JAVA BASIC

27 May 2024 10:04

Largest Number Problem

Q: Assume 3 numbers,

1) a = 10
2) b = 20
3) c = 30

Largest Num?

Soln:

Assume a is the Largest ∴ max = ~~10~~ 30

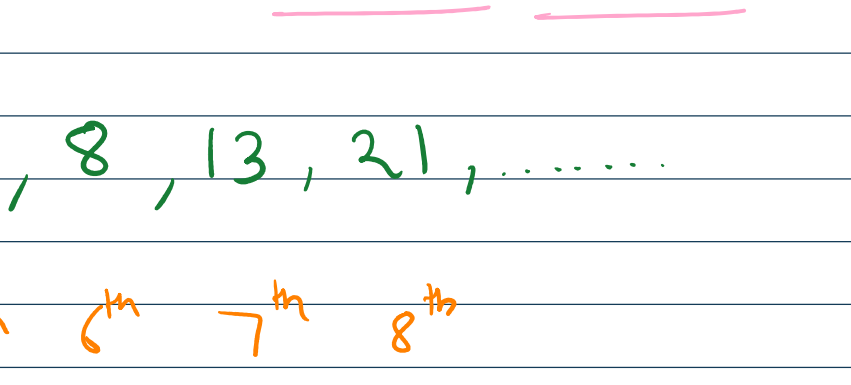
Check with b & c

If b > max:

max = b

If c > max

max = c



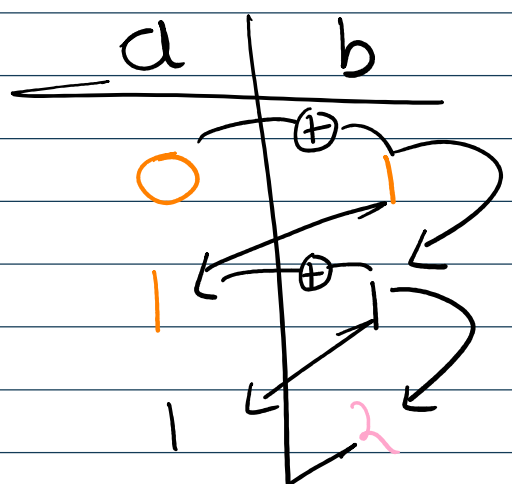
Fibonacci Numbers

→ Starts with 0,1. Series then continues by adding the previous 2 numbers. This series is known as Fibonacci Series

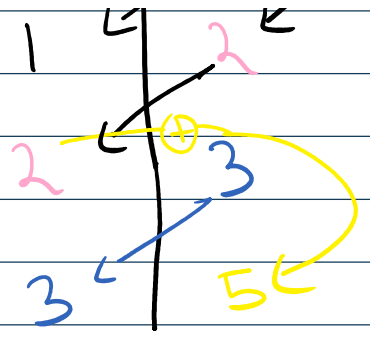
a = 0
b = 1

For eg Q is to find 7th position of Fibonacci Series, hence n = 7

Thus, loop will run till n=7 & since we know till what number we will run the loop, For loop will be used



Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...



p	i
p = 0	i = 1
temp 1	i+p 1
temp 1	i+p 2
2	3
3	5

```
while (count<=n)
{
    int temp = i
    i = i + p
    p = temp
    Count ++.
}
Sout(i);
```

Counting of Occurences

Eg: 123787567591 Find no of occurences of number "7"

Logic: Every num % 10 gives the remainder as the last digit & the last digit will obviously not exist

For eg: 13839 Count No of 3's

```
    1383
10 ) 13839
     13830
     -----
         9
```

Next Step

```
    1380
10 ) 1383
     1380
     ----
        3   -> Rem is 3
```

increase the cant

Likewise, proceed till num does not become 0.

Reverse

Eg Number is 23597 ∴ Output should be: 79532

Logic: Take remainder of num & divide it with 10 ie num % 10. Every time you get a number, multiply

num % 10. Every time you get a number, multiply that by 10 & add that by next number you will reverse

i.e firstly 23597 % 10 will be 2359
next, 2359 % 10 will give 235
235 % 10 will give 23
23 % 10 will give 2
2 % 10 will give 2

O/P
7
7*10+9 = 79
79*10+5 = 795
795*10+3 = 7953
7953*10+2 = 79532

Reversed string

Calculator Program

Switch Case Statements

→ == V/S .equals

eg. a ⟶ Kunal
b ⟶ Kunal
.equals() will give TRUE

In this case "==" value will give TRUE, because 'b' are referencing the 'a' same object

eg. a ⟶ Kunal
b ⟶ Kunal

Here "==" will give FALSE, since a & b are referencing diff objects

But, .equals() will still give TRUE because the value remains the same.

In switch statements, you can jump to various cases based on your expression.

Syntax:

```
switch (expression) {

    // cases
    case one:
        // do something
        break;

    case two:
        // do something
        break;

    default:
        // do something
}
```

NOTE:
- cases have to be the same type as expressions, must be a constant or literal
- duplicate case values are not allowed
- break is use to terminate the sequence
- if break is not used, it will continue to next case
- default will execute when none of the above does
- if default is not at the end, put break after it

```java
import java.util.Scanner;

public class Switch {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String fruit = in.next();

        switch(fruit) {
            case "Mango":
                System.out.println("King of Fruits");
                break;
            case "Apple":
                System.out.println("King of Fruits");
                break;
            case "Orange":
                System.out.println("King of Fruits");
                break;
            case "Grapes":
                System.out.println("King of Fruits");
                break;
            default:
                System.out.println("Aha Tamatar bade mazedaar");
        }
    }
}
```

Old Syntax of Switch case

```java
import java.util.Scanner;

public class Switch {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String fruit = in.next();

        switch (fruit) {
            case "Mango" -> System.out.println("King of Fruits");
            case "Apple" -> System.out.println("King of Fruits");
            case "Orange" -> System.out.println("King of Fruits");
            case "Grapes" -> System.out.println("King of Fruits");
            default -> System.out.println("Aha Tamatar bade mazedaar");
        }
    }
}
```

Enhanced Switch Case syntax (Doesn't require 'break')

```java
int day = in.nextInt();
switch (day) {
    case 1 -> System.out.println("Monday");
    case 2 -> System.out.println("Tuesday");
    case 3 -> System.out.println("Wednesday");
    case 4 -> System.out.println("Thursday");
    case 5 -> System.out.println("Friday");
    case 6 -> System.out.println("Saturday");
    case 7 -> System.out.println("Sunday");
}
```

Program 2: Displaying Day name btwn 1 and 7

```java
switch (day) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        System.out.println("Weekday");
        break;
    case 6:
    case 7:
        System.out.println("Weekend");
        break;
}
```

Program 3: Displaying Weekdays and Weekends in switch case style

Nested Switch Case

Code on Github Repo

Methods in Java

→ Bundling up of code in some sort of function / method such that reusability principle is used here → code is also efficient to use (DRY) principle should be used.

→ functions are that block of codes that run only when they are

→ functions are that block of codes that run only when they are called. Here, we define the code once → use it many times

Syntax:

```
public class main{

    access_modifier return_type method(){
        // code

        return_statement  (Code ends)
    }
```

- method() ⟶ Calling the fcn in code

return_type

→ Firstly, the function call (eg. sum()) will have some value which it takes from the fcn.
→ Secondly, the content returned in that fcn will be the final value which will be displayed when we call the fcn.

