“Array” Data Structure

An Illustrative Introduction

*(These slides are inspired by 'Data Structures- An Illustrative Introduction by **Kamran Ahmed**, founder of https://roadmap.sh/ , Google Developer Expert and a GitHub Star)*

In this article, we will discuss about array data structure in Java.

What is an array ?

An array is a collection of items stored consecutively in the memory. An array has three main features:

- All items in an array must be of the same data type.
- An array is indexed, meaning that we can access each item by its position in the array.
- An array has a fixed size, meaning that once we create an array, we cannot change its size.

We have a memory representation below, in which each yellow block represents a slot in the memory.

representation of a memory

Now, lets initialise one array of integers and another of characters;

```
int[] numbers = {2, 5, 9, 1, 0};
char[] characters = { 'R', 'O', 'A', 'D', 'M', 'A', 'P' };
```

The items of each array are stored in each block right next to each other in the memory.

each item is stored consecutively in the memory

All items in array need to have same data type, and each array has to have a fixed size.

an array should be fixed in size and all items must be of same data type

Why does an array items need to have same data type ?

```
int[] numbers = {4, 9, 6};                    // 2 bytes
boolean[] states = {false, true};             // 1 byte
char[] characters = {'A', 'G', 'U'};          // 1 byte
```

Lets initialise three arrays, one of Integer type which takes 2 bytes of memory, second of type Boolean which takes 1 byte and third of Character type which takes 1 byte of memory. Now lets see how the Integer type which takes 2 bytes of memory is stored in a memory representation in which each block or slot hold 1 bytes of memory.

```
int[] numbers = {4, 9, 6};                // 2 bytes
```

memory representation of Integer type array.

Since we know an integer takes 2 bytes of memory, each item in the 'numbers' array takes two blocks in the memory representation. When accessing these items from the memory, our program knows that the array is an array of Integers, so, it reads two bytes starting from the first and keeps on reading every next two bytes to complete the traversing.

Now, lets see what happens if we have mixed data types in an array.

```
mixed[] =  {4,      "BMW",     true};
         (2 bytes)  (3 bytes) (1 byte)
```

memory representation of mixed type array.

We have an integer which takes 2 bytes, a string which takes 3 bytes and a boolean which takes 1 byte. When reading values from this array, since it is of mixed types, our program does not know how many bytes should it read to get the first, second and the third value.

However, in languages like Ruby and JavaScript, we can have an array of mixed types with dynamic size. This is because Ruby and JavaScript are dynamically typed languages, which means that the type of a value is determined at runtime rather than at compile time. This allows them to have arrays that can store values of different types because the type of an element can be changed at any time. Lets see an example what happens in the memory;

boxing the size of an array by JS

JavaScript can represent an array of mixed type values with the help of 'Boxing' of the size. JS will find the maximum size of an element, in our case is the string "BMW" which take 3 bytes. And that size is allocated to each of the other elements in the array. In our case, JS will allocate a total of 9 bytes of memory for the 'mixed' array.

memory representation of a JS array

Why does an array need to have a fixed size ?

```java
3  public class Main {
4
5      public static void main(String[] args) {
6          // initialising an array to hold 3 integers
7          int[] numbers = new int[3];
8          numbers[0] = 10;
9          numbers[1] = 59;
10         numbers[2] = 11;
11         System.out.println(numbers.length); // prints 3
12
13         // adding one more integer
14         numbers[3] = 594;
15         System.out.println(numbers.length); // Error
16
17     }
18 }
19
```

Console Terminal

```
<terminated> Main (41) [Java Application] /Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java  (3 Jan 2023
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
        at arrays.Main.main(Main.java:14)
```

adding data to a Java array

Here, when tried to add the fourth item in the 'numbers' array defined to have only 3 items, the program throws an "ArrayIndexOutOfBoundsException" error. Lets see what happens in the memory representation;

```
int[] numbers = new int[3];
                numbers[0] = 10;
                numbers[1] = 59;
                numbers[2] = 11;
```

fixed array size in memory

The program allocates 3 slots in the memory and populates the values in those slots. Since, the computer's memory is not only specific to us, and it needs to store the data for other programs as well and give its slots to them.

```
int[] numbers = new int[3];
numbers[0] = 10;
numbers[1] = 59;
numbers[2] = 11;
numbers[3] = 294; // not allowed
```

memory is used by other program as well

This is why we have arrays which are fixed in size.

We have to define the size of an array before using it, so the computer can reserve the consecutive memory blocks for the array to use. However, we might wonder why cannot we defined a large size of our 'numbers' arrays so we can push new numbers whenever we need. The reason why we cannot do that is, we will be keeping all those memory slots for ourselves even if we do not need them. This will not let the other programs to use those memory slots which will cause "Memory Leak".

```
int[] numbers = new int[1000];
                numbers[0] = 10;
                numbers[1] = 59;
                numbers[2] = 11;
```

memory leak

However, in JavaScript, it is possible to have an array of dynamic size. Lets see why is it possible;

array in JS

The 'numbers' array be allocated in three blocks of the memory and other blocks can be used by other programs.

adding new item in a JS array

When we pushed a new item to the 'numbers' array, the program checks if there is a space for the item and if yes, it will allocated that space for the new item.

adding more items without consecutive space available

If we try to push a new number, it cannot be consecutively added to the memory as the block is already occupied my another program. In that case, the program tries to find a whole new space in the memory where it can consecutively populate the space with all the items , moves all the items to the space, frees up the previous space and pushes the new item to the new consecutive space;

new space for the extended array

What is meant by an array is indexed ?

Each element in the array is assigned a unique numerical position, or index, starting from 0. We can use the index to access an element in the array directly, by specifying the index in square brackets after the array name.

```java
public class Main {

    public static void main(String[] args) {

        int[] numbers = { 1, 5, 7, 2, 9 };

        int firstNumber = numbers[0];
        int thirdNumber = numbers[2];

        System.out.println("First Number is: " + firstNumber);
        System.out.println("Third Number is: " + thirdNumber);
    }
}
```

```
Console   Terminal
<terminated> Main (41) [Java Application] /Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java (3 Jan
First Number is: 1
Third Number is: 7
```

indeces of an array

Using an index to access an element in an array is generally very fast, because the index corresponds directly to the position of the element in the array. This makes arrays very efficient for certain types of operations, such as looking up a value based on its position in the collection.

When to use arrays?

- When we need to store a large number of values that are all of the same data type, such as a list of names or a list of prices.

- When we need to be able to access any element in the collection quickly, using its index. For example, if we need to look up a value based on its position in the collection.

- When we need to be able to efficiently iterate over all the elements in the collection. For example, if we need to perform the same operation on every element in the collection.

- When we need to store a fixed number of elements and we know in advance how many elements we will need.

When not to use arrays ?

- When we need to store values of different data types.

- When we need to frequently insert or delete elements from the collection. Arrays are not very efficient for these operations because they require shifting all the elements after the insertion or deletion point to make room or close the gap.

- When we don't know in advance how many elements we will need to store. Arrays have a fixed size, so if we need to store more elements than the array can hold, we will need to create a new array and copy all the elements over, which can be time-consuming.

What are array methods in Java ?

- We can use the '**length**' field to get the size of an array. For example:

```
int[] numbers = {1, 5, 7, 2, 9 };
System.out.println(numbers.length); // prints 5
```

- We can also use the '**fill**' method to set all the elements of an array to the same value. For example;

```
int[] numbers = new int[5];
Arrays.fill(numbers, 1);
System.out.println(numbers.length); // prints 5
System.out.println(numbers[0]); //prints 1
System.out.println(numbers[4]); //prints 1
```

- We can use the '**sort**' method to sort the elements of an array in ascending order. For example;

```
int[] numbers = { 1, 5, 7, 2, 9 };

Arrays.sort(numbers);

for (int i = 0; i < numbers.length; i++) {
        System.out.println(numbers[i]);          // prints
};
                                                   2
                                                   5
                                                   7
                                                   9
```

- We can use the '**binarySearch**' method to search for a specific element in a sorted array. For example;

```
int[] numbers = { 1, 5, 7, 2, 9 };
Arrays.sort(numbers);
int index = Arrays.binarySearch(numbers, 7);
System.out.println(index);  //prints 3
```

- We can use the '**copyOf**' method to create a copy of an array. For example;

```
int[] numbers = { 1, 5, 7, 2, 9 };
int[] copy = Arrays.copyOf(numbers, numbers.length);
System.out.println(copy[1]); //prints 5
```

- We can use the '**toString**' method to convert an array to a string representation. For example;

```
int[] numbers = { 1, 5, 7, 2, 9 };
String str = Arrays.toString(numbers);
System.out.println(str);  //prints [1, 5, 7, 2, 9]
```

These are pretty much all the basic concepts on "Array" data structure and how an array is computed by the program under the hood.

Thank you all for reading.

Please give review and insights, if I have mistaken about something.

Keep Learning !!