

# Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering  
National Taiwan University

Java 405  
Spring 2024

```
1 class Lecture4 {  
2  
3     "Arrays and More Data Structures"  
4  
5 }
```

# Arrays

An array is an object which stores **multiple** values of the **same** type.

```
1 ...  
2     // Assume that T is any type and the size is known.  
3     T[] A = new T[size];  
4 ...
```

- We now proceed to explain Line 3 in two stages.

## Stage 1: Array Creation

- We first focus on the RHS of Line 3.
- One array is allocated in the **heap** by invoking the **new** operator followed by **T** and **[ ]** surrounding its size,
- Then its starting address is returned and should be cached.
- Note that the size **cannot** be changed after allocation.<sup>1</sup>

---

<sup>1</sup>What if the array is full?! Stay tuned.

## Stage 2: Reference

- We then declare one variable, say  $A$  in this case, to store the starting address of the array.
- I strongly emphasize that  $A$  is not the array, but the reference to the array!
- To understand the type correctly, one should read the type from right to left.
- For example,  $A$  is the reference to an array whose elements are of the  $\mathbf{T}$  type.
- Note that the array type is declared like  $\mathbf{T}[\ ]$  but without the size.

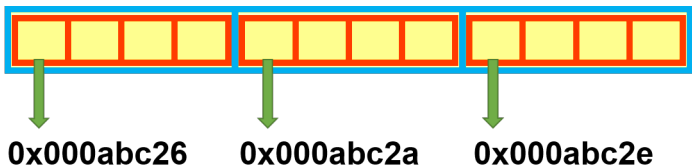
## Zero-Based Array Indexing

- We access any array element by using its index, which starts from 0 but not 1.
- Explicitly, the first element is  $A[0]$ , followed by  $A[1]$ ,  $A[2]$ , and so on. (Why?)
- So the last index of one array is  $size - 1$ .
- When the index is out of range, the program will fail due to the runtime exception **ArrayIndexOutOfBoundsException**.

## Memory Allocation for Arrays

- An array is **allocated contiguously** in the memory.
- To fetch the second element, jump to the address stored by  $A$  and shift by **1** unit size of  $\mathbf{T}$ , denoted by  $A[1]$ .
- For example,


```
1 ...  
2   int[] A = new int[3];  
3 ...
```



## Zero-Based Array Indexing (Concluded)

- You now could explain why  $A[0]$  denotes the first array element.
- Array index clearly acts as an **offset** from the starting address of the array!
- It is worth to mention that we can treat the whole memory as an array, indeed.
- This convention is applicable commonly among the mainstream languages!<sup>2</sup> (Why?)

---

<sup>2</sup>For example, C, C++, Java, JavaScript, and even Python. However, to the best of my knowledge, R and MATLAB manipulate arrays with the first index starting from 1. So it is just an option between choosing 0 and 1. 



# Array Initialization

- Every array is initialized implicitly once the array is created.
- Default values for different types are listed below:
  - 0 for all numeric types;
  - `\u0000` for `char` type;
  - `false` for `boolean` type;
  - `null` for all reference types.<sup>3</sup>
- An array can also be created by **enumerating** all elements without using the `new` operator, for example,

```
1 ...  
2     int[] A = { 10, 20, 30 }; // Syntax sugar.  
3 ...
```

<sup>3</sup>We will visit the keyword `null` in the chapter of OOP.

# Arrays & Loops

We often use `for` loops to process array elements.

- Arrays have the attribute called `length`, which indicates the array capacity.
  - For example, `A.length`.
- So it is natural to use a `for` loop to manipulate arrays.