Eg. of return types. int, char, bool, void
↳ returns nothing

⇒ there are a few important things to understand about returning the values:

- The type of data returned by a method must be compatible with the return type specified by the method.
  eg: if return type of some method is boolean, we cannot return an integer.
- The variable recieving the value returned by a method must also be compatible with the return type specified for the method.

```java
public class Sum {
    public static void main(String[] args) {
//          sum();
        int ans = sum2();
        System.out.println(ans);
    }

    //return value

    static int sum2(){  1 usage
        Scanner in = new Scanner(System.in);
        System.out.print("Enter num1: ");
        int num1 = in.nextInt();
        System.out.print("Enter num2: ");
        int num2 = in.nextInt();
        int sum = num1 + num2;
        return sum;
    }

    //doesn't return anything
    static void sum(){  no usages
        Scanner in = new Scanner(System.in);
        System.out.print("Enter num1: ");
        int num1 = in.nextInt();
        System.out.print("Enter num2: ");
        int num2 = in.nextInt();
        int sum = num1 + num2;
        System.out.println("Sum is "+sum);
    }
}
```

This shld be also compatible with the type returned

Both shld be of same type

→ Fxn over, is the age much nahi rhaloga


```java
import java.util.Scanner;

public class Greeting {
    public static void main(String[] args) {
//          greeting();
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String naam = in.next();
        String personalized = myGreet(naam);
        System.out.println(personalized);
    }

    static String myGreet(String name) {  1 usage
        String Greeting = "Hello " + name;
        return Greeting;
    }
```

Shouldn't be neccesarily same. Can be different too & it won't affect code.

```java
public class Swap {
    public static void main(String[] args) {
//          System.out.println(a + " "+b);

        // swap numbers code
//          int temp = a;
//          a=b;
//          b=temp;
        swap(a,b);
        System.out.println(a + " "+b);
    }

    static void swap(int a , int b){  1 usage
        int temp = a;
        a=b;
        b=temp;
    }
}
```

→ Will these values swap

Answer: No, they won't.

```java
    String name = "Kunal Kushwaha";
    changeName(name);
    System.out.println(name);
}

static void changeName(String name) {
    name = "Rahul Rana";
}
```

→ Will the name change to "Rahul Rana"

Answer: no, it won't

Let's understand why the changes won't happen

```java
package com.kunal;

public class PassingExample {
    public static void main(String[] args) {
        String chacha = "Iron Man";
        greet(chacha);
    }

    static void greet(String naam) {
        System.out.println(naam);
    }
}
```

This won't care ki upar ka name kya hai. It can be different

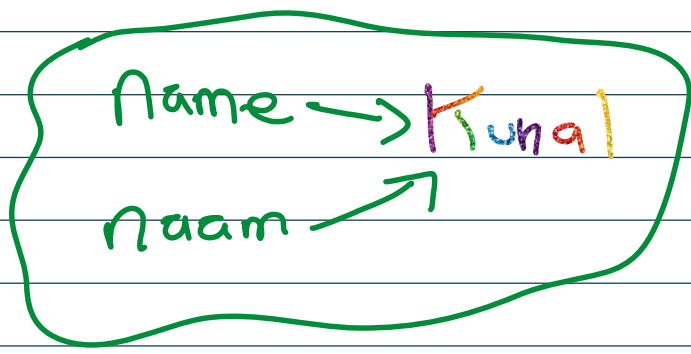
main(){

name = "Kunal"

greet(name), → Value of name is passed here
(Value of reference variable is passed)

Copy is created & passed to "naam". Thus both point at same object

}



greet(naam) {

Print(naam)

}

In Java, there is only the concept of Pass By Value. Pass by Reference isn't present here
Thus in this example only a copy of that Reference is passed

Let's take below example

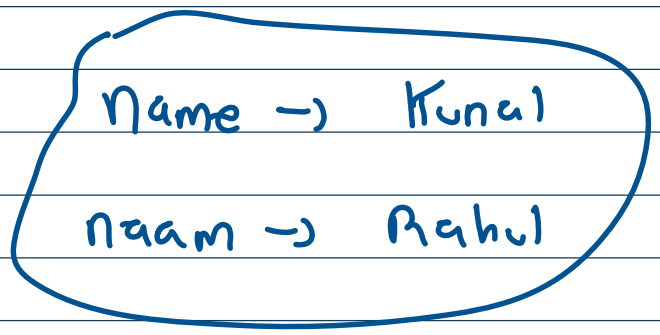
```java
String name = "Kunal Kushwaha";
changeName(name);
System.out.println(name);
}

static void changeName(String naam) {
    naam = "Rahul Rana";
}
```

// not changing creating a new obj

only accessible inside the fun & not outside the fun

Here, the o/p will be "Kunal Kushwaha" & not "Rahul Rana"

Why ??



In any way, the "name" isn't changing so how will the o/p change. This just means that the value isn't updated & instead a new object is created here

Pts to remember

Pts to remember

1] for primitives: int, char, short, bool, ...

⇓

Just pass by value

2) for objects etc stuff: Passing the value of reference

i.e name → 'a' ← naam pointing to same obj

Eg(2)

```
psvm(){
    a=10
    b=20
    Swap(a,b)
}

Swap(int num1, int num2)
{
    temp = num1
    num1 = num2
    num2 = temp
}
```

[a → 10
  b → 20]

2] But not here

[temp → 10
  num2 → 10
  num1 → 20]

1] Values change at this level of Scope

Why? Because it is only passing the value

Let's look at an example now where the change in one reference variable changes other i.e. here there is Pass by value of reference variable

a → 10 ← b Change in one brings change in other

```
1   import java.util.Arrays;
2
3   public class ChangeValue {
4       public static void main(String[] args) {
5
6           int[] arr = {1 , 3 , 2 , 45 ,6};
7           change(arr);
8           System.out.println(Arrays.toString(arr));
```

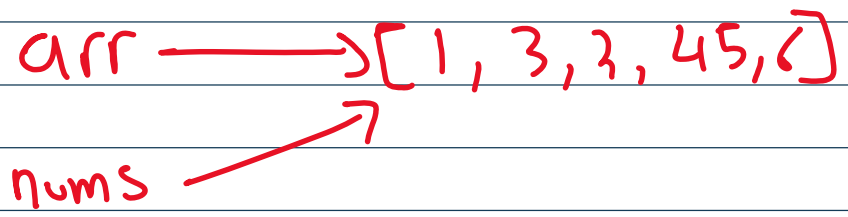
Q. Will the 0th position in main fxn change?

→ Ans: Yes

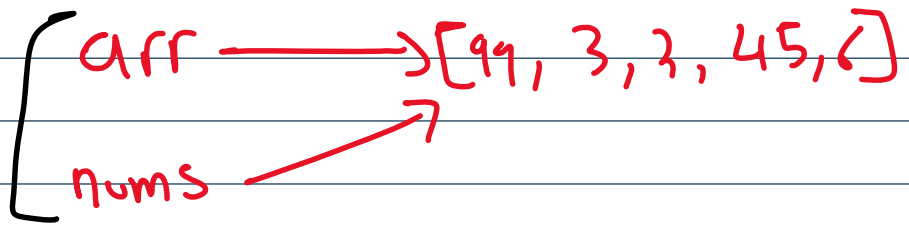
Let's see how

```java
        change(arr);
        System.out.println(Arrays.toString(arr));

    }

    static void change(int[] nums){  1 usage
        nums[0] = 99;
    }
}
```

Let's see how



nums[0] = 99



Thus, "arr" values also change.

