PROGRAMMING LANGUAGES

Procedural:

* Specifies a series of well-structured steps and procedures to compose a program.
* Contains a systematic order of statements functions and commands to complete a task.

Functional:

* Writing a program only in pure functions that is modify variables, but only create new ones as an output.
* Used in situations where we have to perform lots of different operations on the same set of data life ML
* First class function? Function is used as variable

Object oriented:

* Revolves around objects
* Code + data=objects
* Developed to make it easier to develop, debug, reuse and maintain software.

Static and dynamic languages:

Static :

* Perform type checking at compile time (Compilation means converting the source code that is human readable code into machine code)
* Errors will show at compile time
* Declare data type before you use it
* More control

Dynamic:

* Perform type checking at runtime (Run time means after compilation when the program is running)
* Error might not show till program is run
* No need to declare data type of variables
* Saves time in writing code but might give error at runtime

Memory in programming:

Memory management: there are two types of memory Heap memory and stack memory

Let us take example a = 10 here ‘a’ is the reference variable,,, ‘10’ is the object ………………... this ‘a’ reference variable is stored in Stack memory and ‘10’ object stored in the heap memory …… reference variables in stack memory will point towards the heap memory….

Eg :

a=[1,2,3,4,5,6,7,8,9,10]  
b=a  
a[0]=99  
print(a)  
print(b)

out put:

[99, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[99, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let us take ‘a’ as a reference variable which has a list of numbers and the values of ‘a’ or objects of ‘a’ is pointed to ‘b’………in this case when there is a change in a ‘a’ then it will reflect the b

