12 Lec ① 5,6,7 Day 09

Introduction to Arrays and Arraylist in Java

int[] ros; // declaration of array. ros is getting defined in the Stack.

Note Carefully! ←

new uses for creating object

ros = new int[5]; // actually here object is being created in the memory (heap)

Initialization ⇒

new used to create an object

note Carefully

int[] arr = new int[5];

Data type

reference variable

Stores in Stack

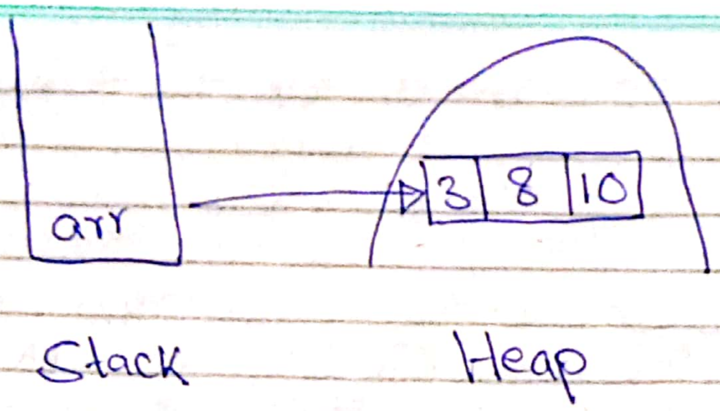
Happens at compile time

creating the object in the heap memory

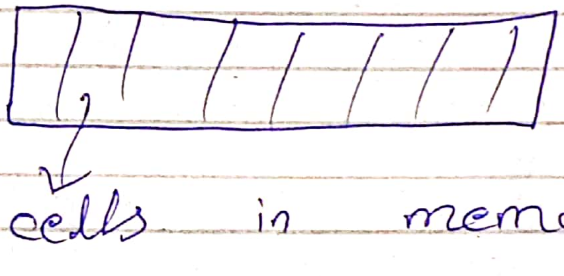
Happens at RunTime

(Dynamic Memory Allocation

12 Lec (2) Day 09



Array in C/C++, is basically continuously memory allocation.



-> This is how arrays working in C/C++.

-> In Java, There is no concept of pointers. We can't get address of anything.

So It totally depends on the JVM, You know whether this is going to continuous or not.

(i) array objects are in heap.

(2) heap object are not continous.

(3) Dynamic Memory Allocation.

Hence:

Array objects in Java may not be continuous.

Traditional Array Defination

- It's continuous data

Java Array Defination

-> may not be continuous

-> It depend's on JVM.

Confusion: Clear JVM Concept

Note:

-> Let suppose we have created an int array by default of length 5. each element have value "0" by default

-> String array have by default "null" of each ~~type~~ element's value

null: It is not basically type, It is basically Literal.

⇒ We can assign null only to non-primitive (breakable types) types, otherwise error will occur.

⇒ Any reference variable that we have bydefault, It is going to be "null" value. (for non-primitive type)
↳ confirm pls!

Note Carefully! Memory Management

→ primitives are stored in the Stack memory only.

→ But all the objects (String type, array type, our own type, Hashmaps, all the classes) are stored in the heap memory.

⟷

→ Strings are im-mutable in Java

→ Arrays are mutable in Java.
↳ means we can change the object.
↳ through function
↳ Passing array as argument, then value change in function.
Accessing change value below function calling.

2D Array

```
int[][] arr = new int[3][];
```

no. of rows, necessary to specify

no of columns, not necessary to specify.

```
Arrays.toString(arr2D);
```

will print whole array.

getting values from user in 2D array

```
for (int row = 0; row < 3; row++ {
        for (int col=0; col <2; col++){
                arr[row][col] = input.in;
        }
}
```

Using enhanced for loop

```
for (int[] a : arr){
        System.out.println (Arrays.toString(arr2D);
}
```

Note:
① Arrays are fixed sized

② Arraylist is similar to vectors in C++. (Dynamic Arrays

ArrayList

Syntax:

```
ArrayList<Integer> list = new
                         ArrayList<>();
list. add(5);
list-contains(5)    //True
list.set(0,6);
list.add(22);
list.get(0);   // 6
list.remove(1);
```

Actual happening of arraylist

① Size is fixed Internally

② Say arraylist fills by some amount

=> It will create a new arraylist of letsay, may be double the Size.

=> Old elements are coppied in new one

=> old one is deleted.

This what happens.

$O(1)$

Constant time Complexity

MultiDimentional Arraylist

```
ArrayList<ArrayList<Integer>> list =
                       new Arraylist<>();
```

Initialization → Note Carefully:

```
for(int i=0; i<3; i++){
    list.add(new Arraylist<>());

// add elements

for(int i=0;i<3; i++){
        for(int j=0;j<3; j++){
            list.get(i).add(in.nextInt());
        }
}

System.Out.println(list);
```

Note Carefully!

i) Integer to String

a) Integer.toString(22); // "22"
b) String.valueOf(myInt);
c) "22".toString

ii) String to Integer

a) Integer.parseInt("22"); //22

iii) Character to Integer

a) Character.getNumericValue('2'); // 2

iv) String (number) to bigInteger

a) BigInteger number = new BigInteger("1234");

O(n) time Complexity

i) For Small arrays, O(n) algorithm may be sufficient, but as the size of the array grows larger, time taken to search for the element also grows linearly.

ii) In general, an algorithm with time complexity of O(n) is considered to be efficient, as it scales well with increasing input sizes. However, It may still be necessary to optimize the algorithm furthur if it is being used for particularly large inputs, as even a linear growth rate can result in long processing times for very large inputs.

→ Carefully Note above two points

Greedy Solution = O(n)

In-place Algorithm:

In-place algorithms usually overwrite their input with output, no additional Space is needed.

→ Google Loves asking dynamic programming Questions

→ House Robber is best problem to Solve in dp manner.