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaage
[99, 3, 2, 45, 6]

Process finished with exit code 0
```

⇒ O/P

```java
        String name = "Kunal Kushwaha";
        changeName(name);
        System.out.println(name);
    }

    static void changeName(String naam) {
        naam = "Rahul Rana"; // creating a new object
    }
```

→ Reason why String wasn't changing here.

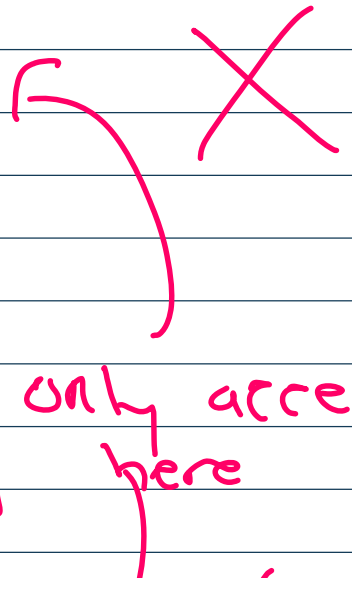
Also, "Strings are immutable"

Scopes

1. Function Scope: Variables declared inside a method can't be accessed outside the fxn. Only accessible inside the fxn

Eg.

```
psum(){
    int a = 10,
    String b = "Hello",

static void greet(){

    int m = 50;
    String greet = "Good day";
```



3

3

X

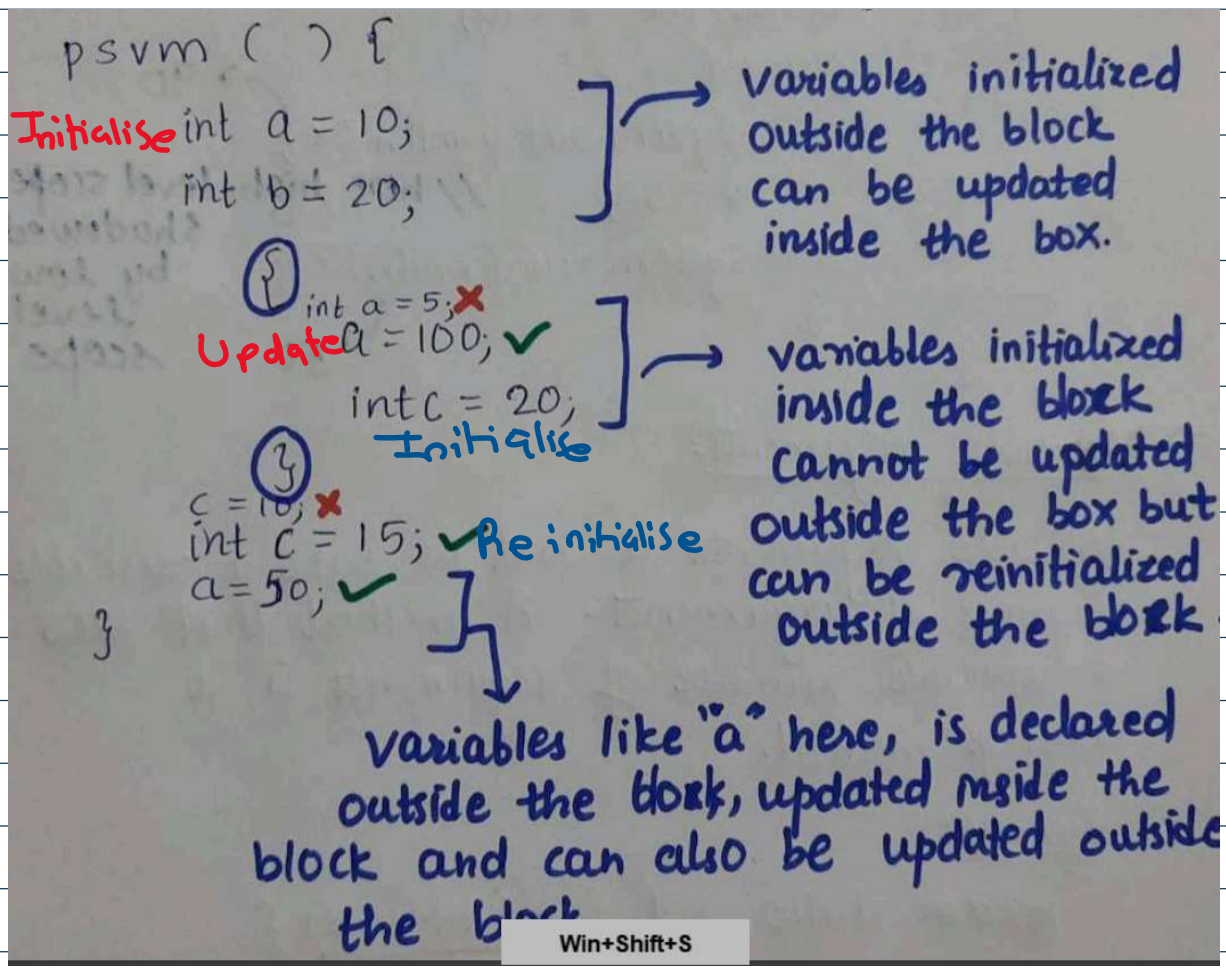
2] Block Level Scope

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

        int a = 10;
        int b = 20;
        {
//          int a = 30; // already initialised outside the block in the same method , hence cannot initialose agai
            a = 80; // reassgin original ref variable to other value
            int c = 99;
            System.out.println(a);
            // values initialised inside the block remain inside the block
        }
        System.out.println(a);
//      System.out.println(num); Will give an error since variable is inside the function

    }

    static void random(int marks) {  no usages
        int num = 67;
        System.out.println(num);
        System.out.println(marks);
    }
}
```

1] We can reassign original of variables to other values. The value also changes outside the block too since we aren't creating any new variable, we are just updating it.

2] int c=99; This thing is initialized inside block & thus can't be used outside block. To use outside the block, we can reinitialize it again outside the block

Anything which is initialised outside the block can be used inside the block and the value can be updated inside the block but anything which is initialized inside the block cannot be used outside the block. We will have to reinitialize it outside the block.