A

1,2,3,4,5,6,7,8,9,10

B

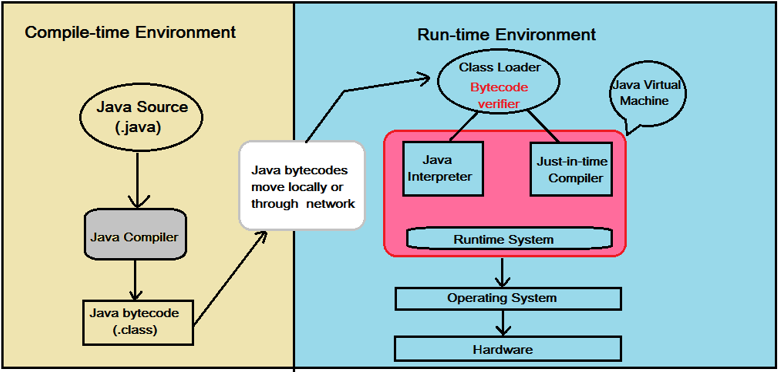
If the value of A changes A[0]=99

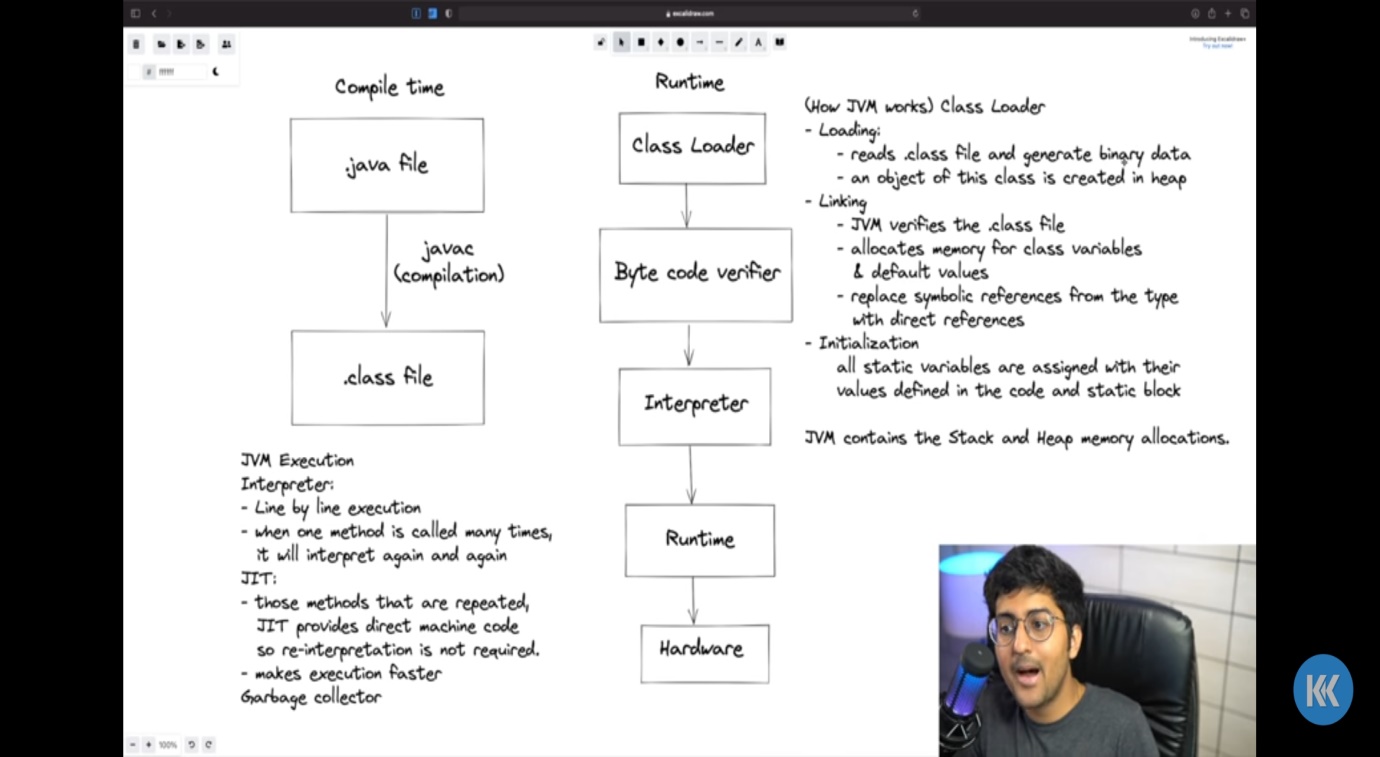
A

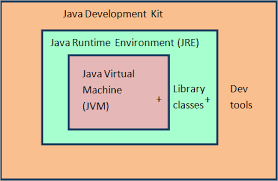
99,2,3,4,5,6,7,8,9,10

B

The values are changed in both A and B





****

**Writing first program in java:**

1)public class get\_input {

2)    public static void main(String[]args){

3)        System.out.println("Hello");

    }

}

**Decoding the java program:**

**Public:** Access specifier or Access modifiers Public means this can be accessed from any where by any class or package

**Static: i)** static does not requires object to run the main function of the main class(with respect to line 2)

**ii)** in gendral with help of static we can use the class with out using the objects

**Void:** it is a return type of the function

**Main:** it is a reserved word, this will be the entry point of the java program

**String [] args:** Command line argument String[] – collection of strings(arrays)

String class that stores java command line arguments. Here, the name of the String array is args (short form for arguments), however it is not necessary to name it args always, we can name it anything of our choice, but most of the programmers prefer to name it args.

**System.out.println();** what ever you type inside the () it will print it, java syntax to print anything

**System:** Class created by the java creators.

**Out:** variable name of printstreamor this is an instance of Printstream

**Println:** The println() method of java printstream class prints an object and terminates the current line by writing line separator string

**More information on line 2:**

We are using public in line 2 because it the main function were the java program start to execute(Entry point of the program is main program) so it has to be public, so that it can be accessible by the other classes of the program.

We are using static in line 2 because , we had discussed that main function is the entry point of the program and main function so we can’t create any objects for that main class in this scenario

**Primitive data types:**

Any data type that you cannot break further is called primitive data type eg char, int , float, double.

**Type casting**

In Java there is a auto upgrading of data types like if you created a float variable and you give integer it automatically takes it as float. It is a implicit operation or internally done

When you want convert float value into integer (This conversion called narrowing) we have to declare them explicitly like the program follow below

int num = (int) (64.34f)

print(num)

it gives output of 64

**interesting example of java in auto promotion**

//let us take a integer variable which has the value of 257

Int num = 257;

We know that the size of byte is 256 when we assign value of num to byte what will happen

Int num =257;

Byte numm = (byte)(257);

Print(numm);

This gives the out put of 1. Because by the maximum value of byte is 256 when it exceeds it will automatically promotes itself and it will start from 1 again, if the value exceeds 256, the given value is modulo to the 256(in the case of byte) and the remainder is the printed answer.

