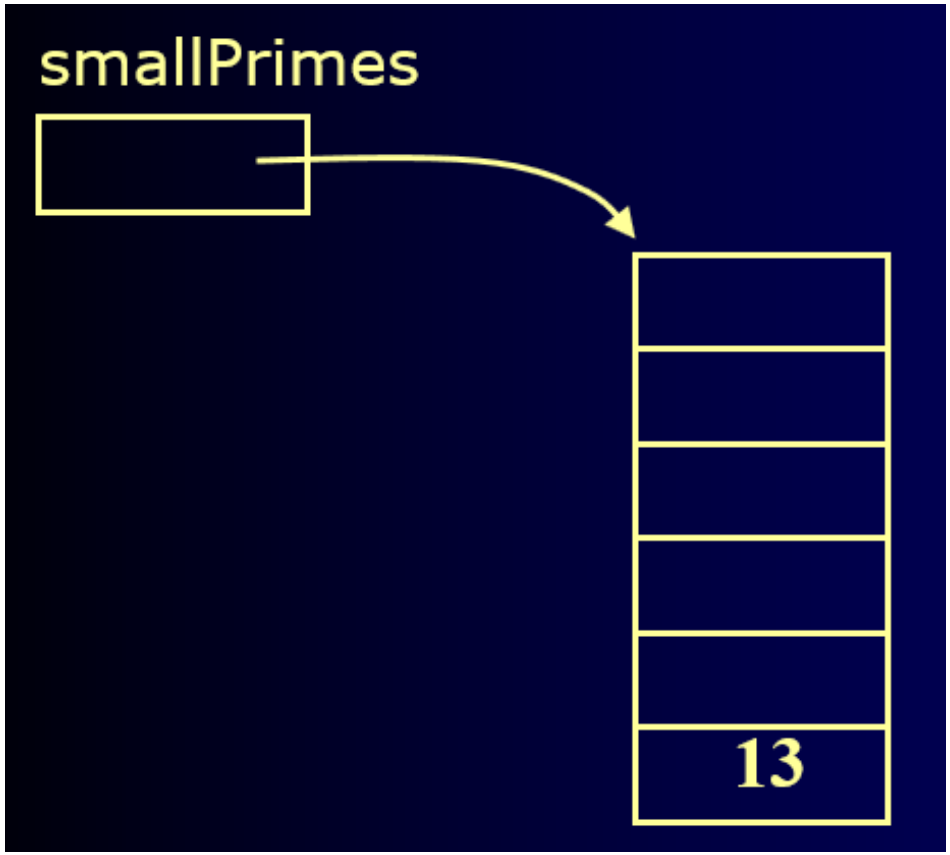
Array Visualization

Here's one way to visualize an array in Java:



Coping Array

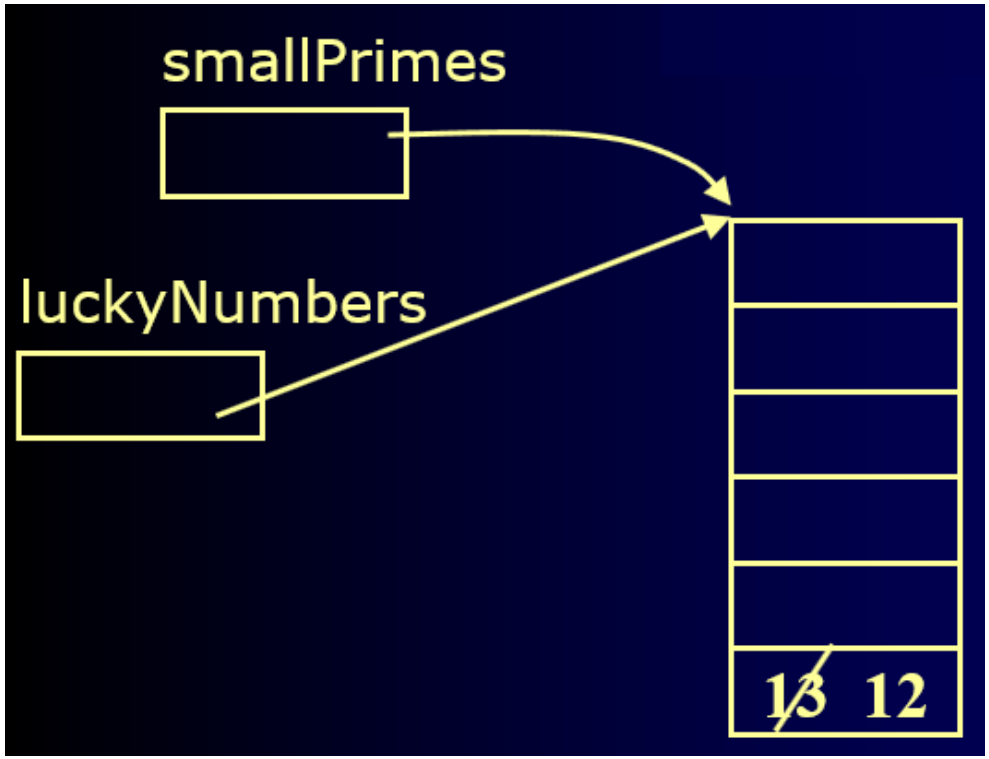
```
int[] luckyNumbers =
smallPrimes;

luckyNumbers[5]=12;
```



Copying an Array (..)

Copying arrays - The wrong way

```
int[] luckyNumbers =
smallPrimes;

luckyNumbers[5]=12;
```



Copying arrays - The right way

```
System.arraycopy (fromObject, fromIndex , toObject,  toIndex, count);

toObject = Arrays.copyOf(fromObject, fromObject.length);  (SE 6)
```

Example:

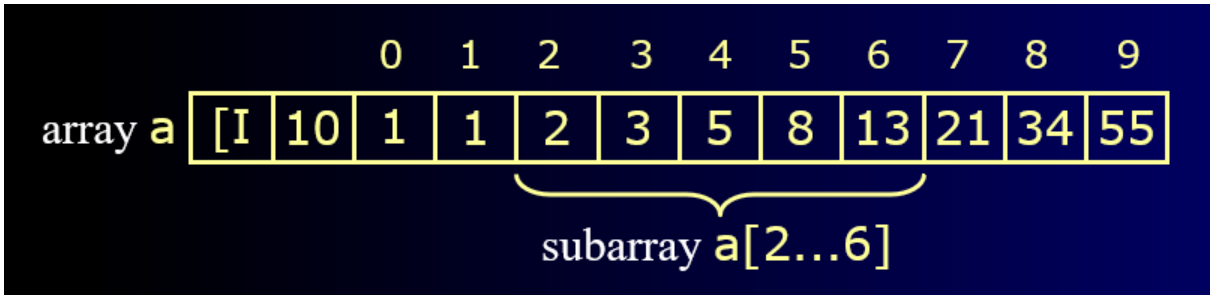
```
int [ ] luckyNumbers;

System.arraycopy (smallPrimes, 0, luckyNumbers, 0,
                                    smallPrimes.length);

int [ ] copiedLuckyNumbers = Arrays.copyOf (luckyNumbers,
                                    luckyNumbers.length);
```

Array Sort

Sorting arrays

```
Arrays.sort (smallPrimes);
```

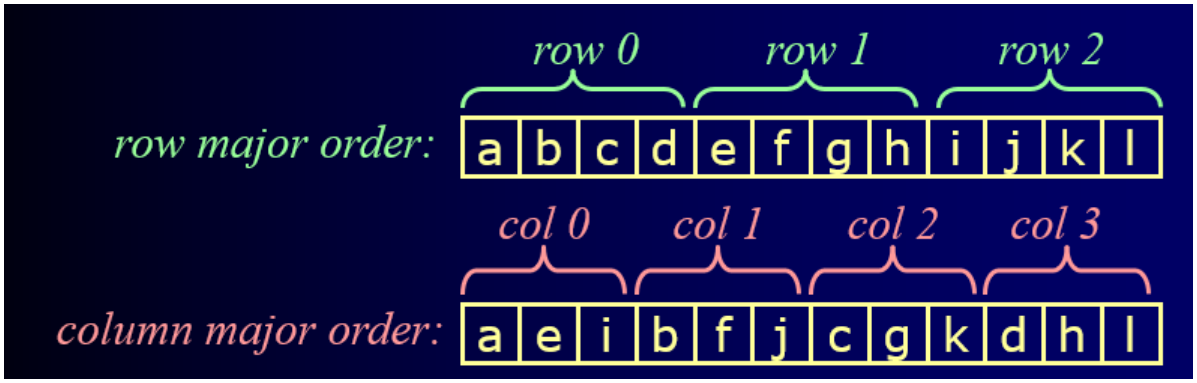
A subarray is a consecutive portion of an array

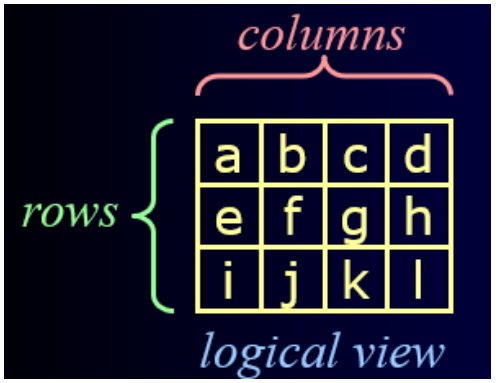


- Java provides *no language support* for subarrays
- To use a subarray, *you* must keep track of (1) the array itself, (2) the lower bound, and (3) the upper bound
- Typically, these will be three parameters to a method that does something with the subarray

Two-Dimensional Array I

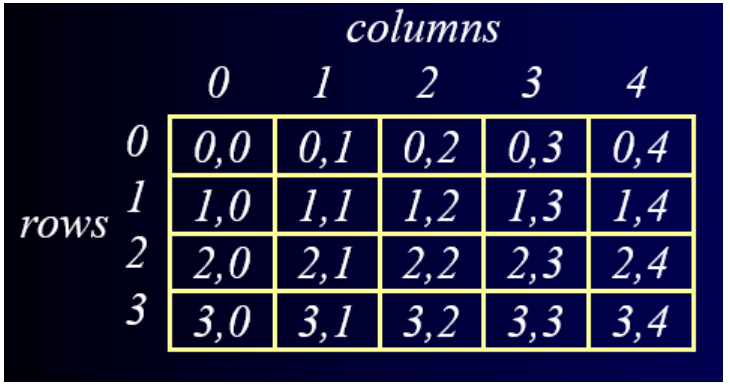
- Some languages (Fortran, Pascal) support two-dimensional (2D) arrays:
- A two-dimensional array may be stored in computer memory in either of two ways:





Two-Dimensional Array II

- In a 2D array, we generally consider the first index to be the row, and the second to be the column: a[*row, col*]
- In most languages we don't need to know the implementation;



- We work with this *abstraction*

Two-Dimensional Array in Java

- Java doesn't have "real" 2D arrays, but array elements can themselves be arrays:

  `int x[][];` denotes an array x of *array* components, each of which is an array of *integer* components

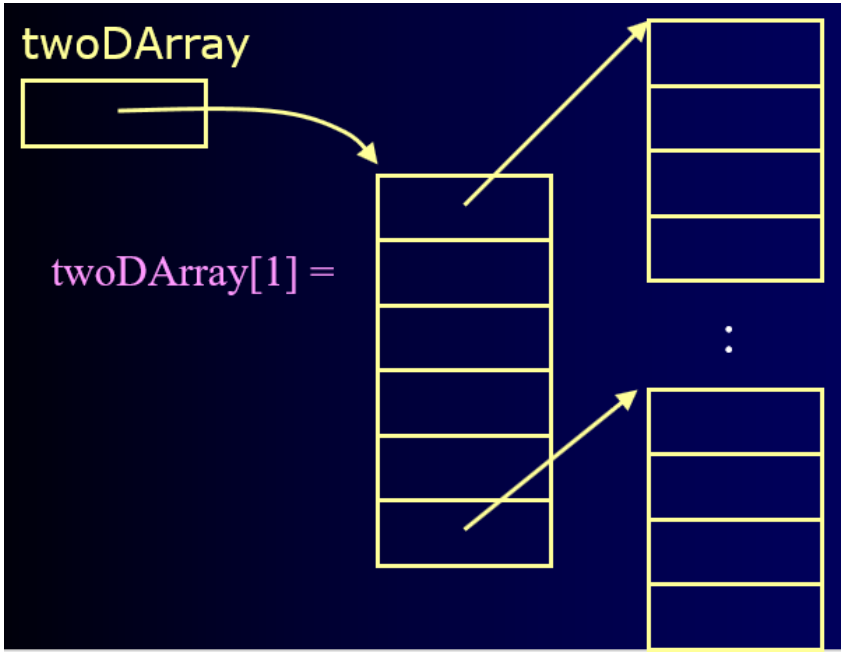
- We can *define* the above array like this:

  ```
  x = new int[5][8];
  ```

  and treat it as a regular 2D array

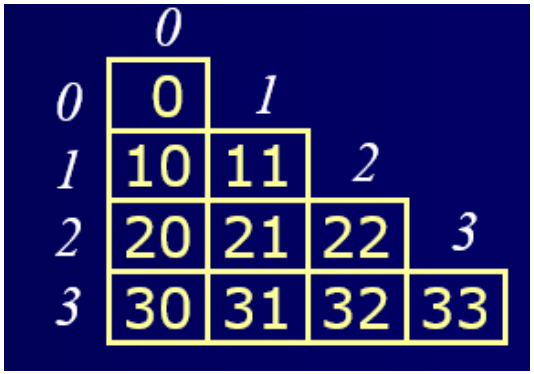
- However, we can do fancier things than this with arrays in Java

Java implements 2D arrays as array of arrays



Ragged (Nonrectangular) arrays

```
int ragged[][] = new int[4][];

for (int i = 0; i < 4; i++) {
    ragged[i] = new int[i + 1];
}
```



-
```
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < ragged[i].length; j++) {
        ragged[i][j] = 10 * i + j;
    }
}
```

Inserting an element into an array

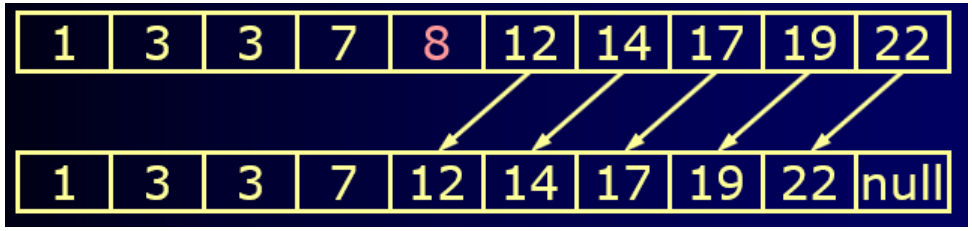
- Suppose we want to insert the value 8 into this sorted array (while keeping the array sorted)

- We can do this by shifting all the elements after the mark right by one location
  - Of course, we have to discard the **30** when we do this

- Moving all those elements makes this a *very slow* operation!

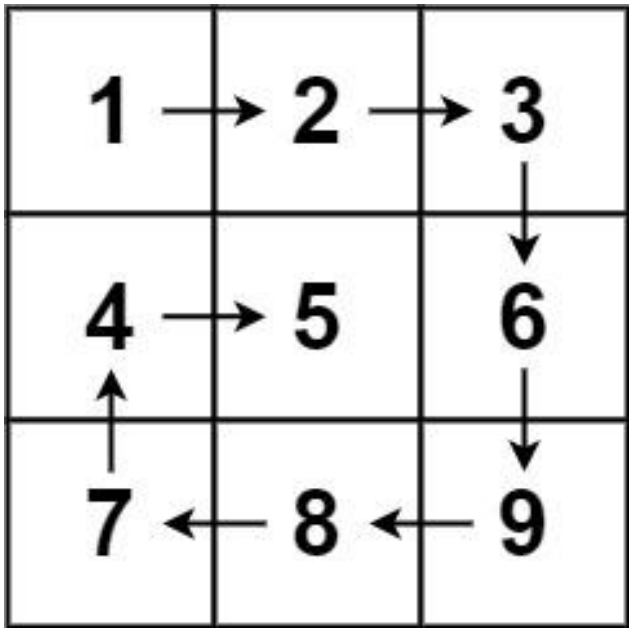
Deleting an element from an array

- Deleting an element is similar--again, we have to move all the elements after it
- This leaves a “vacant” location at the end--we can fill it with null



- Again, this is a slow operation; we don’t want to do it very often

Array Problems (Spiral Matrix)

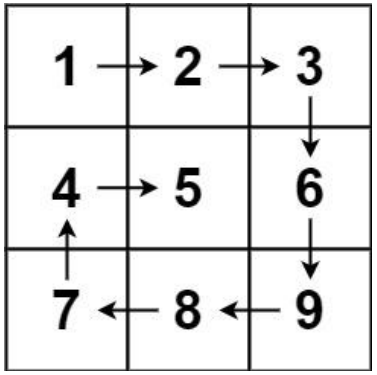


Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]

From position (row, col)

- move right: (row, col + 1)
- move downwards: (row + 1, col)
- move left: (row, col - 1)
- move upwards: (row - 1, col)

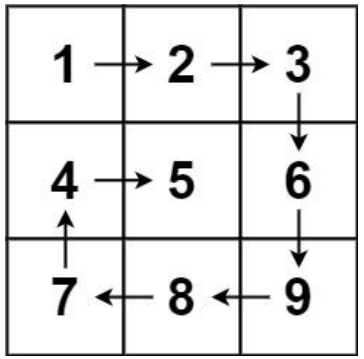
Array Problems (Spiral Matrix-Algorithm)



1. Initialize the top, right, bottom, and left boundaries as up, right, down, and left.
2. Initialize the output array result.
3. Traverse the elements in spiral order and add each element to result:
    1. Traverse from left boundary to right boundary.
    2. Traverse from up boundary to down boundary.
    3. Before we traverse from right to left, we need to make sure that we are not on a row that has already been traversed. If we are not, then we can traverse from right to left.
    4. Similarly, before we traverse from top to bottom, we need to make sure that we are not on a column that has already been traversed. Then we can traverse from down to up.
    5. Remember to move the boundaries by updating left, right, up, and down accordingly.
4. Return result.

Array Problems (Spiral Matrix-Algorithm)

1. Initialize the top, right, bottom, and left boundaries as up, right, down, and left.




```
int rows = matrix.length;
int columns = matrix[0].length;
int up = 0;
int left = 0;
int right = columns - 1;
int down = rows - 1;
```

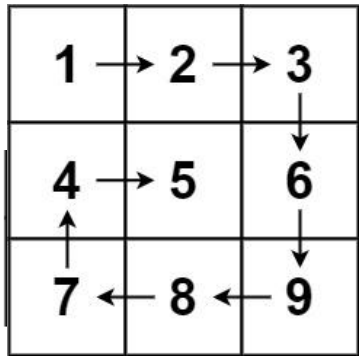
Array Problems (Spiral Matrix-Algorithm)

2. Initialize the output array result.

```
List<Integer> result = new ArrayList<>();
```

Array Problems (Spiral Matrix-Algorithm)

3. Traverse the elements in spiral order and add each element to result:



Traverse from left boundary to right boundary.

```
// Traverse from left to right.
for (int col = left; col <= right; col++) {
        result.add(matrix[up][col]);
}
```

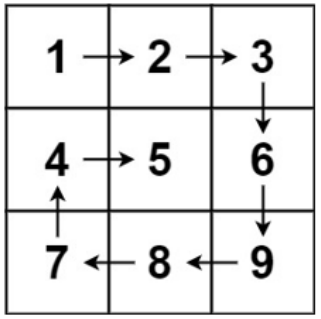
Traverse from up boundary to down boundary.

```
// Traverse downwards.
for (int row = up + 1; row <= down; row++) {
        result.add(matrix[row][right]);
}
```

Array Problems (Spiral Matrix-Algorithm)

3. Traverse the elements in spiral order and add each element to result:

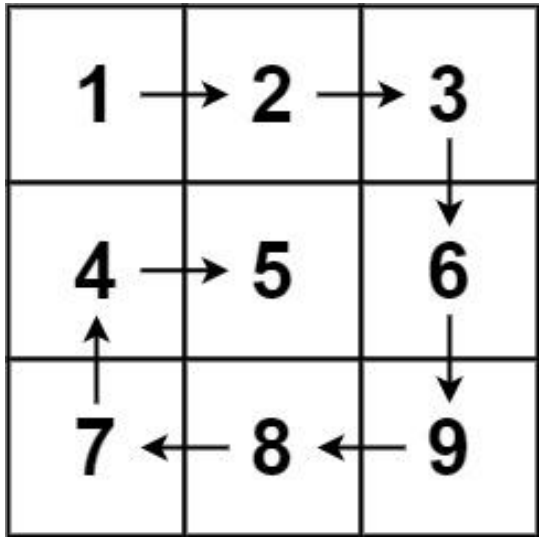
```
// Make sure we are now on a different row.
if (up != down) {
        // Traverse from right to left.
    for (int col = right - 1; col >= left; col--) {
        result.add(matrix[down][col]);
    }
}

// Make sure we are now on a different column.
if (left != right) {
        // Traverse upwards.
        for (int row = down - 1; row > up; row--) {
                result.add(matrix[row][left]);
        }
}
```



Array Problems (Spiral Matrix-Algorithm)

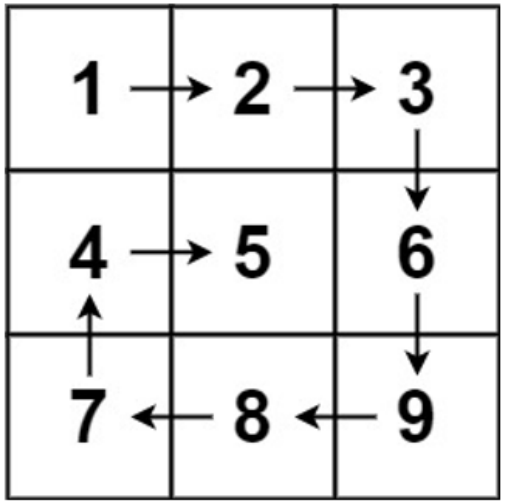
Remember to move the boundaries by updating left, right, up, and down accordingly.

```
left++;
right--;
up++;
down--;
```



Array Problems (Spiral Matrix)

```
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        int rows = matrix.length;
        int columns = matrix[0].length;
        int up = 0;
        int left = 0;
        int right = columns - 1;
        int down = rows - 1;

        while (result.size() < rows * columns) {
            // Traverse from left to right.
            for (int col = left; col <= right; col++) {
                result.add(matrix[up][col]);
            }
            // Traverse downwards.
            for (int row = up + 1; row <= down; row++) {
                result.add(matrix[row][right]);
            }
            // Make sure we are now on a different row.
            if (up != down) {
                // Traverse from right to left.
                for (int col = right - 1; col >= left; col--) {
                    result.add(matrix[down][col]);
                }
            }
            // Make sure we are now on a different column.
            if (left != right) {
                // Traverse upwards.
                for (int row = down - 1; row > up; row--) {
                    result.add(matrix[row][left]);
                }
            }
            left++; right--; up++; down--;
        }
        return result;
    }
}
```



Conclusion

- Arrays have the following advantages:
  - Accessing an element by its index is very fast (constant time)

- Arrays have the following disadvantages
  - All elements must be of the same type
  - The array size is fixed and can never be changed
  - Insertion into arrays and deletion from arrays is very slow