3) Loop Scope: Variables declared inside a loop will have loop scope

For eg. For(int i=0, i<=5, i++){

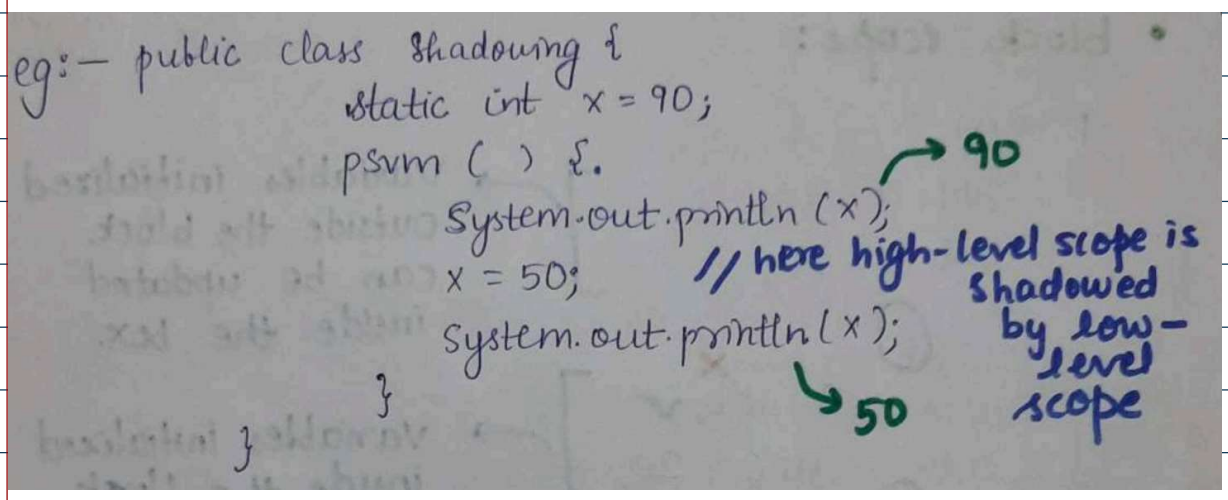
//code

}

'i' can't be accessed outside the loop.

Shadowing

→ Practice of using variables in overlapping scope with same name. The variables in low-level scope here override the variable of high level scope. Thus variable of high level scope is shadowed by low-level scope variable.




```java
package com.kunal;

public class Shadowing {
    static int x = 90; // this will be shadowed at line 8
    public static void main(String[] args) {
        System.out.println(x); // 90
        int x; // the class variable at line 4 is shadowed by this
//        System.out.println(x); // scope will begin when value is initialised
        x = 40;
        System.out.println(x); // 40
        fun();
    }

    static void fun() {
        System.out.println(x);
    }
}
```

The shadowing only occurs when the variable is initialised & not just declared If we just declare it & not initialise it then printing it will give error

Variable Args

→ Variable Args are basically when you don't know how many number of args you want to pass in a function. A method that takes variable arguments is varargs method.

Syntax :

```
static void funC(int ...v){
//body
}
```

This internally takes it as an array of integer or any other data type specified here

3

```java
import java.util.Arrays;

public class VarArgs {
    public static void main(String[] args) {
        fun( ...v: 2,3,4,8,9);
    }

    static void fun(int ...v){  1 usage
        System.out.println(Arrays.toString(v));
    }
}
```

→ Mtne bhi args chalege idhar, no restriction since varargs is used

```java
import java.util.Arrays;

public class VarArgs {
    public static void main(String[] args) {
        fun( ...v: 2,3,4,8,9);
        multiple( a: 2, b: 5, ...v: "Kunal");
    }

    static void multiple(int a , int b , String ...v){ 1 usage
        System.out.println(a+ " "+b);
        System.out.println(Arrays.toString(v));
    }

    static void fun(int ...v){ 1 usage
        System.out.println(Arrays.toString(v));
    }
}
```

Varags are always at the end of the list

Function Overloading

→ Happens when 2 fcn's have the same name but diff number of arguments in them

eg → 1) `fun ( ) { //code }` × `fun ( ) { //code }` — function overloading

2) `fun (int a) { //code }` `fun (int a, int b) { //code }` — This is allowed having different arguments with same method name.

In fcn overloading, either the args should be different or the type of arguments should differ (If there's only 1 argument in both fcn's)

Armstrong Number

→ Armstrong number is that number in which if we take cube of all the individual digits, then the sum of that is also the same number before

i.e $a = 153$

$(1)^3 \quad (5)^3 \quad (3)^3$

$1 + 125 + 27 = 153$

Armstrong Number

Logic:

$a = 153$

```
while (q>0){

    rem = q % 10;
    Cube the remainder

    Sum += Cube
     q = q/10;
```

Arrays

→ Whats an Array?

```
// Q: store a roll number
int a = 19;

// Q: store a person's name
String name = "Kunal Kushwaha";

// Q: store 5 roll numbers
int rno1 = 23;
int rno2 = 55;
int rno3 = 18;
```

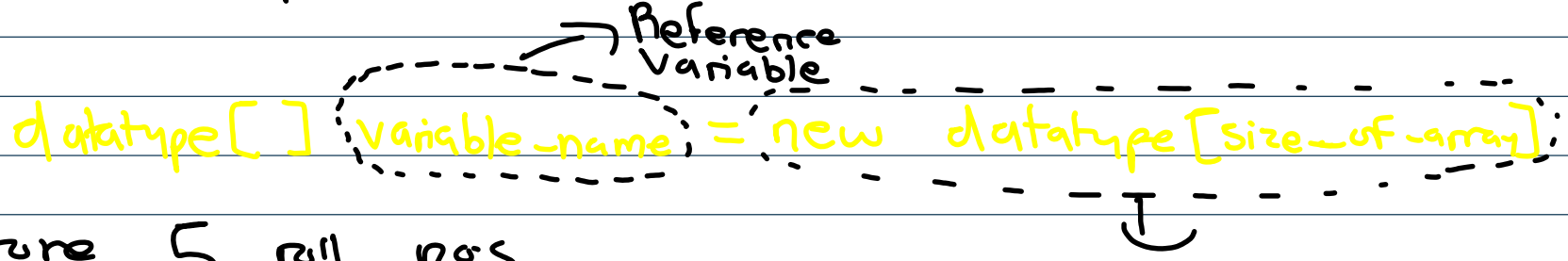
Imagine a scenario where we are storing roll numbers of 5 students

What if we want to store roll numbers of 50 or 500 students?

We use Array Data Structure for that.

Array : A Data structure used to store collection of Data (maybe simple datatypes or complex data types or primitives).

Syntax :



Eg. Store 5 roll nos

```
int[] rnos = new int[5]
```

↑ Size of array, stores 5 roll nos

OR

```
int[] rnos = {51, 82, 83, 99, 104}
```

Elements of the array.

Internal working of Array:

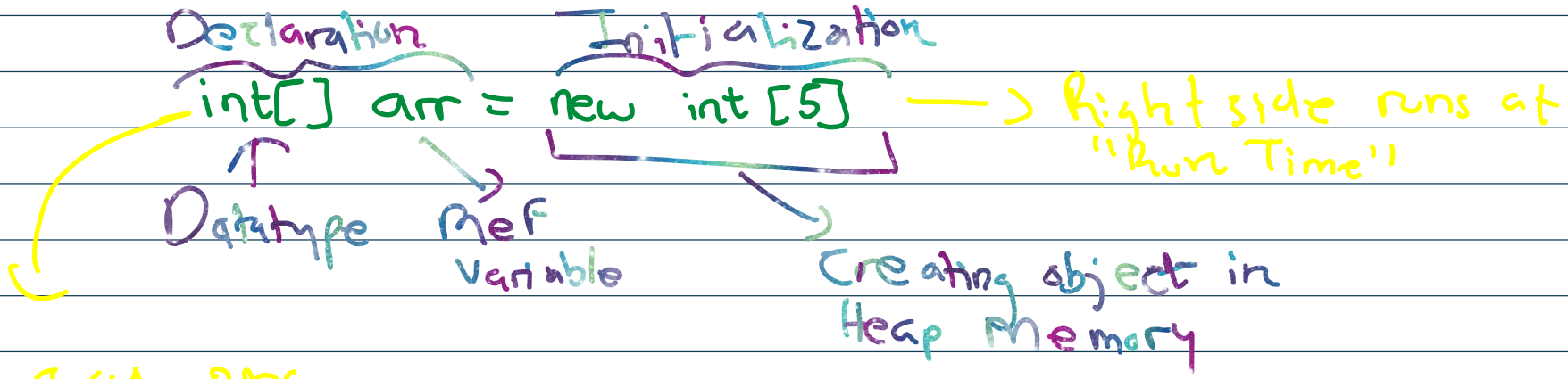
```
int rollnos[];
```

Decleration of the array.