That is the given value%256 = the printed answer

**Loops:**

public class loops {

    public static void main(String[] args) {

      /\*

      // use for loops when you know number of iterations is known

        for (int i=1;i<=5;i++)// intialization, condition, increament or decrement {

            System.out.print(i+" ");

        }

        \*/

        /\*

        // use while loops when you don't know number of times you need to loop or iterate

        int num = 1;//intialization

        while(num<=5){//condition

            System.out.println(num);

            num+=1;//increment or decrement

        }

        \*/

        /\*

        //use do while when atleast one time loop have to ru

        int numm = 1;

        do {

            System.out.println(numm);

            numm+=1;

        }while(numm<=5);

        \*/

    }

}

Important points:

There is no pass by reference in java, there is only pass by value in java.

2 things to be remembered

* For primitives(int, short, char, byte..etc) it just passing the value
* For objects passing value of the reference variable’s value.

**Let** us this example for primitive type:

public class swap {

    public static void main(String[] args) {

        int a,b;

        a=90; //step1

        b=30; //step1

        Swap(a,b);// this will pass the value of a, b to the variable of //function a,b static function swap’s

        System.out.println(a+" "+b);

    }

    static void Swap(int a,int b)// step2… this ’a’ and ‘b’ are local //variables and it will be accessed only within the scope of this function, //when the main function values are passed to this ‘a’ and ‘b’ since integer //is primitive it creates a new object or the refrence variable points towards //the new created object {

        int tem = a;

        a = b;

        b=tem;

    }

}

In common point of view we think that the value of the ‘a’ and ‘b’ variables will change but it will not work like that the **values of the ‘a’ and ‘b’ remains unchanged when we run this we will get the below answer.**

Out put:

90 30

Let us see why this happens, we know that for primitive data types like ‘int’ when we pass the value it creates a copy of the object and it will store in the mentioned reference variable

b

a

Step 1: stack memory heap menory

b of swap

b of main

a of swap

a of main

From the above illustration we can understand for primituve data types, that when we pass the values of the main function to the user created function it creates new object and it stores it on its refrence variable when we make any changes on the newly created refrence variable it only changes that vaibles object and the main object values remains constant.

**Let us take example of object and other complex data types:**

import java.util.\*;

public class swap {

    public static void main(String[] args) {

        int arr [] = {1,2,3,4,5,6,7};

        Swap(arr);// here the values of the reference variable passed so it can be changed in the function

        System.out.println(Arrays.toString(arr));

    }

    static void Swap(int[] akk){

        akk[0]=99;// here the value is changed

    }

**This program gives output of**

**[99, 2, 3, 4, 5, 6, 7]**

So here

arr

When it passes the value of the reference variable to akk

arr

akk

So now when we change the value of the akk the value of arr will automatically changed.

This is the example of passing the value of the reference variable.

**Shadowing:**

Shadowing in Java is the practice of using variables in overlapping scopes with the same name where the variable in low-level scope overrides the variable of high-level scope. Here the variable at high-level scope is shadowed by the low-level scope variable.

**Example :**

public class Shadowing {

    static int i =10; //declared in class and it will be available through out the class

    public static void main(String[] args) {

        System.out.println(i);// it prints 10

        int i = 20;// the higher level gets shadowed and 'i' is assigned to 20

        System.out.println(i);// it prints 20

        func();// this function prints the value of 'i' in the class scope prints 10

    }

    static void func()// this will take the static int i

    {

        System.out.println(i);

    }

}

**Variable Arguments (Varargs) in Java:**

Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments. Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

// Java program to demonstrate varargs

class Test1 {

// A method that takes variable

// number of integer arguments.

static void fun(int... a)

{

System.out.println("Number of arguments: "

+ a.length);

// using for each loop to display contents of a

for (int i : a)

System.out.print(i + " ");

System.out.println();

}

// Driver code

public static void main(String args[])

{

// Calling the varargs method with

// different number of parameters

// one parameter

fun(100);

// four parameters

fun(1, 2, 3, 4);

// no parameter

fun();

}

}

Output:

Number of arguments: 1

100

Number of arguments: 4

1 2 3 4

Number of arguments: 0

Varags can face ambiguity problem.

Varags can be overloaded.