# Examples

```
1 ...
2 // Create an integer array of size 5.
3 int[] A = new int[5];
4
5 // Generate 5 random integers ranging from 0 to 99.
6 for (int i = 0; i < A.length; ++i) {
7     A[i] = (int) (Math.random() * 100);
8 }
9
10 // Display all elements of A: O(n).
11 for (int i = 0; i < A.length; ++i) {
12     System.out.printf("%d ", A[i]);
13 }
14 System.out.println();
15 ...
```

- To show all elements, you need to iterate over the array by loops instead of simply printing *A*. (Why?)

```
1 ...
2 // Find maximum and minimum of A: O(n).
3 int max = A[0];
4 int min = A[0];
5 for (int i = 1; i < A.length; ++i) {
6     if (max < A[i]) max = A[i];
7     if (min > A[i]) min = A[i];
8 }
9 ...
```

- How to find the locations of extreme values?<sup>4</sup>
- Can you find the 2nd maximum value in  $A$ ?
- Can you track and maintain a record of the first  $k$  maximum values in  $A$ ?

---

<sup>4</sup>See also [Arguments of Maxima](#) (argmax) and [Arguments of Minima](#) (argmin).

```
1 ...
2     // Sum of A: O(n).
3     int sum = 0;
4     for (int i = 0; i < A.length; ++i) {
5         sum += A[i];
6     }
7 ...
```

- Calculate the following descriptive statistics:
  - the mean of  $A$ ;
  - the median<sup>5</sup> of  $A$ ;
  - the mode<sup>6</sup> of  $A$ .

---

<sup>5</sup>See <https://en.wikipedia.org/wiki/Median>.

<sup>6</sup>See [https://en.wikipedia.org/wiki/Mode\\_\(statistics\)](https://en.wikipedia.org/wiki/Mode_(statistics)).

## Alternative Way: for-each Loops

- A for-each loop is designed to **iterate** over a collection of objects, such as arrays and other data structures, in strictly sequential fashion, from start to finish.

```
1 ...  
2     T[] A = { ... };  
3     for (T element : A) {  
4         // Loop body.  
5     }  
6 ...
```

# Example

```
1 ...
2     int s = 0;
3     for (int i = 0; i < A.length; ++i) {
4         s += A[i];
5     }
6 ...
```

```
1 ...
2     int s = 0;
3     for (int item : A) {
4         s += item;
5     }
6 ...
```

- Short and sweet!
- You may consider using the for-each loop if you **iterate over all elements** and **the order of iteration is irrelevant**.

# Exercise

```
1 ...  
2     String[] letters = { "A", "B", "C", "D", "E" };  
3  
4     for (String letter: letters) {  
5         System.out.printf("%s ", letter);  
6     }  
7     System.out.println();  
8 ...
```



## More Examples (1/4): Cloning Arrays

- One might duplicate an array for some purpose, say a backup.
- For example,

```
1 ...
2     int x = 1;
3     int y = x; // You can say that y copies the value of x.
4     x = 2;
5     System.out.println(y); // Output 1.
6
7     int[] A = { 10, ... }; // Ignore the rest of elements.
8     int[] B = A;
9     A[0] = 100;
10    System.out.println(B[0]); // Output?
11 ...
```

- The result differs from our expectation. (Why?)
- This is called the **shallow copy**.

- To clone an array, you should create a new array and use loops to copy every element, one by one.

```
1 ...
2     // Let A be an array to be copied.
3     int[] B = new int[A.length];
4     for (int i = 0; i < A.length; ++i) {
5         B[i] = A[i];
6     }
7 ...
```

- This is called the **deep copy**.
- Alternatively, you may use the method **System.arraycopy()** for the same purpose.

```
1 ...
2     // Assume that B is ready.
3     System.arraycopy(A, 0, B, 0, A.length);
4 ...
```

## More Examples (2/4): Shuffle Algorithm

```
1 ...
2     for (int i = 0; i < A.length; ++i) {
3
4         // Choose a random integer j.
5         int j = (int) (Math.random() * A.length);
6
7         // Swap A[i] and A[j].
8         int tmp = A[i];
9         A[i] = A[j];
10        A[j] = tmp;
11
12    }
13 ...
```

- However, this naive algorithm is fundamentally broken!<sup>7</sup>
- How to swap by using XOR (that is,  $\wedge$ )?

<sup>7</sup>See <https://blog.codinghorror.com/the-danger-of-naivete/>.

## Exercise

Write a program to deal the first 5 cards from a deck of 52 shuffled cards.

- As you can see, RNG produces only random numbers.
- How to shuffle nonnumerical objects?
- Simply label 52 cards by  $0, 1, \dots, 51$ .
- Shuffle these numbers!

```
1 ...
2     String[] suits = { "Club", "Diamond", "Heart", "Spade" };
3     String[] ranks = { "3", "4", "5", "6", "7", "8", "9",
4                       "10", "J", "Q", "K", "A", "2" };
5
6     int size = 52;
7     int[] deck = new int[size];
8     for (int i = 0; i < deck.length; i++)
9         deck[i] = i;
10
11     // Shuffle algorithm: correct version.
12     for (int i = 0; i < size - 1; i++) {
13         int j = (int) (Math.random() * (size - i)) + i;
14         int z = deck[i];
15         deck[i] = deck[j];
16         deck[j] = z;
17     }
18
19     for (int i = 0; i < 5; i++) {
20         String suit = suits[deck[i] / 13];
21         String rank = ranks[deck[i] % 13];
22         System.out.printf("%2s %-7s\n", rank, suit);
23     }
24 ...
```

## More Examples (3/4): Sorting Problem

- In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain **order**.
- You may call **Arrays.sort(A)** to rearrange *A* in ascending order, for example,

```
1 import java.util.Arrays;
2
3 ...
4     int[] A = { 5, 2, 8 };
5     Arrays.sort(A); // Becomes { 2, 5, 8 }.
6
7     String[] B = { "www", "csie", "ntu", "edu", "tw" };
8     Arrays.sort(B); // Result?
9 ...
```

- Note that we sort strings in lexicographical (dictionary) order for most cases.

## Exercise: Bubble Sort

```
1 ...
2 // Bubble sort: O(n ^ 2).
3 boolean swapped;
4 do {
5     swapped = false;
6     for (int i = 0; i < A.length - 1; i++) {
7         if (A[i] > A[i + 1]) {
8             int tmp = A[i];
9             A[i] = A[i + 1];
10            A[i + 1] = tmp;
11            swapped = true;
12        }
13    }
14 } while (swapped);
15 ...
```

- Try to implement [selection sort](#) and [insertion sort](#).<sup>8</sup>

<sup>8</sup>See <https://visualgo.net/en/sorting>.

## More Examples (4/4): Searching Problem

- It is often to query one **key** for its corresponding **value**.
- For example, the program plans to find one client's credit card number.
- In this case, the client name is the query key and his/her credit card number is the value associated.



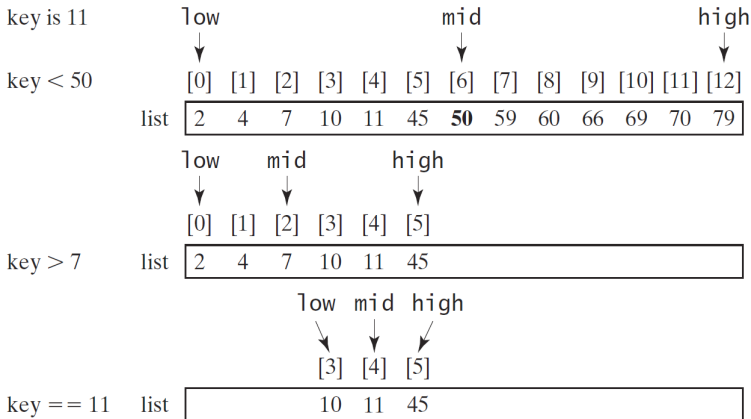
## Solution 1: Linear Search

- **Linear search** compares the query key with all elements in sequential order.

```
1 ...
2     // Linear search: O(n).
3     int[] A = { ... };
4     int founds = 0;
5     for (int i = 0; i < A.length; i++) {
6         if (A[i] == key) {
7             System.out.printf("%d ", i);
8             founds++;
9         }
10    }
11    System.out.println("\nFounds: " + founds);
12 ...
```

- Could we do better?

## Solution 2: Binary Search (Revisited)



```

1 ...
2     int idx = -1; // Why?
3     int high = A.length - 1, low = 0, mid;
4     while (high > low && idx < 0) {
5         mid = low + (high - low) / 2; // Why?
6         if (A[mid] < key)
7             low = mid + 1;
8         else if (A[mid] > key)
9             high = mid - 1;
10        else
11            idx = mid;
12    }
13
14    if (idx > -1)
15        System.out.printf("%d: %d\n", key, idx);
16    else
17        System.out.printf("%d: not found\n", key);
18 ...

```

- It can be shown that binary search runs in  $O(\log n)$  time.
- However, binary search works only for **ordered** data!

## Discussions

Scenario / Operation	Insert	Search
Immutable unsorted array	N/A	$O(n)$
Immutable sorted array	N/A	$O(\log n)$
Mutable unsorted array	$O(1)^*$	$O(n)$
Mutable sorted array	$O(n)$	$O(\log n)$

\*: insert by attaching behind the array.

- Assume that the data is immutable (unchangeable).
- We sort the data once for all and the binary search works well.
- What if the data may be changed all the time?
- Is it possible to make it run in  $O(1)$  time for both operations?<sup>9</sup>

<sup>9</sup>See [https://en.wikipedia.org/wiki/Hash\\_table](https://en.wikipedia.org/wiki/Hash_table).

## Short Introduction to Data Structures

- A data structure is a particular way of **organizing** data in a program so that it can perform **efficiently**.<sup>10</sup>
- **The choice for data structures depends on applications.**
- As an alternative to arrays, **linked lists**<sup>11</sup> are used to store data in the way different from arrays.
- You will see plenty of data structures in the future.<sup>12</sup>
  - For example, trees, graphs, tables, and more.
- You could also find a huge number of questions about data structures on [LeetCode](#).

---

<sup>10</sup>See <http://bigocheatsheet.com/>.

<sup>11</sup>See [https://en.wikipedia.org/wiki/Linked\\_list](https://en.wikipedia.org/wiki/Linked_list).

<sup>12</sup>See [Introduction to Collections](#) by Oracle and [Java Collections Framework](#) from Wikipedia.

## Beyond 1D Arrays

- 2D or higher dimensional arrays are widely used in various applications.
  - For example, RGB images are stored as 3D arrays.
- We can create 2D **T**-type arrays simply by adding one more [ ] with its size.
- For example,

```
1 ...  
2     int rows = 4; // Row size.  
3     int cols = 3; // Column size.  
4     T[][] M = new T[rows][cols];  
5 ...
```

- It is similar to create 3D or higher-dimensional arrays.

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

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

(a)

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

matrix[2][1] = 7;

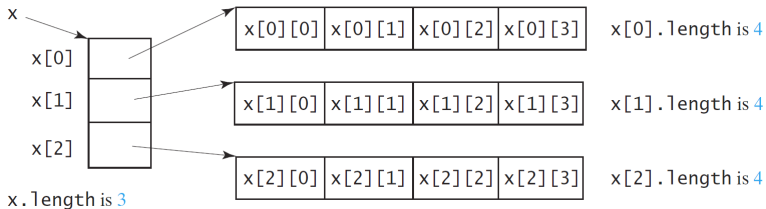
(b)

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

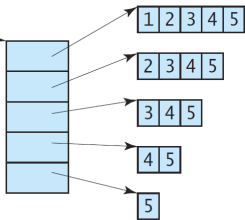
```
int[][] array = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
    {10, 11, 12}
};
```

(c)

## Memory Allocation for 2D Arrays



```
int[][] triangleArray = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```





## Example: 2D Arrays & Loops<sup>13</sup>

```
1 ...
2     int[][] A = { { 10, 20, 30 }, { 40, 50 }, { 60 } };
3
4     // Conventional for loop.
5     for (int i = 0; i < A.length; i++) {
6         for (int j = 0; j < A[i].length; j++)
7             System.out.printf("%3d", A[i][j]);
8         System.out.println();
9     }
10
11    // For-each loop.
12    for (int[] row : A) {
13        for (int item : row)
14            System.out.printf("%3d", item);
15        System.out.println();
16    }
17 ...
```

<sup>13</sup>Thanks to a lively discussion on January 31, 2016.

## Exercise: Matrix Multiplication

Let  $A_{m \times n}$  and  $B_{n \times q}$  be two matrices for  $m, n, q \in \mathbb{N}$ . Write a program to calculate  $C = AB$ .

- Let  $a_{ik}$  and  $b_{kj}$  be elements of  $A$  and  $B$ , respectively.
- For  $k = 1, 2, \dots, n$ , use the formula

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

for  $i = 1, 2, \dots, m$  and for  $j = 1, 2, \dots, q$ .

- Following the formula, it takes  $O(n^3)$  time. (Why?)

## Digression: ArrayList

```
1 ...
2     int[] A = new int[3]; // The size should be known in advance.
3     A[0] = 100;
4     A[1] = 200;
5     A[2] = 300;
6     for (int item : A)
7         System.out.printf("%d ", item);
8     System.out.println();
9
10    ArrayList<Integer> B = new ArrayList<>(); // Size?
11    B.add(100);
12    B.add(200);
13    B.add(300);
14    System.out.println(B); // Short and sweet!
15 ...
```

- Arrays are the simplest form of data structures but not convenient to use.
- For example, resizing arrays can be costly when you frequently move data to a newly created, larger array. (Why?)
- So it is advisable to use **ArrayList**<E>, where E is the **type parameter**.
- Using angle brackets < · > in Java is called the **generics** starting from JDK5 in 2004.

## Digression: Generics

- Generics are widely used in data structures, like **Stack**<T>, **Map**<K, V>, **Graph**<V, E>, etc.<sup>14</sup>
- To use **ArrayList**<E> correctly, we need to replace E with **Integer**, which is the wrapper class<sup>15</sup> for **int** values.
- Be aware that only reference types can substitute the type parameters.
- This technique is also utilized in C++ and C#.

---

<sup>14</sup>See also [Generics](#) by Oracle. Stay tuned in [Java Programming 2](#).

<sup>15</sup>See [The Numbers Classes](#).

## Case Study: Order Reversing

- How to rearrange an input array in reverse order?
- Let  $A$  be an integer array.
- The first attempt is to create another array with same size and **copy** each element from  $A$  to  $B$ .

```
1 ...
2     int[] A = { 1, 2, 3, 4, 5 };
3     int[] B = new int[A.length];
4     for (int i = 0; i < A.length; i++) {
5         B[A.length - 1 - i] = A[i];
6     }
7     A = B; // Why?
8 ...
```

## Another Attempt

```
1 ...
2     int[] A = { 1, 2, 3, 4, 5 };
3     for (int i = 0; i < A.length / 2; i++) {
4         int j = A.length - 1 - i;
5         int tmp = A[i];
6         A[i] = A[j];
7         A[j] = tmp;
8     }
9 ...
```

Approach	Time Complexity	Space Complexity
1st attempt	$O(n)$	$O(n)$
2nd attempt	$O(n)$	$O(1)$

- The second is better in both time<sup>16</sup> and space.
- This is an **in-place** algorithm.

<sup>16</sup>It runs in only half time of the first attempt.