↳ rollnos are getting defined in the stack

`rollnos = new int[5];` Initialization

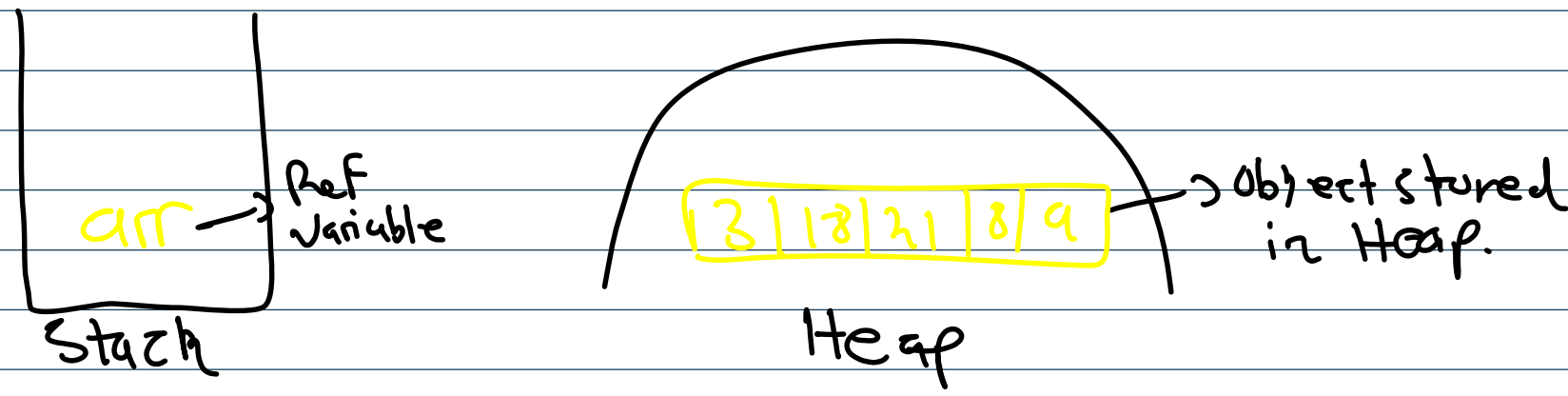
Actual memory allocation happens here
Objects is created in Heap memory.



→ Above concept is known as "Dynamic Memory Allocation"

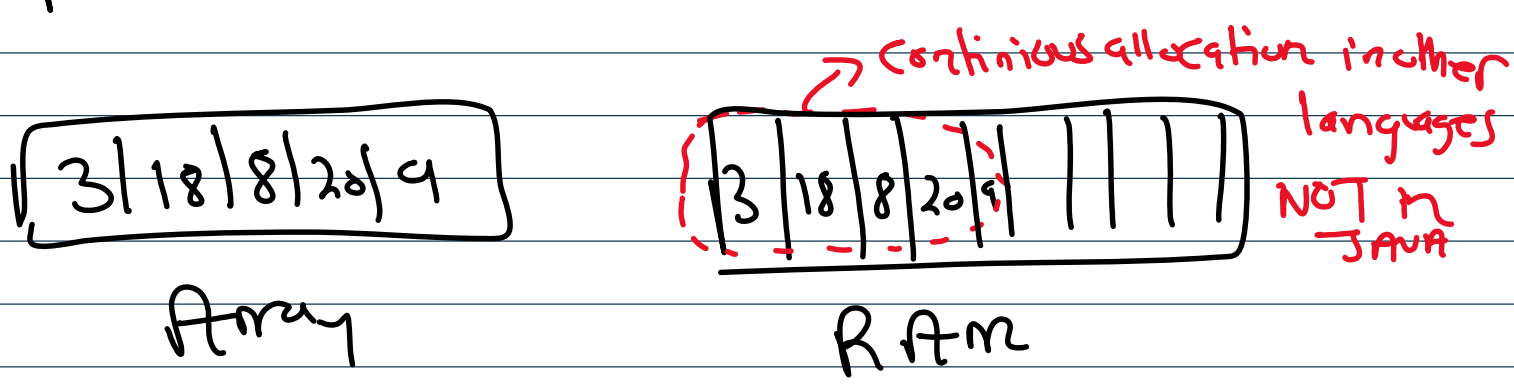
↓

Memory is allocated at runtime



→ In C/C++ or any other prog. lang, array data is stored as continuous data. But in Java, the array data may or may not be allocated continuously

→ It totally depends on JVM whether to be continuous or not



Reasons:

1] Objects are stored in Heap Memory.
2] Heap objects aren't continuous (mentioned in JLS)
↓
Java Language Specification

3] Dynamic Memory Allocation

3] Dynamic Memory Allocation

Thus, array objects may or may not be continuous.
(Totally depends on JVM)

Index of Array

0	1	2	3	4	5	6
3	8	19	12	7	28	33

arr[0] = 3
arr[1] = 8
arr[2] = 19
arr[3] = 12

Suppose we want to change value of certain index

arr[3] = 99

0	1	2	3	4	5	6
3	8	19	99	7	28	33

new keyword

→ int[] arr = new int[5]

⇓

Used to create an object
Will create an object in heap memory of size 5

If no values are provided, then the default stored values will be [0,0,0,0,0] →for size 5

String arr[] = new String[4]

By default, "null" is the output (Similar to "None" of Python)

```
String str = null;
int num = null;
```

⇓

Will give error since null can't be assigned to primitive data types.

- Primitives are stored in Stack memory (int, char, bool, etc)
- All other objects (array, any other type) are stored in Heap memory.

String [] arr = new String [5];

// internal working of object

arr[0] = "S"

arr[0] = null

no value initially hence null

```
for(int num : arr) { // for every element in array, print the element
    System.out.print(num + " "); //  here num represents element of the array
}
```

num → variable of for loop
arr → array

This loop is known as Enhanced – For loop (For-Each loop)

Another best way for printing the array is using toString() method.

```
36
37          System.out.println(Arrays.toString(arr));
38      }
39  }
```

```
Run  Main
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
1 2 3 4 5
[1, 2, 3, 4, 5]

Process finished with exit code 0
```

Internally uses the for loop but gives the o/p in proper format (with comma, bracket)

Array of objects

```
// array of objects

String[] str = new String[5];
for (int i = 0; i < str.length; i++) {
    str[i] = in.next();
}

System.out.println(Arrays.toString(str));
```

2D Array.

2D Array.