**Method Overloading in Java:**

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

**There are two ways to overload the method in java**

1. By changing number of arguments
2. By changing the data type

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

1. **class** TestOverloading4{
2. **public** **static** **void** main(String[] args){System.out.println("main with String[]");}//print
3. **public** **static** **void** main(String args){System.out.println("main with String");}
4. **public** **static** **void** main(){System.out.println("main without args");}
5. }

### Example of Method Overloading with TypePromotion

**class** OverloadingCalculation1{

**void** sum(**int** a,**long** b){System.out.println(a+b);}

**void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}

**public** **static** **void** main(String args[]){

  OverloadingCalculation1 obj=**new** OverloadingCalculation1();

  obj.sum(20,20);//now second int literal will be promoted to long

  obj.sum(20,20,20);

   }

}

Output:40

60

### Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

1. **class** OverloadingCalculation2{
2. **void** sum(**int** a,**int** b){System.out.println("int arg method invoked");}
3. **void** sum(**long** a,**long** b){System.out.println("long arg method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation2 obj=**new** OverloadingCalculation2();
7. obj.sum(20,20);//now int arg sum() method gets invoked
8. }
9. }

Output:int arg method invoked

### Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

1. **class** OverloadingCalculation3{
2. **void** sum(**int** a,**long** b){System.out.println("a method invoked");}
3. **void** sum(**long** a,**int** b){System.out.println("b method invoked");}
5. **public** **static** **void** main(String args[]){
6. OverloadingCalculation3 obj=**new** OverloadingCalculation3();
7. obj.sum(20,20);//now ambiguity
8. }
9. }

Output:Compile Time Error

**Java array**

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



### Advantages

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### Disadvantages

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically

### Types of Array in java

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

## Single Dimensional Array in Java

**Syntax to Declare an Array in Java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

**Instantiation of an Array in Java**

1. arrayRefVar=**new** datatype[size];

### Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. **class** Testarray{
4. **public** **static** **void** main(String args[]){
5. **int** a[]=**new** **int**[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. **for**(**int** i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}

Output:

10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. **int** a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. **class** Testarray1{
4. **public** **static** **void** main(String args[]){
5. **int** a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. **for**(**int** i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}

Output:

33

3

4

5

## For-each Loop for Java Array

We can also print the Java array using [**for-each loop**](https://www.javatpoint.com/for-each-loop). The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

1. **for**(data\_type variable:array){
2. //body of the loop
3. }

Let us see the example of print the elements of Java array using the for-each loop.

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

Output:

33

3

4

5

## Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. **class** Testarray2{
4. //creating a method which receives an array as a parameter
5. **static** **void** min(**int** arr[]){
6. **int** min=arr[0];
7. **for**(**int** i=1;i<arr.length;i++)
8. **if**(min>arr[i])
9. min=arr[i];
11. System.out.println(min);
12. }
14. **public** **static** **void** main(String args[]){
15. **int** a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}

Output:

3

## Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. **public** **class** TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. **static** **void** printArray(**int** arr[]){
6. **for**(**int** i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
10. **public** **static** **void** main(String args[]){
11. printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method
12. }}

Output:

10

22

44

66

## Returning Array from the Method

We can also return an array from the method in Java.

1. //Java Program to return an array from the method
2. **class** TestReturnArray{
3. //creating method which returns an array
4. **static** **int**[] get(){
5. **return** **new** **int**[]{10,30,50,90,60};
6. }
8. **public** **static** **void** main(String args[]){
9. //calling method which returns an array
10. **int** arr[]=get();
11. //printing the values of an array
12. **for**(**int** i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}

Output:

10

30

50

90

60

## ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. **public** **class** TestArrayException{
4. **public** **static** **void** main(String args[]){
5. **int** arr[]={50,60,70,80};
6. **for**(**int** i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

## Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

**Syntax to Declare Multidimensional Array in Java**

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

**Example to instantiate Multidimensional Array in Java**

1. **int**[][] arr=**new** **int**[3][3];//3 row and 3 column

**Example to initialize Multidimensional Array in Java**

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

### Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

1. //Java Program to illustrate the use of multidimensional array
2. **class** Testarray3{
3. **public** **static** **void** main(String args[]){
4. //declaring and initializing 2D array
5. **int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. **for**(**int** i=0;i<3;i++){
8. **for**(**int** j=0;j<3;j++){
9. System.out.print(arr[i][j]+" ");
10. }
11. System.out.println();
12. }
13. }}

Output:

1 2 3

2 4 5

4 4 5

## Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

1. //Java Program to illustrate the jagged array
2. **class** TestJaggedArray{
3. **public** **static** **void** main(String[] args){
4. //declaring a 2D array with odd columns
5. **int** arr[][] = **new** **int**[3][];
6. arr[0] = **new** **int**[3];
7. arr[1] = **new** **int**[4];
8. arr[2] = **new** **int**[2];
9. //initializing a jagged array
10. **int** count = 0;
11. **for** (**int** i=0; i<arr.length; i++)
12. **for**(**int** j=0; j<arr[i].length; j++)
13. arr[i][j] = count++;
15. //printing the data of a jagged array
16. **for** (**int** i=0; i<arr.length; i++){
17. **for** (**int** j=0; j<arr[i].length; j++){
18. System.out.print(arr[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. }
23. }