→ Represented by a Matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Syntax : `int[][] arr = new int[][]` → optional

Size ↑

`[]` → no. of rows (mandatory, i.e can't leave it blank)

`[]` → column

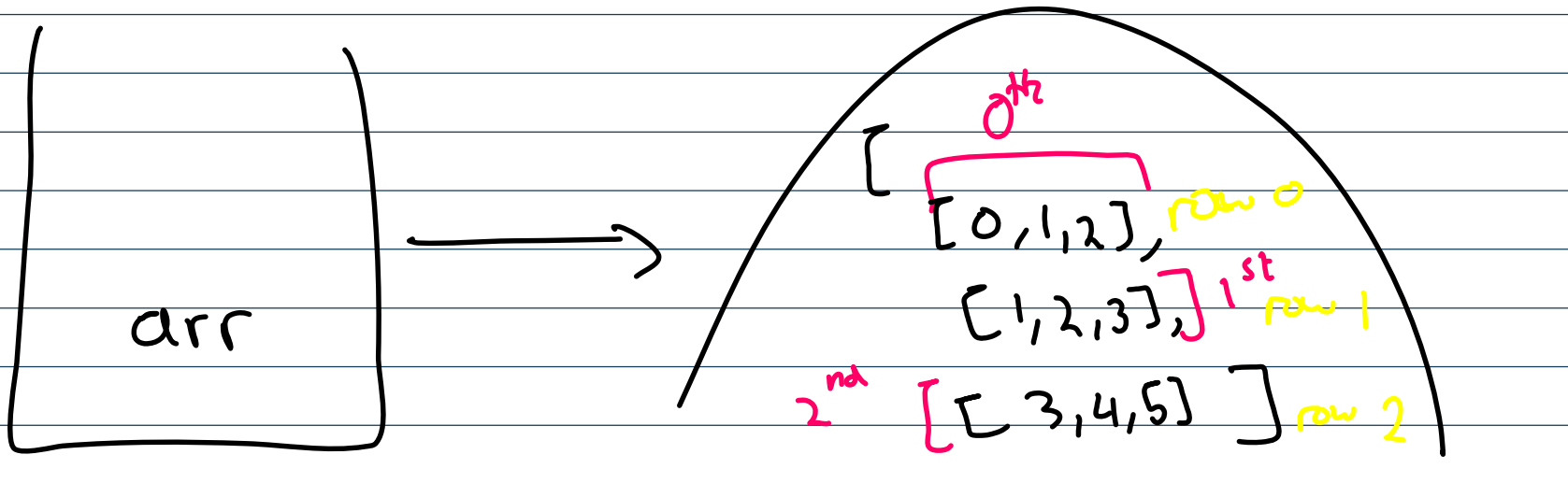
1)

OR

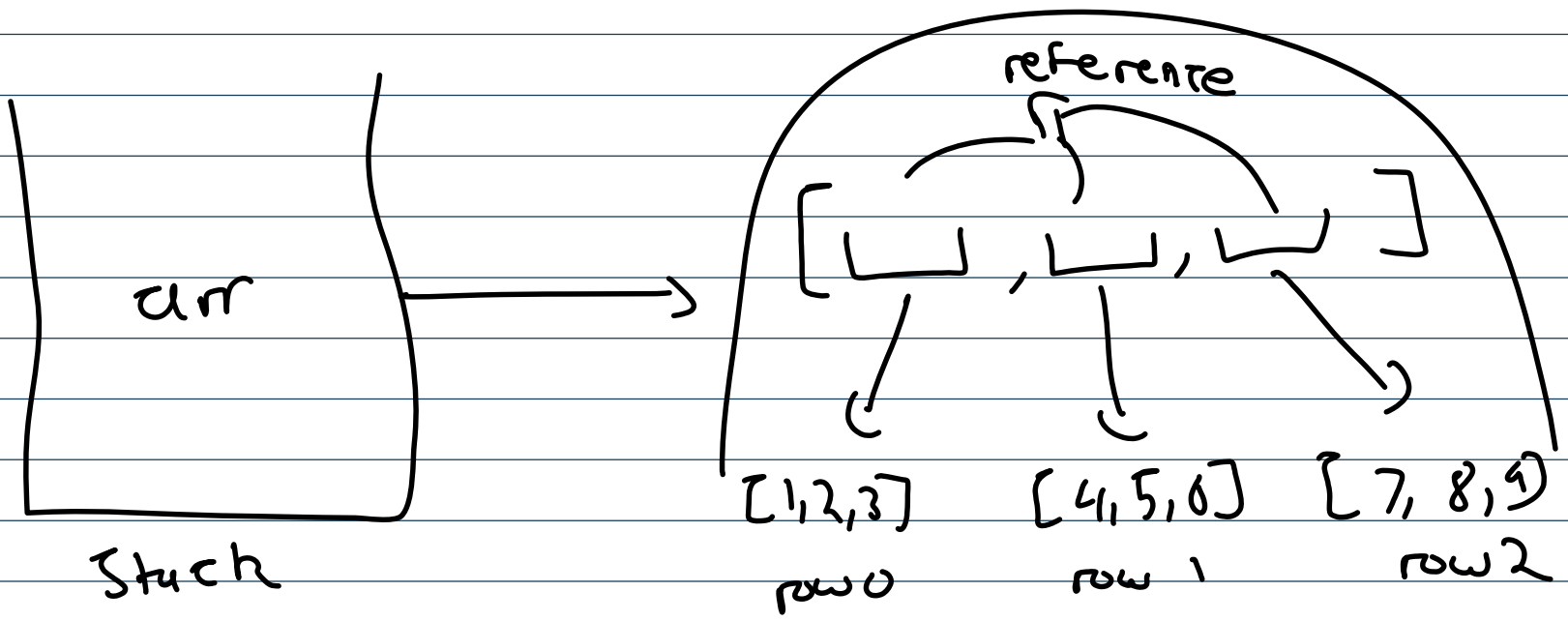
2)

```
int[][] arr = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
              };
```

3



Imagine this as Array of Arrays



arr[1] = [4,5,6]
arr[1][0] = 4

Column size dosen't matter because:

```
int[][] arr2D = {
        {1, 2, 3},
        {4, 5},
        {6, 7, 8, 9}
};
```

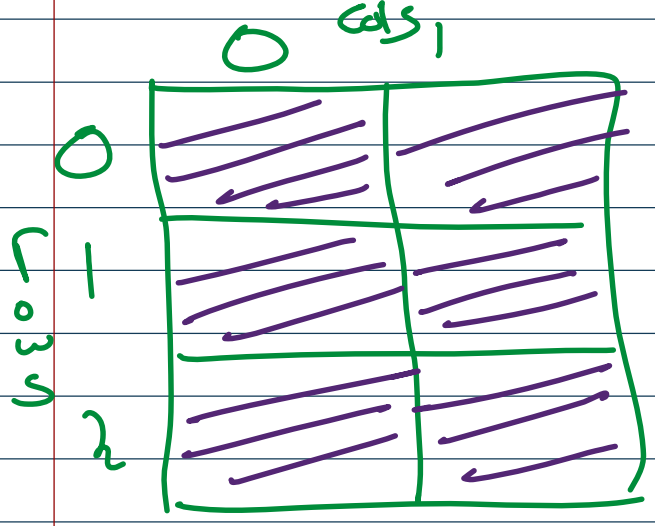
each array can have diff sizes

Input of 2D Array:

int[][] arr = new int[3][2]
Sout(arr.length) → Prints only no of rows

```
for(int row=0; row < arr.length(); row++){
    // for each column in every row
    for(int col=0; col < arr[row].length; col++){

        arr[row][col] = in.nextInt();
    }
}
```

Working:



[2]*[2] array

```
for(int row=0; row<3; row++)
{
    // now we take every row
    for(int col=0; col<3; col++)
    {
        arr[row][col] = in.nextInt();
    }
}
```

row<3
0<3
col<3
0
1
2

[0][0]
[0][1]
[1][0]
⋮

3
3

Entire DataType array since inside every 2D array, there is a 1D array.

arr[row].length() → Individual size of columns

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

Enhanced for loop o/p for 2D Array

Array List

→ Part of collection framework & is present in Java.util.package.
→ Provides us with dynamic arrays in Java
→ Used when we don't know what size do we have to take in

→ Provides us with dynamic arrays in Java
→ Used when we don't know what size do we have to take in an array.

Syntax:

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

Wrapper Classes

Capacity

you can add a datatype here. NOT mandatory

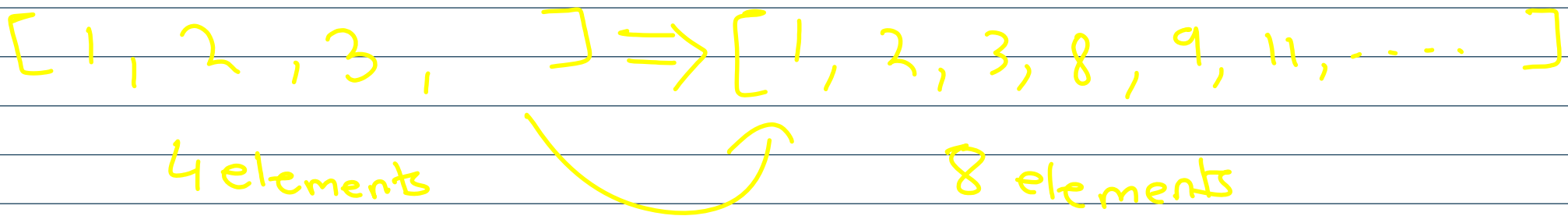
→ Slower than the standard array.

Internal working of ArrayList:

① The size is fixed internally

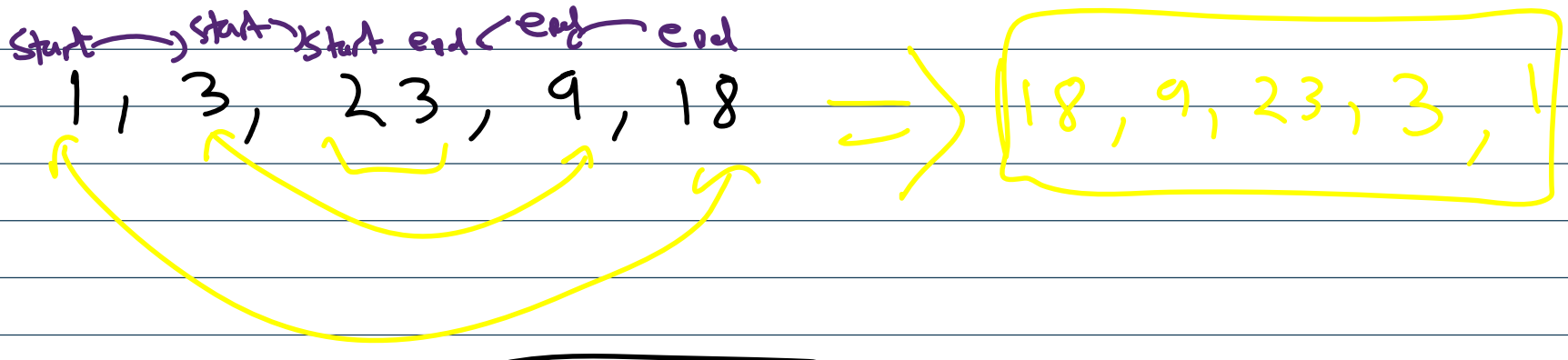
② If the array gets filled by some amount (eg 50%), then

⇒ It will create a new ArrayList of Double the size (eg.)
⇒ old elements are copied in new Array List
⇒ old Arraylist is deleted.



Process keeps going on until input is stopped

○ Reversing an Array



Loop ends when start > end

Arrays in Java

◇ An array is a collection of similar data type values

- An array is a collection of similar data type values.
- Syntax: `datatype[] variable_name = new datatype[size];`

OR

```
datatype[] variable_name;

variable_name = new datatype[size];
```

OR

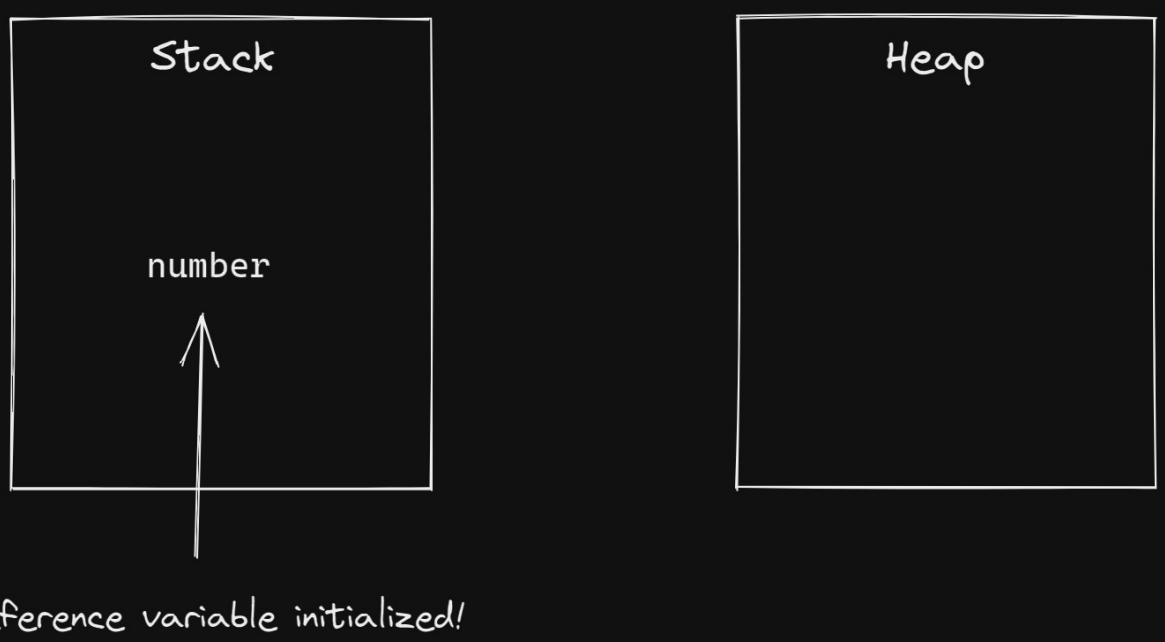
```
datatype[] variable_name = {value1, value2, value3, . . ., valueN};
```

Understanding the Syntax

- `datatype[] variable_name;`

This step will initialize the variable and it will get initialized in the stack during compile time.