Output:

0 1 2

3 4 5 6

7 8

## What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

1. //Java Program to get the class name of array in Java
2. **class** Testarray4{
3. **public** **static** **void** main(String args[]){
4. //declaration and initialization of array
5. **int** arr[]={4,4,5};
6. //getting the class name of Java array
7. Class c=arr.getClass();
8. String name=c.getName();
9. //printing the class name of Java array
10. System.out.println(name);
12. }}

Output:

I

## Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

**Syntax of arraycopy method**

1. **public** **static** **void** arraycopy(
2. Object src, **int** srcPos,Object dest, **int** destPos, **int** length
3. )

### Example of Copying an Array in Java

1. //Java Program to copy a source array into a destination array in Java
2. **class** TestArrayCopyDemo {
3. **public** **static** **void** main(String[] args) {
4. //declaring a source array
5. **char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6. 'i', 'n', 'a', 't', 'e', 'd' };
7. //declaring a destination array
8. **char**[] copyTo = **new** **char**[7];
9. //copying array using System.arraycopy() method
10. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11. //printing the destination array
12. System.out.println(String.valueOf(copyTo));
13. }
14. }

Output:

caffein

## Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

1. //Java Program to clone the array
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. **for**(**int** i:arr)
7. System.out.println(i);
9. System.out.println("Printing clone of the array:");
10. **int** carr[]=arr.clone();
11. **for**(**int** i:carr)
12. System.out.println(i);
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
17. }}

Output:

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

false

## Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

1. //Java Program to demonstrate the addition of two matrices in Java
2. **class** Testarray5{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,3,4},{3,4,5}};
6. **int** b[][]={{1,3,4},{3,4,5}};
8. //creating another matrix to store the sum of two matrices
9. **int** c[][]=**new** **int**[2][3];
11. //adding and printing addition of 2 matrices
12. **for**(**int** i=0;i<2;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
20. }}

Output:

2 6 8

6 8 10

## Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

1. //Java Program to multiply two matrices
2. **public** **class** MatrixMultiplicationExample{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. **int** b[][]={{1,1,1},{2,2,2},{3,3,3}};
8. //creating another matrix to store the multiplication of two matrices
9. **int** c[][]=**new** **int**[3][3];  //3 rows and 3 columns
11. //multiplying and printing multiplication of 2 matrices
12. **for**(**int** i=0;i<3;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=0;
15. **for**(**int** k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]\*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}

Output:

6 6 6

12 12 12

18 18 18

int [] arr = new int[5];

In Java there is a shortcut LHS is done in the Compile time, RHS is done in runtime(Dynamic memory allocation)

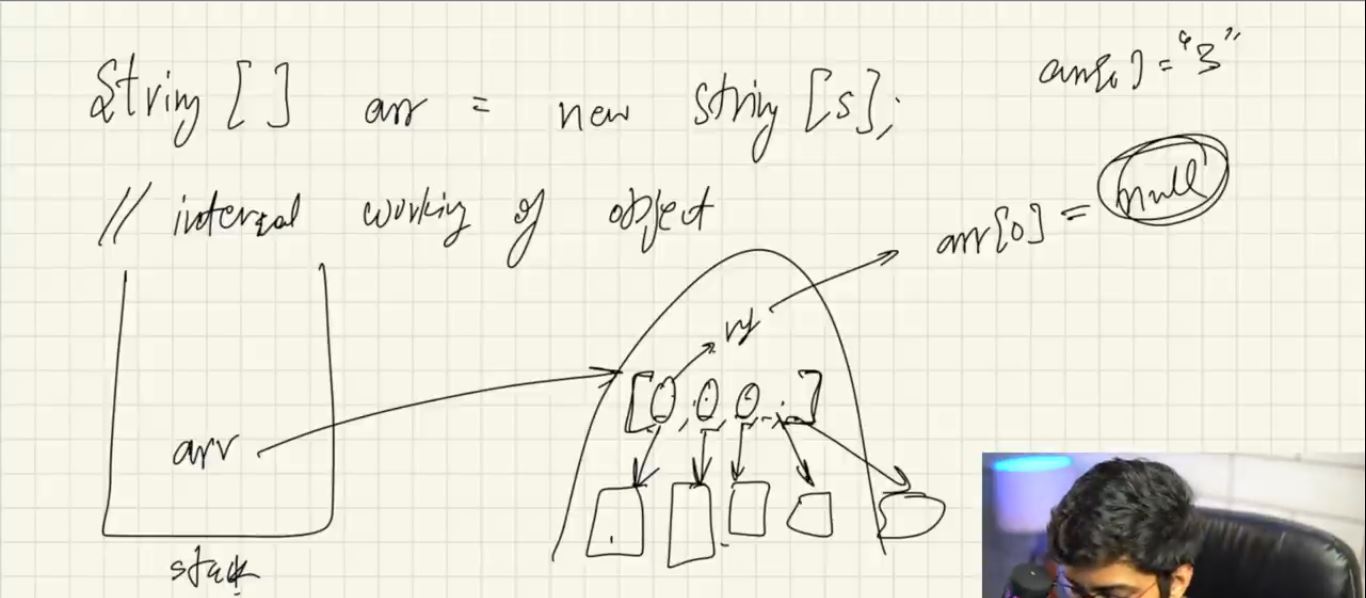
Dynamic memory allocation, it is basically at the run time memory is allocated.

1)Array objects are in Heap memory

2) Heap objects are not continuous.

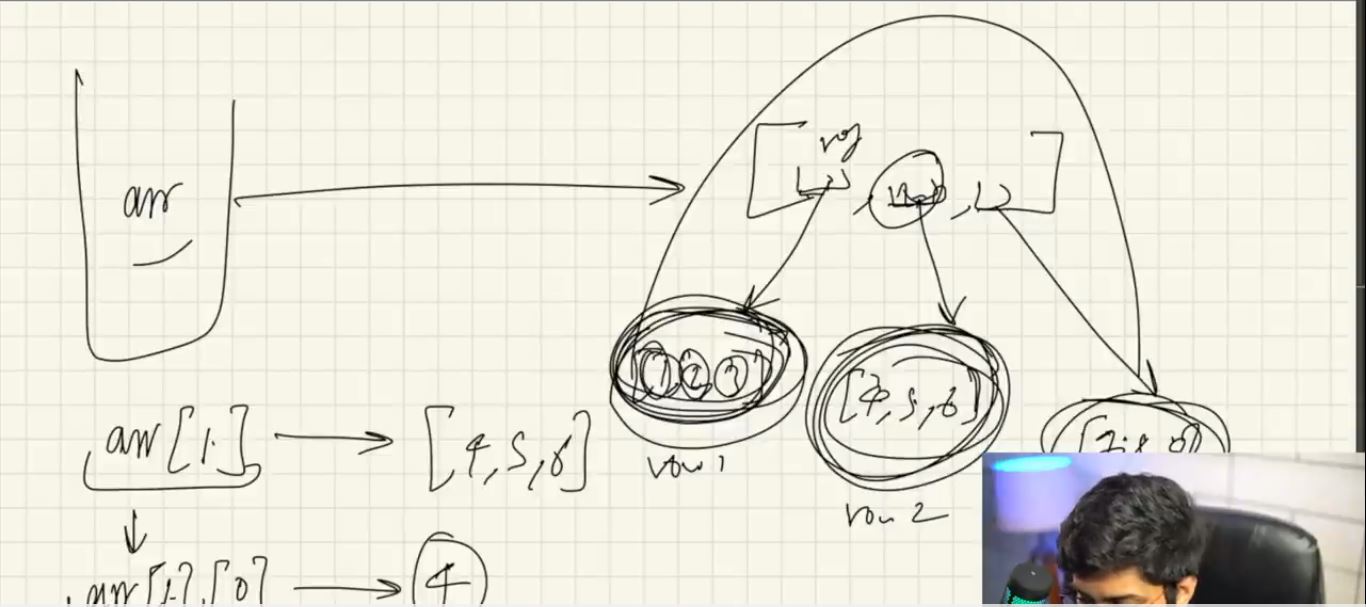
3) Array objects may not be continuous. It depends on JVM, because memory allocation is done by JVM.

4) Primitive data type is stored in stack memory.

5) Non-primitive data types are stored in heap memory.

In java creating 2d arrays follows this syntax

Int[][] arr2d = new int[3][];

Here the first box mentions rows and the second box mentions the column, we have to definitely give the value of row but the value of column is not mandatory, because the column length can be variable size. The Size of the individual rows doesn’t matter.

Array list :

The best thing about array list is we don’t need to mention the size of the array.

Syntax of the array list

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

Here list is reference variable