Example: `int[] number;`

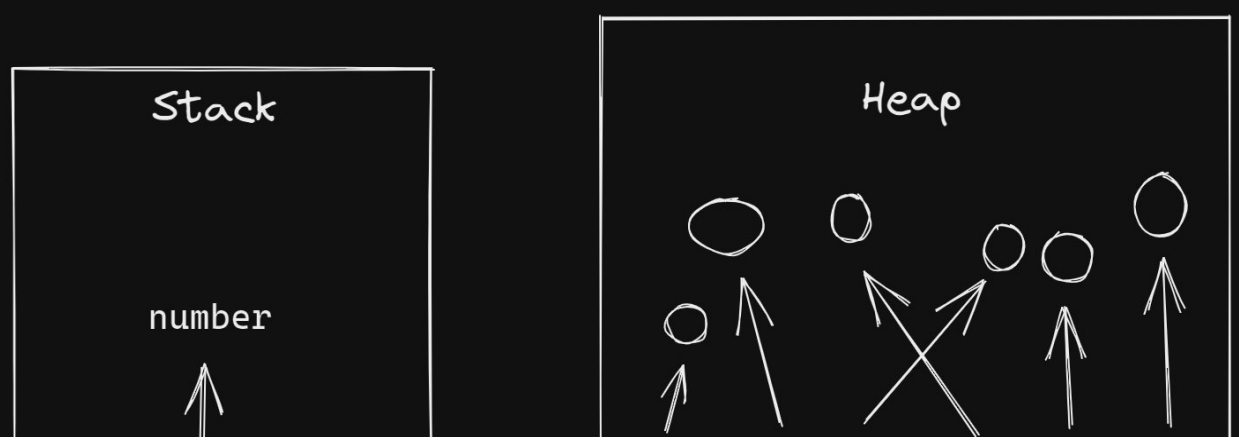


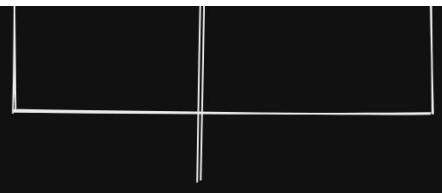
- `variable_name = new datatype[size];`

In this step, a new object will be created in the heap memory during runtime.

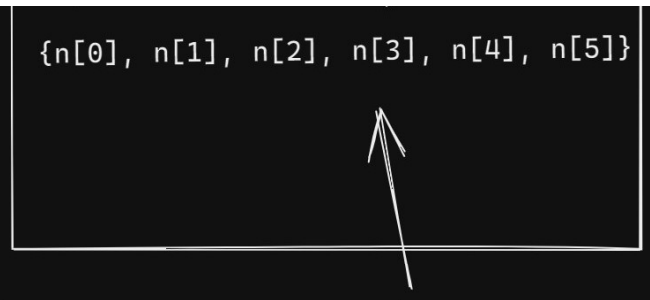
The 'new' keyword is used to create a new object.

Example: `n = new int[6];`





Reference variable initialized!



A new object is created with a size of six reference variables

If these reference variables have nothing to point to, they will return 'null' when called.

Arrays in Java

2D Arrays

◇ A 2D array can be visualized as a matrix. Let's understand how?

◇ → First of all, let's take a 1D array like this,

```
int[] num = {2, 5, 6, 9, 3};
```

→ Now, write this array vertically like this,

```
int[] num = {2,
             5,
             6,
             9,
             3};
```

rows = 5

→ Then, replace each element with an array like this:

```
int[][] num = {
              {1, 2, 3, 4},
              {5, 6, 7, 8},
              {9, 10, 11, 12},
              {13, 14, 15, 16},
              {17, 18, 19, 20}
            };
```

(2D array)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

rows = 5

columns = 4

total elements = 5*4 = 20

→ You can also do something like this:

```
int[][] num = {
              {1, 2, 3, 4},
              {5, 6, 7},
              {8, 9, 10, 11},
              {12, 13},
              {14}
            };
```

(dynamic array)

rows = 5
columns = dynamic

◇ Syntax: `datatype[][] variable_name = new datatype[row_size][column_size];`

OR

```
datatype[][] variable_name;
variable_name = new datatype[row_size][column_size];
```

OR

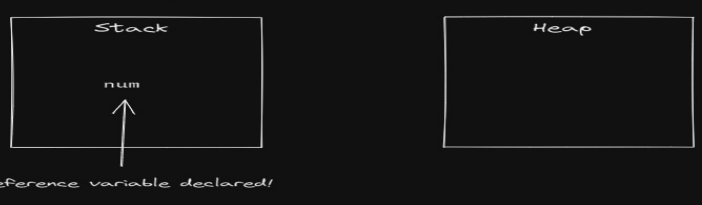
```
datatype[][] variable_name = {{array1}, {array2}, {array3}, . . ., {arrayN}};
```

Understanding the Syntax

◇ `datatype[][] variable_name;`

This step will declare the variable and it will be declared in the stack during compile time.

Example: `int[][] num;`

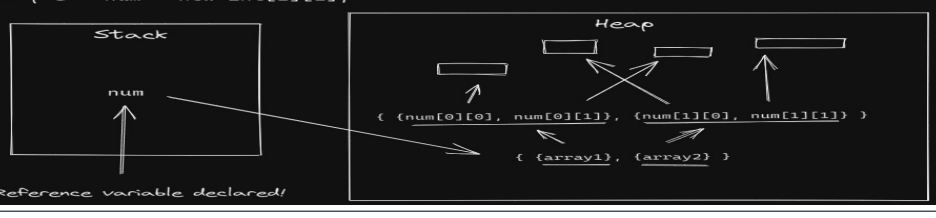


Reference variable declared!

◇ `variable_name = new datatype[row_size][column_size];`

In this step, a new object will be created/initialized in the heap memory during runtime. The 'new' keyword is used to create a new object.

Example: `num = new int[2][2];`



Reference variable declared!

ArrayList in Java

First of all let's understand the difference between Arrays and ArrayList

Array	ArrayList
◇ An Array has a fixed length or size.	◇ An ArrayList can contain as many elements as you want even though an initial size is specified.
◇ An Array can be created using both primitive datatypes as well as non-primitive datatypes.	◇ An ArrayList cannot be created using primitive datatypes. It can only be created using objects or wrapper classes.

What is an ArrayList?

Let's say you don't want to set a fixed size or length of an array because either you don't know how many elements you may add or you may want to update the size in the future.

What will you do? That's where ArrayLists come into action.

Syntax

```
ArrayList<Integer> list = new ArrayList<Integer>(10);
```

- `ArrayList` → Class
- `<Integer>` → Datatype (Wrapper Class) [not primitive]
- `list` → reference variable
- `=` → creates a new object
- `new ArrayList<Integer>` → ArrayList constructor
- `(10)` → Initial size

Some methods

- ◇ `add()` :- Adds a new element to the ArrayList
- ◇ `set(index, value)` :- Updates an existing value for a specified index
- ◇ `get(index)` :- Used to retrieve an existing value for a specified index

Internal Working

Actually the size of the ArrayList is fixed but not permanently.
It can change according to the input you provide.

Example:-

◇ You provide the input for the first time and the initial size is set to 5

◇ Then, you decide that you want to add another four elements to the ArrayList.
But we don't have enough space. So, what will Java do?

The answer is, it will create a new list with a new size (it depends) enough to accommodate new elements.

Then, it will copy the old list to the new list and will delete the old list.

It might look like this: