Avishek (8210928619)

1.

Core Java

* Language Fundamentals.

1.

5-Apr-23

* Language Fundamentals.
1) Identifiers
2) Reserved words
3) Data Types
4) Literals
5) Arrays
6) Types of variables
7) Var-Arg methods
8) main method
9) Command line arguments
10) Java coding standards.

Identifiers

→ A name in java program is called as identifier.

```
class Test {
    public static void main(String[] args)
    { int sc = 10 ; }
}
```

* Rules :-
1) The only allowed characters are a to z, A to Z, 0 to 9, $, _.
   eg :- total_num ✓ , total #X
2) Identifers can't start with digits
   total123 ✓ 123total X

→3. Java identifers are case sensitive.

```
class Test {
    int number = 10;
    int Number = 20;
    int NUMBER = 30;
}
```

} we can differentiate wrt case.

4) There is no length limit but it is not recommended to take too lengthy identifier.

5) We can't use reserved words as identifiers.

eg `int x = 10;` ✓

`int if = 20;` ✗

6) All pre-defined java class name and interface name we can use as identifiers but not recommended.

```
class Test {
    p s v main(String[] args) {
        int String = 888;
        Sopln(String); #888
```

✓

```
        int Runnable = 999;
        Sopln(Runnable);
```

eg. ✓ total_number, ✗ total# ✗ 123total

✓ total123 ✓ ca$h ✓ _$_$_$_$

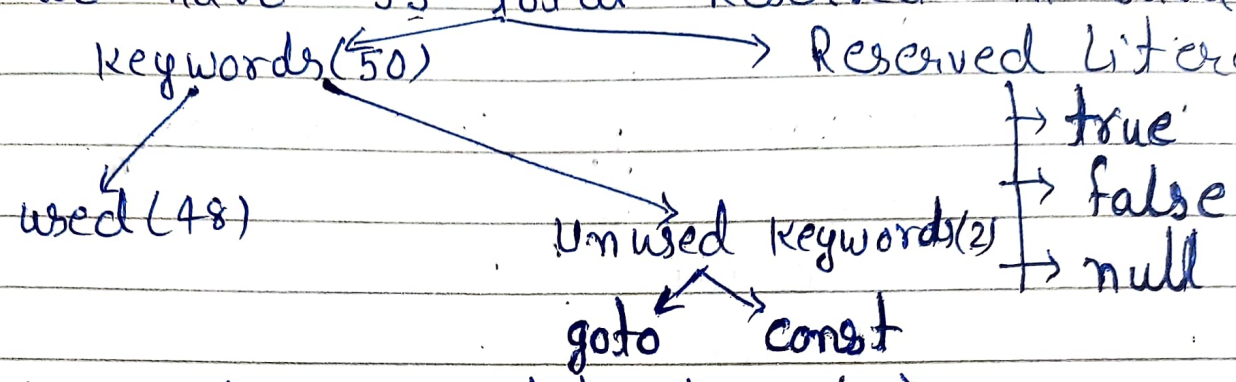
✗ all@hands ✓ Java2share ✓ Integer

✓ Int ✗ int

Reserved words

5-Apr-23

→ We have 53 total Reserved in Java



* Keywords for data-type. (8)
  byte short int long
  float double boolean char } 8
* For flow ~~key~~ control. (11)
  if else switch case
  default while do for } 11
  break continue return
* For modifiers (12)
  public private protected static
  final abstract synchronized native } 12
  strictfp(1.2v) transient volatile default
* For Exception handeling (6)
  try catch finally throw } 6
  throws assert(1.4v)
* class related (6)
  class interface extends implements
  package import } → 6
* object related (4)
  new instanceof super this } → 4

* return type (1)
  void } → 1.
→ In java return type is mandatory.
* Un-Used keywords (2)
  goto const } 2
* Reserved literls.
  true false null } 3
  values for bool data-type — default value for object reference
* Enum keyword
  enum
  ↳ to define a group of named constant.
  enum month {
  JAN, FEB, ... DEC }
* Conclusions :-
→ All 53 words contains only lower case symbols
→ In Java we have only new keyword, no delete keyword bcz it is Garbage collector job.
→ strictfp, assert & enum are new keywords.
→ strictfp ~~StrictFp~~
→ instanceof ~~instanceOf~~
→ Synchronized ~~Synchronize~~
→ extends ~~extend~~
→ implements ~~implement~~
→ import ~~imports~~

#2. Data Types

5-Apr-23

Data Types

→ In Java, every variable and expression have type. Each and every data type is clearly defined.

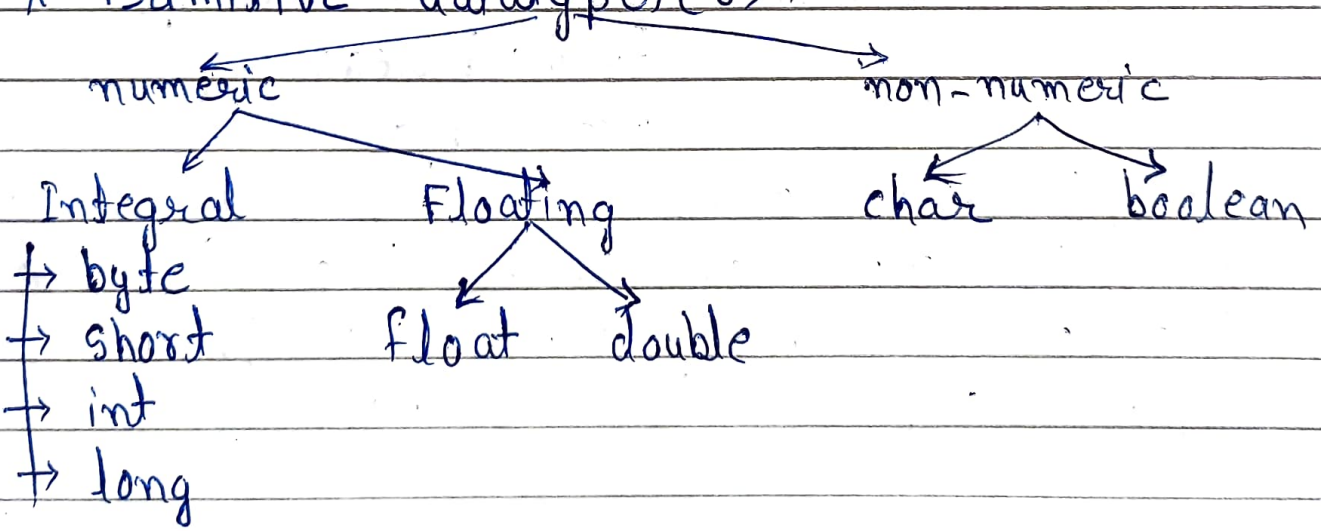
→ Every assignment is checked by compiler for type compatibility.

→ Java is **Strongly typed.**

→ Java is not considered as pure object oriented Programming language because several oops feature is not supported by java (like operator overloading, multiple inheritence etc.)

We are depending on premitive data types which are non-objects

* Primitive datatypes (8)



→ Except boolean & char remaining data types are considered as signed data types (+ve & -ve)

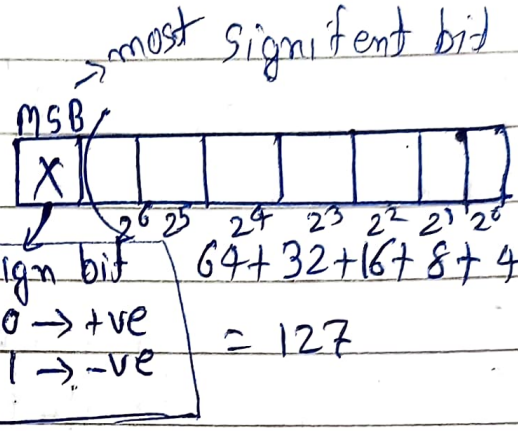
* byte :-

Size - 1 byte (8 bits)

Max_value : + 127

MIN-VALUE : - 128

Rang : -128 to 127



→ +ve number directly represented in memory where -ve number represented in 2's compliment

eg: ✓ byte b = 10 ✓ byte b = 127

✗ byte b = 128 : CE : Possible loss of Precision
found : int
required : byte.

✗ byte b = 10.5; CE : PLP
found : double
req : byte

✗ byte b = true; CE : incompatible type
found : boolean
req : byte

→ byte is best choice if we want to handle data in terms of steams either from the file or from the network (file supported or network supported form is byte.

#3. Data Types

* short :-
  Size → 2 byte (16 bits)
  Range → -2^{15} to $2^{15}-1$
  [-32768 to 32767]
→ Most rarely used.
→ Most popular around 1995, 16 bits processors.

* int :-
  Size :- 4 bytes (32 bits)
  Range : -2^{31} to $2^{31}-1$
  [-2147483648 to 2147483647]

* long :-
  Size :- 8 bytes (64 bits)
  Range : -2^{63} to $2^{63}-1$

* Floating-point data types :-

float	double
1) 5 to 6 decimal place accuracy.	1) 14 to 15 decimal place accuracy.
2) Single precision	2) double precision
3) Size : 4 bytes	3) Size : 8 bytes
4) Range : -3.4e38 to 3.4e38	4) Range -1.7e308 to 1.7e308

* Boolean data types :-
  Size : NA [VM dependent]
  Range : NA [but allowed values : true/false]

✓ boolean b = true;

✗ boolean b = 0; CE: incompatible types
found: int
req : boolean

✗ boolean b = True; CE: cannot find symbol
Symbol: variable True
loc : class Test.

⇒ int x = 0;
if (x) { sopln ("Hello") } CE: incompatible types
else { sopln ("Hi"); found: int
⇒ while(1) { sopln("Hello"); } req: boolean

* char data types :-
size : 2 bytes
→ old languages (C/C++) are ASCII based (allowed char. ≤256) for them 1 byte is ~~enough~~ enough but Java is Unicode based and number of unicode char are > 256 & ≤ 65536 to represent these many char 8 bits may not enough so we should go for 16 bits. hence size of char in Java is 2 bytes
→ Range : 0 to 65535

* Summary

data type	Size	Range	wrapper class	Default value
byte	1 byte	-2^7 to $(+2^7-1)$	Byte	0
short	2 bytes	-2^{15} to $(2^{15}-1)$	Short	0
int	4 bytes	-2^{31} to $(2^{31}-1)$	Integer	0
long	8 bytes	-2^{63} to $(2^{63}-1)$	Long	0
float	4 bytes	-3.4e38-3.4e38	Float	0.0
double	8 bytes	-1.7e308-1.7e308	Double	0.0
boolean	NA	NA (true/false)	Boolean	false
char	2 bytes	0 to 65535	Character	0 (space)

→ null is default value for object refernce and we can't not apply for premitive.

#1. Literals

05-Apr-23

Literals

→ A constant value which can be assigned to a variable is called literal.

> int x = 10 ;
>
> ↳ constant value / literal

① decimal literals : (base - 10) ; (0 to 9)

int x = 10 ;
↳ decimal form

② octal form : (base - 8) (0 to 7)

int x = 010 ;

③ Hexa decimal form : (base - 16)

int x = 0X 10 ;
int y = 0x 10 ;

0 to 9, a to f
both case chose A to F

→ These are the only way for integral literal

✓ int x = 10 ;
✗ int x = 0786 ; CE:
✓ int x = 0777 ;
✗ int x = 0x Beer ;
✓ int x = 0XFace ;

```
class Test {
    p s v main (String [] args) {
        int x = 10 ;
        int y = 010 ;
        int z = 0x10 ;
        Sop ln (x + "--" + y + "--" + z)
    }
}
```

→ 0X8⁰ + 1X8¹ = 8
(10)₁₀ = (2)₁₀
0X16⁰ + 1X16¹ = 16

provide 10 --- 8 --- 16

→ JVM output put only in decimal.

→ bydefault every integral literals are int type to make long we need to add suffix 'l', 'L'

✓ int x = 10; ✓ long l = 10L;
✗ int x = 10L; long l = 10;

byte b = 10; ✓ byte b = 127;
↳ within limit of byte.
✗ byte b = 128; → CE;
↳ out of range so CE;

✓ short s = 32767;
✗ short s = 32768; CE:

✗ float f = 123.456; CE:
✓ float f = 123.456F; → need to add F
✓ double d = 123.456;
↳ by default double
✓ double d = 123.456D;
✗ float f = 123.456d; CE:

✓ double d = 123.456; → it is decimal literal but not octal literal.
✓ double d = 0123.456;
✗ double d = 0X123.458; CE: malformed floating point literal.

✗ double d = 0786; → CE: integral no. too large.
✓ double d = 0XFace;
→ If needed for literals watch it again.

Skipped note making.

#5. Literals

06/Apr/23

* Boolean literals

→ The only allowed value for boolean is true or false.

✓ boolean = true; ✗ boolean b = 0;

✗ boolean = True; ✗ boolean b = "True";

* Char literals

✓ char ch = 'a';

✗ char ch = a; CE: cannot find symbol

✗ char ch = "a"; CE: " " "

char ch = 'ab'; CE: Unclosed char literal

CE: " " "

CE: not java statement.

→ We can specify char literal as integral literal (Unicode of char).

✓ char ch = 0xFace;

✓ char ch = 0777;

✓ char ch = 65535;

✗ char ch = 65536; CE: PLP

16|97
a=1

* www.unicode.org

'\uXXXX'

↳ 4 digit hexa-decimal number

char ch = '\u0061'; // a

char ch = '\n';

→ Every escape char. is valid char literal.

memory rep. is diff.

byte → short → int → long → float → double
1b 2b 4b 8b 4 byte 8
char → Integral floating-point

Escape char	description
\n	New line
\t	Horizontal Tab
\r	Carriage return
\b	Back space
\f	form feed
\'	Single quote
\"	double quote
\\	Back slash

✗ char ch = 65536 ✗ char ch = 0xBeer;
✗ char ch = \uface; ✓ char ch = '\ubeef';
✗ char ch = '\m'; ✗ char ch = '\iface

* String literals
String s = "Avishek";

→ 1.7 v enhancement

① Binary literals
int x = 0B1111; # 15

② Usage of '_' Symbol in numeric literals
double d = 123456.789;
double d = 1_23_456.7_8_9;
→ readibility of code improved.
✓ double d = 1___23__4_5__6.7_8_9;
✗ double d = _1-23_456.7_8_9;
✗ double d = 1-23_456_.7_8_9;
double d = 1-23_456.789_;

#6. Arrays

06/Apr/23

1) Introduction
2) Array declaration
3) Array creation
4) Array initialization
5) Array declaration, creation & initialization in a single line.
6) length .Vs length()
7) Anonymous Arrays
8) Array element assignments
9) Array variable assignments.

* Introduction

(i) Arrays are fixed in size.

→ An array is an indexed collection of Homogeneous data elements.

→ Adv:- of array is we can represent huge no. of values by using single variable.

→ Dis. Adv :- fixed in size, we should know size in advance, which may not possible always.

* Declaration

→ 1-D array

int[] x; → R → bcz name is clearly seperated from type.

int []x;

int x[];

→ At time of declaration we can't specify size. int [6] x; ✗

→ 2-D array :-

✓ int[][] x; int[] []x;
✓ int []x; int[] x[];
✓ int x[][]; int []x[];

✓ int[] a, b; a→1, b→1
✓ int [] a[], b; a→2 b→1
✓ int [] a[], b[]; a→2 b→2
✓ int [] []a, b; a→2 b→2
✓ int [] []a, b[]; a→2 b→3
✗ int [] []a, []b; → CE

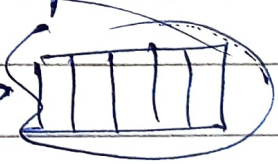
→ 3-D array :-

int[][][] a; int[] [][]a;
int [][][] a; int [] a[][];
int a[][][]; int [] []a[];
int[][] []a; int [][] a[];
int [] [][]a[]; int []a[][];

* Creation :-

→ Every array in java is an object. hence we create array by using new operator.

eg. int[] arr = new int[5];
Sop(arr.getClass().getName()); EI a→



Array type	corresponding class name
int []	[I
int [][]	[[I
double []	[D
short []	[S
byte []	[B
boolean []	[Z

✗ int [] x = new int []; ✗

✓ int [] x = new int [6]; ↪ compulsory to give size.

✓ int [] x = new int [0];

✓ int [] x = new int [-3]; RE: -ve Array size Exception

but no CE

int [] x = new int ['a'];

int [] x = new int [b];
byte b = 20;

byte → short → int
char → int

int [] x = new int [10L]; CE: PLP

may get CE (due to memory)

✓ int [] x = new int [2147483647];

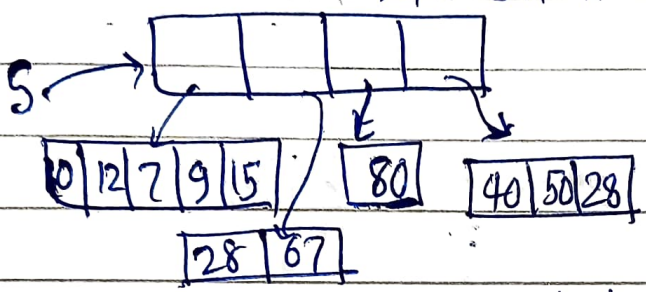
✗ int [] x = new int [2147483648];

#7. Arrays

06|Apr|23

* 2-D array creation

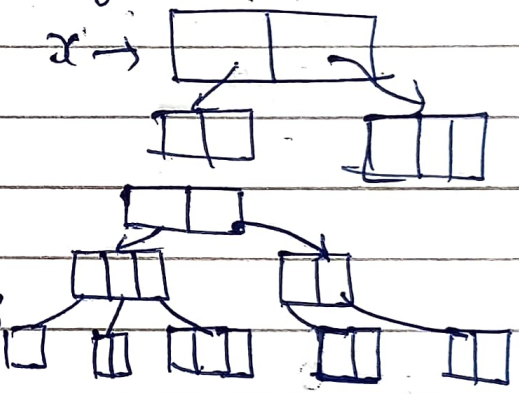
Arrays of arrays in JAVA



S1	10	12	07	09	15
S2	28	67			
S3	80				
S4	40	50	28		

→ Adv. → memory utilization get improved.

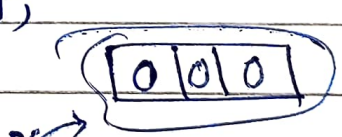
→
```
int[][]x= new int[2][];
   x[0]  = new int[2];
   x[1]  = new int[3];
```




```
int[][][]x= new [2][][];
   x[0] = new int[3][];
   x[0][0] = new int[1];
   x[0][1] = new int[2];
   x[0][2] = new int[3];
   x[1] =  new int[2][2];
```

* Initialization

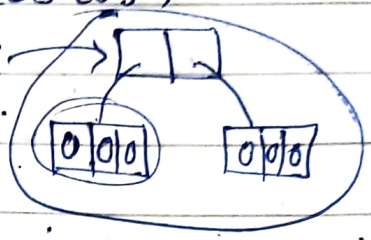
```
int[] x = new int[3];
```



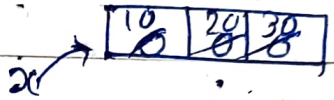

```
Sopln(x); [I@3e25a5
Sopln(x[0]); 0
```

→ Once we create an array every element by deafult initialize with default values.

```
int [][] x = new int[2][3];
Sop(x);  [[I@3e25a5
Sop(x[0]); [I@1982fe
Sop(x[0][0]); 0
```




```
int [] x = new int[3];
int[0] = 10;
int[1] = 20;  int[2] = 30;
int[4] = 70;  RE: out of Bound Exc.
int[-4] = 80;  RE: "
int[2.5] = 90;  CE: PLP (Possible loss of Precision)
```



#8. Arrays

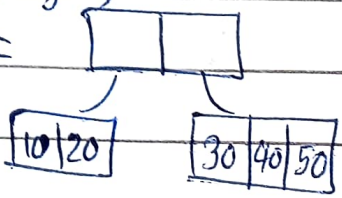
07/Apr/23

* Array Declaration, Creation & Initialization in a single line

```
int [] x;
x = new int[3];
x[0] = 10;
x[1] = 20;
x[2] = 30;
```

```
int [] x = {10,20,30};
char [] ch = {'a','e','i','o','u'};
```

```
int [][] x = {{10,20},{30,40,50}};
```



✓ int [] x = {10, 20, 30};

int [] x;

x = {10, 20, 30}; : CE: illegal start of Exp

* length vs length()

length:

int [] x = new int [6];

Sopln (x.length()); → CE:

Sopln (x.length): 6

→ length is a final variable applicable for arrays

* length()

String s = "Avishek";

Sopln (s.length); CE:

Sopln (s.length()); 7

→ length() is final method applicable for string objects.

String s = {"A", "AA", "AAA"};

✓ Sopln (s.length); 3 ✗ Sopln (s[0].length); CE

✗ Sopln (s.length()); CE ✓ Sopln (s[0].length()); 1

int [][] x = new int [6][3];

Sopln (x.length); 6 ✓

Sopln (x[0].length); 3 ✓

* Anonymous Arrays

→ Nameless array is called anonymous array

→ the main purpose of these array is just for instant use (one time).

```
class Test {
    p s v main(String[] args){
        sum(new int[] {10,20,30,40});
    }
    p s v sum(int[] x){
        int total =0;
        for(int x1 : x){
            total = total + x1;
        }
        Sopln("The Sum: " + ~~sum~~total);
    }
}
```

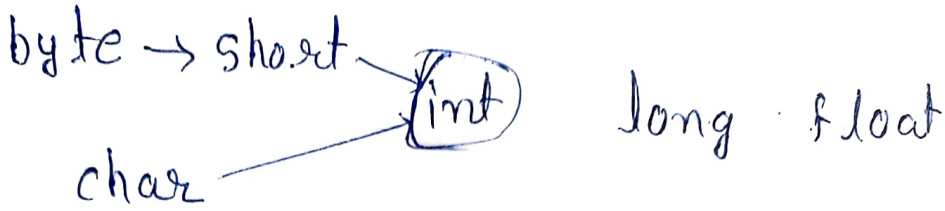
⇒ new int[] {10,20,30,40};

✗ new int[3] {10,20,30};

new int[][] {{10,20}, {30,40,50}};

→ Based on requirement we can give the name for anonymous array.

int[] x = new int[] {10,20,30};



* Array element assignments

⇒ Case 1 :- As array element we can assign any type which can be implicitly promoted to declared type.

```
int[] x = new int[5];
x[0] = 10;   x[1] = 'a';
byte b = 20;   x[2] = b;
short s = 30;   x[3] = s;
x[4] = 10l;  CE: PLP
```

⇒ Case 2 :- In case of object type arrays as array elements we can provide either declared types objects or it's child class objects.

eg :-
```
Object[] a = new Object[10];
✓ a[0] = new Object();
✓ a[1] = new String("durga");
✓ a[2] = new Integer(10);

Number[] n = new Number[10];
✓ n[0] = new Integer(10);
✓ n[1] = new Double(10.5);
X n[2] = new String("durga"); CE:
```

⇒ Case 3 :- For interface type arrays as array elements, it's implementation class objects are allowed.

```
Runnable[] r = new Runnable[10];
✓ r[0] = new Thread();
X r[1] = new String("durga"); CE:
```

#9. Arrays

07/Apr/23

Array typ.	Allowed element type.
primitive Arrays	Any type which implicitly promoted to declared type
Object type Arrays	Either declared or its child class obj.
Abstract class type	its child class obj.
interface type array	its imp. class obj.

* Array Variable assignment.

⇒ Case 1: Element level promotions are not applicable at array level.

1. char ele. → int type
   char[] ✗⇒ int array

```
int[] x = {10, 20, 30, 40};
char[] ch = {'a', 'b', 'c', 'd'};
✓ int[] b = x;
✗ int[] c = ch; CE:
```

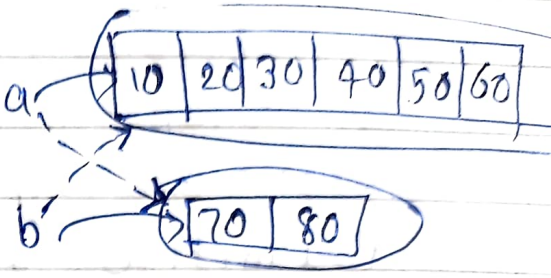
→ Obj. type → child to parent ⇒ possible.

```
String[] s = {"A", "B"};
Object[] o = s;
```

⇒ Case 2:- Whenever we are assigning one array to another array internal elements won't be copied just reference variable will be re-assigned

```
int[]a = {10,20,30,40,50,60};
int[] b = {70,80};
```

1) `a = b;`
2) `b = a;`

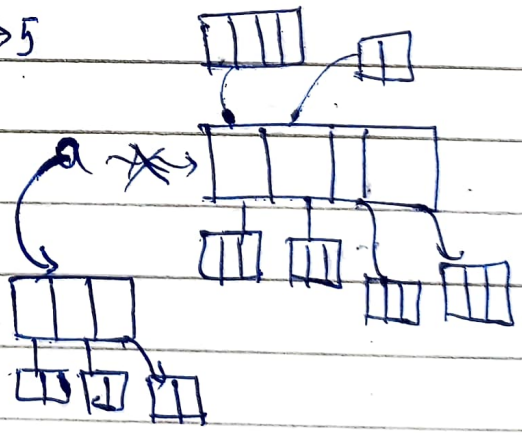


⇒ **Case-3 :-** Whenever we are assigning one array to another array ~~internal elements~~ the dimensions must be matched.

```
int [][] a = new int [3][0];
✗ a[0] = new int [4][3]; CE:
✗ a[0] = 10; C.E
✓ a[0] = new int [2];
```

Note :- whenever we are assigning one array to another [] both dim. & type must be matched but sizes not required to match.

```
int [] a = new int [4][3]; → 5
a[0] = new int [4]; → 1
a[1] = new int [2]; → 1
a = new int [3][2]; → 4
```



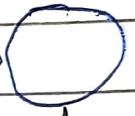
→ Total obj. created → 11
→ Total obj. of GC? → 7

#10. Types of Variables

07/Apr/23

→ Based on type of value represented by a variable all variables are divided into two types.

1) Primitive variables
   - It can be used to represent primitive value.
   int x = 10;

2) Reference variables
   It can be used to refer object.
   Student s = new Student(); s → 

* Division -2 based on Position of declaration & Behaviour

1) Instance variable
2) Static variable
3) Local variable

1) Instance variable
   If the value of variable is varied object to object. such type of variables are called instance variable.
   → For every object seperate copy of instance variables will be created. (school stud. example)
   → Declared within the class but outside of any method, block or constructor.
   → It get created at the time of object creation.
   → Scope is of contained object.
   → It is stored in heap memory as part of object.

→ We can't access it from static area but we can access using object reference.
→ We can access it directly from instance area.

```
class Test {
    int x = 10;
    P S V main(String[] args) {
      { Sopln(x); CE:
        Test t = new Test();
        Sopln(t.x); 10 ✓        }
    public void m1() {
        Sopln(x); ✓ }
}
```

→ JVM provide default values & we not require to perform initilization explicitly.
→ It is also known as object level var. or attrib

* Static variables

→ If the value of variable is not varied object to object then we have to declare such type of variables at class level by using static modifier.
→ Single copy will be created and shared by every object of class.
→ Static var. should be declared within the class directly but out side of any method block or constructor.

→ Also known as class level var. (or) Fields.
→ Scope of static var. = Scope of class.
→ JVM will provide default value.

Java Test ↵

1) Start JVM
2) create & start main Thread
3) Locate Test.class file.
4) Load Test.class ● → static var. creation
5) Execute main() method
6) Unload Test.com ● → static var. destruction.
7) Terminate main Thread
8) Shut down JVM

→ Static var. will be stored in method area.
→ We can access static var. either by object reference or by class name (Recommended)
→ Within the same class we can access directly.

```
class Test {
    static int x = 10;
    P S V main (S [] args) {
        Test t = new Test();
        Sopln (t.x);
        Sopln (Test.x);
        Sopln (x);
    }
}
```

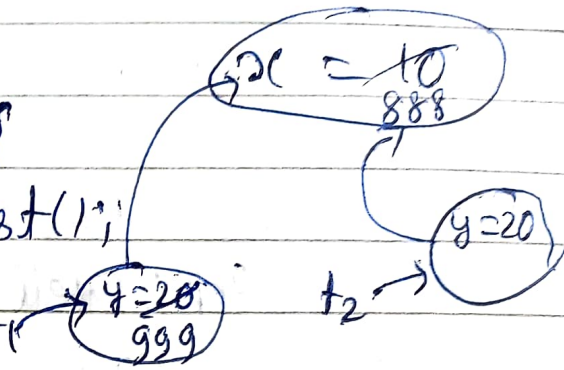
→ Can access from both instance & static are

```
public void m1() {
    Sopln (x)
}
```

#11. Types of Variables

```
class Test {
    static int x = 10;
        int y = 20;
    p s v m(S[] args) {
        Test t1 = new Test();
        t1.x = 888;
        t1.y = 999;
        Test t2 = new Test();
        Sopln(t2.x + "---" + t2.y);
    }
}
```

888 --- 20



* Local variables

→ Variables declared inside a method (or) block or constructor such type of variable are called local (or) Temporary (or) Stack (or) automatic variables. ↳ stored in stack.

→ Local var. will be created while executing the block in which it declared, once block execution complete it get distroyed.

→ Scope of local = Scope of block.

```
class Test {
    P S V main(S[] args) {
        int i=0;
        for(int J=0; J<3; J++) {
            i = i+J;
        }
    } Sopln(i + "---" + J); → CE:
```

→ for local var. JVM won't provide default val.

```
class Test{
  psvm(S[]args){
    int x;
    Sop(x);}
}
```
CE:

```
class Test{
  psvm(S[]args){
    int x;
    Sop("Hello");}
}
```
o/p: Hello

→ We need to perform initilization before using the variable.

```
class Test{
  psvm(S[]args){
    int x;
    if(args.length>0){
      x=10;}
    Sop(x);}
}
```
CE:

```
class Test{
  psvm(S[]args){
    int x;
    if(args.length>0){
      x=10;}
    else{x=20;}
    Sop(x);}
}
```

Note:- It is not recommended to perform initilization for local var. inside logical block.

→ Highly recommended to perform init. at the time of declaration.

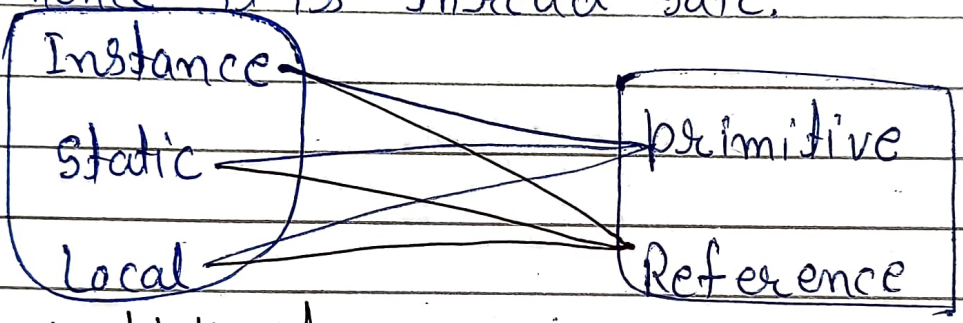
→ The only applicable modifier for local var is **final**. other will give **CE**.

→ If we are not declaring with any modifier the by default it is "default". but it is only applicable for instance & static var. not for local var.

Conclusion :~

1) For instance & static var. JVM will provid default value. but for local JVM won't provide
2) Instance and static var. can be accessed by multiple thread simultaneously & heance these are not thread safe. but local var. for every thread separte copy will be creat & hence it is thread safe.

3)



* Uninitialized array

```
Class Test {
    int [] x;
    p s v m(S [] args) {
        Test t = new Test();
        Sopln(t.x) ; null
        Sopln (t.x[0]); }
}
```

RE: NPE

II / Static

Instance level

1) int [] x;
Sopln(obj.x); null
Sopln(obj.x[0]);
RE:

III Local level

1) int[]x;
Sopln(x); } CE:
Sopln(x[0]);

2) int[]x = new int[3];
Sopln(x); [I@...
Sopln(x[0]; 0

② int [] x = new int [3];
Sopln(obj.x); [I@e23
Sopln(obj.x[0]); 0

→ Array property will dominate.

#12. Var-arg method

07/Apr/23

* Variable number of argument method
→ Declaration
m1 (int... x) → int[] x
→ we can call this method by passing any value
✓ m1(); ✓ m1(10); ✓ m1(10,20,30);
→ Internally var-arg parameter converted into 1-D array.

```
Class Test {
    p s v m (S [] args) {
        sum(10,20);  30
        sum(30,10,20,40);  100
    p  s  v  sum (int... x) {
        int total = 0;
        for (int x1 : x) {
            total += x1; }
        Sop(total);
    }
```

C-1
m1 (int... x); ✓	m1(int ...x); ✗
m1 (int ...x); ✓	m1(int x...); ✗
m1 (int x...); ✗	

C-2
✓ m1 (int x, int... y)
✓ m1 (String s, double... y)

C-3 m1 (double... d, String s) ✗
→ ~~Only~~ In mix parameter, var-arg para. should be last one.

→ We can take only one var-arg parameter.
m1(int... x, double... d); X

→ We can't have two method with same Signature.

```
class Test{
    p s v m1(int... x){
        Sopln("int...");}
    p s v m1(int[] x){
        Sopln("int[]");}
```
} CE:

→ In general, var-arg method get least priority

* Equivalence b/w var-arg para. & 1-D array

1) m1(int[] x) ≡ m1(int... x)
main(String[] args) ≡ main(String... args)

→ reverse is not possible.

X m1(int... x ≡ m1(int[] x)

Note:- m1(int... x) ≡ int[] x 1-D
m1(int[]... x) ≡ int[][] x 2-D

#13. main() method

08/Apr/23

→ Code will compile without main().
→ Runtime, it will be checked by JVM.

public static void main(String[] args)

- public: To call by JVM from anywhere
- static: without existing object also JVM has to call this method.
- void: main() method won't return anything to JVM
- main: This is the name which is configured inside JVM
- String[] args: Command-line arguments

→ Above syntax is very strict.
→ We can change order of modifiers.
⇒ 1) ~~static~~ public
2) ~~main~~(String[] args) ; ~~main~~(String args[]);
3) main(String[] durga);
4) main(String... args);
→ We can declare main method with following modifiers.

static final synchronized strictfp public
void main(String ... durga) {
    sopln("valid main");

Case I - Overloading of main method is possible but JVM will call String[] method only, other overloaded method need to call explicitly:

```
class Test {
    p s v main(String[] args) {
        Sopln("String[]");
    }
    p s v main(int[] args) {
        Sopln("int[]");
    }
}
```

overloaded method

o/p : String[]

Case II - Inheritence is applicable for main()
→ If child class don't have main() then parent class main() will be executed.

P.Java

```
class P {
    p s v main(String[] args) {
        Sopln("parent main");
    }
}
class C extends P { }
```

Java P↵ o/p; Parent main
Java C↵ o/p; parent main

Case III - It seems method overriding is applicable for main() but it is not overriding but it is ~~the~~ method hiding.

#14

```
class P {
    p s v main(String [] args) {
        Sopln("Parent main"); }
}
class C extends P {
    p s v main(String [] args) {
        Sopln("child main"); }
}
```

If is method hiding but not overriding

Java P ↵ O/P: Parent main
Java C ↵ O/P: child main

#14. main() method.

* 1.7 V enhancenment 08/Apr/23

→ If no main() method in class we get more meaningful error.

→ main() method is mandatory to start program execution.

```
class Test {
    static {
        Sopln("Static Block"); }
    p s v main(String [] args) {
        Sopln("Main method"); }
}
```

#15 Command Line Arguments

08/Apr/23

→ The argument which are passing from command prompt are called command-line arguments.

→ JVM creates array with all the argument.

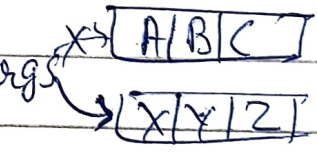
> Java Test A B C ↵

args[0] args[1] args[2]

→ The main objective of command line arg. is we can customized of beahiviour of main method.

```
class Test {
    P S V main(String [] args) {
        int n = Integer.parseInt(args[0]);
        Sopln("The Square of"+n+"is"+n*n); }
}
```

```
class Test {
    p s v main(String [] args) {
        String [] argh = {"x","y","z"};
        args = argh;  → ①
        for (String s : args) {
            Sopln(s);
        }
}
```



Java Test : A B C ↵ # X Y Z

→ Space is separater b/w cmd argu. use " "

22 min

#16. Java Coding Standards

```
class A{
    public int m1(int x,int y){ return x+y; }
}
```
⇒ Accepted std.

```
package com.durgasoft.scjp;
public class Calculator {
    public static int add(int number1,int number2)
    { return number1+number2; }
}
```

Hi-tech city standard

⇒ It is highly recommended to follow Java coding standard.

→ Name should reflect the purpose of the component (~~functionality~~) (functionality).

→ Adv. is readability & maintainiblity improved

* Coading Standard for Classes :-

→ class name are Noun.

→ Should start with Upper Case & follow CamelCase

eg:- String, Account, Dog, StrBuffer.

* for Interface

→ Interface names are Adjective.

→ Should start with upper Case & follow camelcase.

eg:- Runnable, Comparable etc.

* for methods
→ methods names are verbs. OR verb-noun com
→ start with lower case follow camelcase.
eg :- print(), sleep(), eat(), getName() etc.
* for variables
→ variables names are Noun.
→ should start with lower case & follow camelcase.
eg :- name, age, salary etc.
* for constants
→ constant names are noun.
→ Should contain only Upper letter seperated by
eg :- MAX_VALUE, MIN_PRIORITY etc.
→ Declared with public static & final modifiers.

* Java Bean coading standard
→ A JavaBean is a Simple Java class with private properties & public getter & setter methods.

```
public class StudentBean {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

* for setter
→ It should be public method.
→ return should be void.
→ method name should be prefixed with "set".
→ It should take some argument.

* for getter
→ It should be public method.
→ return type should not be void.
→ method name should be prefixed with "get".
→ It should not take any argument.

Note:- for boolean ⇒ "get" (or) "is" → recommended.

* for Listeners :-
* To register listener
→ method name should be prefixed with "add"
eg. public void addMyActionListener(
MyActionListener l)
X public void registerMyActionlist---
public void addMyActionListener(ActionListener l)

* To unregister listener
→ method name should be prefixed with 'remove"
eg public void removeMyActionlist(MyActionlist l)
X public void unRegisterMyActionlist(---
X " " ~~rem~~ delete " " "

40:00 min

#17. Operators & Assignments

08/Apr/23

Kathy Sierra book

1. Increment & Decrement operators
2. Arithmetic operators
3. String concatenation operators
4. Relational operators
5. equality operators
6. Instanceof operators
7. bitwise operators
8. Short circuit operators
9. type cast operator
10. assignment operators
11. Conditional operators
12. new operator
13. [] operator
14. operator precendence
15. Evaluation order of operands
16. new Vs newInstance()
17. instance Vs isInstance()
18. classNotFoundException Vs NoClassDefFoundError.

* Increment & Decrement operator.

Increment: pre, post
- pre: `y = ++y`
- post: `y = x++`

Decrement: pre, post
- pre: `y = --x`
- post: `y = x--`

Expression	initial val. of x	val. of y	final val. of x
y = ++x	10	11	11
y = x++	10	10	11
y = --x	10	9	9
y = x--	10	10	9

int

→

int x = 10;	int x = 10;
y = ++x;	int y = ++10;
Sopln(y); 11	Sopln(y); CE

→ int x = 10;
int y = ++(++x); → nesting not allowed
↑ value
Sopln(y); CE

→

final int x = 10;	final int x = 10;	⇒ Can't apply for final var.
x = 11;	x++;	
Sopln(x); CE	Sopln(x); CE	

→

int x = 10	char ch = 'a';	double d = 10.5;	boolean b = true
x++	ch++;	d++;	d++;
Sopln(x);	Sopln(ch); 98	Sopln(d); 11.5	Sopln(b); CE

→ All primitive except boolean

```
byte b=10;          | byte b=10;          | byte a=10,
b = b+1             |   b++;              | byte b=20;
Sopln(b); ✗         | Sopln(b); ✓         | byte c=a+b;
   CE               |                     | Sopln(c);
                    |                     |   CE:
```

max(int, type of a, type of b)

max(int, byte, byte)

b = (byte)(b+1)

byte c = (byte)(a+b);

→ In case of inc. & dec. internal typecasting performed { b++; ⇒ b = (byte)(b+1); }

* Arithmetic operator (+, -, *, /)

→ If we apply any arithmetic operator b/w two var. a & b the result max(int, type of a, type of b)

byte + byte = int
byte + short = int
short + short = int
byte + long = long
long + double = double
float + long = float
char + char = int
char + double = double

→ Sopln('a'+'b'); 195
97 + 98 → int

Sopln('a' + 0.89); 97.89
97 + 0.89

Sopln(10/0); int/int → RE: AE: / by zero
Sopln(10/0.0); int/double Infinity
Sopln(0/0); int/int → RE: AE
Sopln(0.0/0); NaN (Not a Number) so AE

→ for byte, short, int, long - no way to represent ∞
→ for float & double we have Infinity(+/-)

x != NaN ⇒ true.

Sopln(10 < Float.NaN); false
Sopln(10 <=); false
Sopln(10 >); false
Sopln(10 >=); false
Sopln(10 ==); false

> NaN ⇒ false
>= NaN
<= NaN
< NaN
== NaN

Sopln(Float.NaN == Float.NaN); false
Sopln(10 != Float.NaN); true
Sopln(Float.NaN != Float.NaN); true

* A. Exception
  - ↳ gt is Runtime Exception not CE
  - ↳ gt is possible only in integral arth. but not floating point art.
  - ↳ The only operators which caus AE are 18%

46:00 min

#18. Operators & Assignments

08/Apr/23

* String concatenation operator (+)

→ The only overloaded operator is '+'.

```
String a = "durga";
int b=10, c=20, d=30;
Sopln(a+b+c+d);   durga102030
Sopln(b+c+d+a);   60 durga
Sopln(b+c+a+d);   30durga30
Sopln(b+a+c+d);   10durga2030
```

(a+b)+c+d

"durga10"+c+d

"durga1020"+d ⇒ "durga102030"

① a = b+c+d ; CE:

② a = a+b+c ;

③ b = a+c+d ; CE:

④ b = b+c+d ;

* Relational operator (<, <=, >, >=).

```
Sopln(10<20);   true
Sopln('a'<10);  false
Sopln(a<97.6);  true
Sopln('a'>'A'); true
Sopln(true>false); CE.
```

→ We can apply relational opr. for every primitive type except boolean.

Sopln("durga123" > "durga"); CE:

s1 > s2 ✗
s1 < s2 ✗

→ We can't apply relational opr. for object types.

Sopln(10<20) true

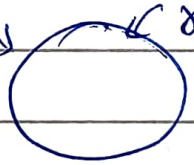
Sopln(10<20<30); CE: true<30

→ Nesting is not allowed.

* Equality operators (==, !=)

Sopln(10==20); false
Sopln('a'=='b'); false
Sopln('a'==97.0); true
Sopln(false==false); true

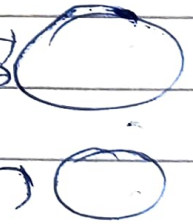
→ we can apply equality opr. for all primitive type.



r1==r2 ⇒ true.

→ we can apply equality opr. to object type also true if same reference

Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
Sopln(t1==t2); false
Sopln(t1==t3); true



```
Thread t = new Thread();
Object o = new Object();
String s = new String("durga");
Sopln(t==o); false
Sopln(o==s); false
Sopln(s==t); CE
```

→ For object type, compulsary there should be relation b/w argument type (P→C, C→P, P↔

* Diff. b/w == operator & equal() method.

→ == is ment for reference comparision.

→ equal() is ment for content comparision.

```
String s1 = new String("durga");
String s2 = new String("durga");
Sopln(s1==s2); false
Sopln(s1.equals(s2)); true
```

s1 → durga
s2 → durga

Note: for any object reference r,

r==null is always false but
null==null is always true

```
String s = new String("durga");
Sopln(s==null); false
Sopln(null==null); true
```

```
String s = null;
Sopln(s==null); true
```

31:00 min

#19. Operators & Assignments

09|Apr|23

* Instanceof operator.

→ To check type of object.

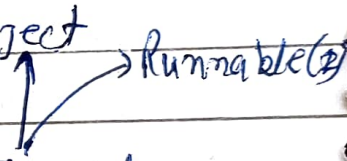
> object o = l.get(0);

> if (o instanceof Student) {
Student s = (Student)o; ... }
else if (o instance of customer) {
Customer c = (customer)o; }

⇒ Syntax: r instanceof X
object reference ↳ class | interface name

eg 1> Thread t = new Thread();

eg 2.> Sopln(t instanceof Thread); true
> Sopln(t instance of object); true
> Sopln(t instanceof Runnable); true



2.> Thread t = new Thread();
> Sopln(t instanceof String); → CE: inconvertible type.

→ Compulsary there should be relation b/w argument -type
(P→C) (C→P) (~~Cast~~ (same class).

⇒ null instanceof X; ⇒ false. {always}

* Bitwise operators (&, |, ^)

& → AND ⇒ Returns true iff both arg.s are true.

| → OR ⇒ Returns true iff atleast one arg. is true.

^ → X-OR ⇒ Returns true iff both arg. are different.

eg:> Sopln(true & false); false

> Sopln (true | false); true

> Sopln (true ^ false); true

> Sopln (4 & 5); 4 → 100 & 101 = 100

> Sopln (4 | 5); 5 → 100 | 101 = 101

> Sopln (4 ^ 5); 1 → 100 ^ 101 = 001

→ we can apply it on integral type also.

* Bitwise complement operator (~);

> Sopln(~true); → CE: Cannot applied to boolean.

→ We can apply it only for integral type but not for boolen type.

> Sopln (~4); -5

4 ≡ 0000 ---- 0100 (sign-bit)

~4 ≡ 1111 ---- 1011 (-ve)

↓ 2's complement

000 ---- 0100

0 ---- 0101

→ -ve no. represented in memory indirectly in 2's compliment form.

#20. Operators & Assignments

09/Apr/23

* Boolean complement operator (!)
> Sopln(!4); → CE:
> Sopln(!false); true
→ Applied only for boolean not integral typ.

& | ^ ⇒ Applicable for both boolean & integral types.

~ → App only for integral

! → App. only for boolean

29:13 min

* Short-Circuit operators (&&, ||)
> if(b_1 && b_2) { }
> else { } ← false

→ _&_ both argument get executed.
→ _&&_ If first arg. is false no execution of 2nd argument.
→ _||_ If first arg. is false then only 2nd argument will be ~~eva~~ executed.
→ &, | → Applicable for both boolean & integral.
&&, || → Applicable only for boolean.
→ performance is high.

> int x = 10, y = 15;
> if(++x < 10 & ++y > 15) { x++};
else {
y++; }
Sopln (x+"---"+y)

	x	y
&	11	17
&&	11	16
|	12	16
||	12	16

> int x = 10;
if (++x < 10 && (x/0 > 10)) {
Sopln ("Hello") }
else {
Sopln ("Hi"); }

① CE:
② RE: AE
③ Hello
④ Hi

→ if && ⇒ & → then RE: AE

* Type-cast operator
1 Implicit 2: Explicit

* Implicit type-casting
> int a = 'a';
Sopln (a); 97
→ compiler converts automatically by implicit type-c
> double d = 10;
Sopln (d); 10.0;
→ Smaller to bigger
→ Also known as widening or Upcasting
→ No loss of information

byte → short → int → long → float → double.
char → int → long → float → double.

2. Explicit type-casting

> int x = 130;
> byte b = x; → CE: PLP
> byte b = (byte) x;
> Sop(b) (-126)

→ Programmer is responcible to perform.
→ Bigger to smaller
→ Narrowing (or) down casting
→ Loss of information.

byte ← short ↖
int ← long ← float ← double
char ←

130

> int x = 130;
> byte b = (byte) x;
> Sop(b)

int x = 130 ⇒ 0000_0100 0010

2|130
2|65-0
2|32-1
2|16-0
2|8-0
2|4-0
2|2-0
1-0

byte b = (byte) x;
⇒ 1 0000010
-ve ↓ 2's complement

1111101
+ 1
11111110
$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$
= 64+32+16+8+4+2+0
= 126

> int x = 150; → 16 bit x = 000_0100 1011 0110
> short s = (short) x; s = 0000_0100 1011 0110
> Sop(s); // 150; true
> byte b = (byte) x; b = 1 0010110
> Sop(b); -ve

↓
1101010 → -106

#21. Operators & Assignments

> double d = 130.456;
> int x = (int) d;
> Sopln(x); 130 → digit after decimal point will be lost
> byte b = (byte) d;
> Sopln(b); -126

34:36 min

* Assignment operators:

1) Simple
   int x = 10;
2) Chained
   a = b = c = d = 20;
3) compound (+=, -=, *=, /=, %=, &=, |=, ^=, >>=, >>>=, <<=, ~~<<~~)
   a += 20;

> int a, b, c, d;
> a = b = c = d = 20;

> int a = b = c = d = 20; CE;

> int b, c, d;
✓ int a = b = c = d = 20;

> byte b = 0;
> b = b + 1;
> Sopln(b); CE:

byte b = 10;
b++;
Sopln(b); 11

byte b = 10;
b += 1; → (byte)(b+1)
Sopln(b); 11

> byte b = 127;
> b += 3
> Sopln(b); -126

→ Internal typecasting in compound assignment

> int a, b, c, d;
> a += b -= c *= d /= 2;

> a = b = c = d = 20;
Sopln(a + " " + b + " " + c + " " + d)
-160 -180 200 10

* Conditional operator (?:)

> int x = (10<20) ? 30:40;
> Sopln(x) ; 30

a++ ; ⇒ Unary
a+b ; ⇒ Binary
(c) ? b:d ⇒ Ternary

> int x = (10 >20) ? 30 : ((40>50) ? 60 : 70);
> Sopln (x) ; 70

* new operator

→ To create new object.
> Test t = new Test();
→ After creating an object constructor will be executed to perform initilization of an object.
Constructor is not for creation of object it is for initilization of object.
→ No delete keyword, it is responsibility of GC.

* [ ] operator

→ To declare & create arrays.
> int [] x = new int [];

21:50 min

#22 Operators & Assignments

09/Apr/23

* operator precedence

1. Unary
   [], x++, x--
   ++x, --x, ~, !
   new, <type>
2. Arithmetic
   *, /, %
   +, -
3. Shift
   >>, >>>, <<
4. Comparison
   <, <=, >, >=, instanceof
5. equality
   ==, !=
6. Bitwise
   &
   ^
   |
7. Short circuit
   &&
   ||
8. Conditional
   ?:
9. Assignment
   =, +=, -=, *=, ---

* Evaluation order of java operends.

> Class Test {

```
    P S V main(String [] args) {
      Sopln(m1(1) + m1(2) * m1(3) / m1(4) + m1(5) * m1(6));}
    public s int m1(int i) {
        Sopln(i);
        return i;          }
}   #op 1, 2, 3, 4, 5, 6, 32
```

→ There is no precedence for operends it get executed left to right.

9:55 min

#23. Operators & Assign. Interview FAQs

09/04/23

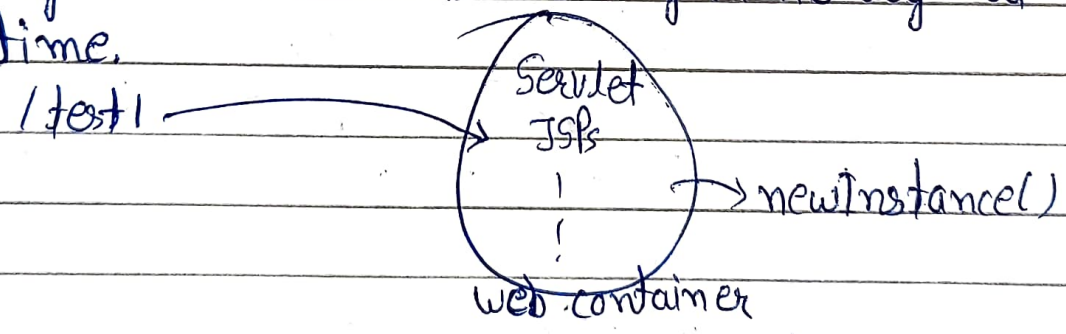
* new (Vs) newInstance()

new:

```
> Test t = new Test();
> Student s = new Student();
```
} ⇒ known class

```
> Class Test {
>     p s v main(String[] args) {
>         object o = Class.forName(args[0]).newInstance();
>         Sop.ln("Obj creat" +
>                 o.getClass().getName());
>     }
> }
```
} Java Test Student
Java Test Customer
↓
dynamic class.

→ new operator ⇒ if we know class name
→ newInstance is a method class class,
→ we can use newInstance() method to create object if we don't know class name in begning. & it is available dynamically at runtime.



→ In new operator, we can invoke any const.

```
> Test t = new Test();
> Test t1 = new Test(10);
> Test t2 = new Test("durga");
```

→ But newInstance() method internally calls no-arg constructor

→ To use it compulsary corresponding class should contain no-arg const. otherwise we will get R.E.: InstantiationException

→ If class file not available in new case of new

> Test t = new Test();
RE: NoClassDefFoundError : Test

→ while using newInstance() → if class file not there.

> Object o = Class.forName(args[0]).newInstance()
Java Test Test123 ↵
RE: ClassNotFoundException Test123

new	newInstance()
① It is operator.	① It is a method.
② Class name already known.	② Class name is dynamic.
③ Class not required to contain no-arg ~~metho~~ Constructor.	③ Compulsary class should contain no-arg constructor.
④ At runtime if class file not available then RE: NoClassDefFoundError.	④. RE: ClassNotFoundException

* Diff. blw ClassNotFoundException Vs NoClassDefFoundError

→ Test t = new Test(); → Unchecked
↳ RE: NoClassDefFoundError

→ Object o = class.forName(args[0]).newInstance();
Java Test Student ↲ → Checked.
RE: ClassNotFoundException : Student

→ For hardcoaded class names, at runtime if corresponding .class file is not available then we will get RE: NoClassDefFoundError,
→ It is unchecked Exception

→ for dynamically provided class name, at runtime if corresponding .class file is not available then we will get RE: ClassNotFoundException
→ It is checked exception.

* Diff instanceof	isInstance()
→ It is operator in java	→ It is method in java.lang
→ Used to check wether the given object is of particular type or not & we know type in the begning.	→ Used to check object type & we don't know type in begning & it is available dynamically at runtime.

→ Thread t = new Thread();
→ Sopln(t instanceof Runnable);
→ Sopln(t instanceof Object);

```
class Test {
    p s v m(S[] args) throws Exception {
        Thread t = new Thread();
        Sopln(class.forName(args[0]).isInstance(t)); }
}
```

Java Test Runnable ↵
o/p true
Java Test String ↵
o/p : false

→ isInstance() method is method equivalance of instanceof operator

(49.13 min)

#24. Flow-Control

09/Apr/23

Flow-Control

1. Selection statement
   - ① if-else
   - ② switch
2. Iterative statement
   - ① while()
   - ② do-while()
   - ③ for()
   - * ④ for-each loop (1.4v)
3. Transfer Stat.
   - ① break
   - ② continue
   - ③ return
   - ④ try-catch-finally
   - ⑤ assert (1.4v)

→ Flow control describe the order in which the statements will be executed at runtime.

4.50 min

25. Flow-Control

09/Apr/23

* Selection statement :-

1. if-else → should be boolean type

```
if (b) {Action}
else {Action}
```

```
> int x=0;
> if(x){Sop("Hello");}
> else{Sop("Hi");} CE:
> int x=10;
  if(x==20){Sop("Hello");}
  else{Sop("Hi");}  Hi
> boolean b=false;
  if(b==false){Sop("Hello");}
  else{Sop("Hi");}  Hello
```

```
>int x=10;
 if(x=20){Sop("Hello");}
 else{Sop("Hi");}  CE
> boolen b=true;
 if(b=false){Sop("Hello");}
 else{Sop("Hi");}  Hi
```

> if (true) | if(true); | if (true) | if(true){
Sopln("Hello"); | ✓ | ✗ int x=10; CE; | ✓ int x=10;}

→ else & {} are optional..

→ ~~One~~ ~~Stat~~ without {} should not be declarative stat.

→ ; is valid java statement (Empty statement)

> if (true)
if (true) ⇒ nearest if will be taken.
Sopln("Hello");
else Sopln("Hi");

12:40 min

26 Flow-Control 09/Apr/23

2. Switch

→ If several options are available go for switch.

> Switch (x) { {byte, short, char, int} ⇒ allowed. 1.4v
Case : Action-1; {Byte, Short, Character, Integer, Enum} 1.5v
break; {String} → 1.7v
default: default action; }

→ {} are mandatory.

→ both case & default is optional.

> ✓ Switch(x){ } ✓

→ Inside switch independent statement are not allowed it should be in case or default.

> int x=10, y=20;
Switch(x){ ⇒ Every case label should be compile time constant.
case 10:
case y: CE: ⇒ if y as final the no CE.
}

→ Duplicate case label is not allowed.
* Fall through inside switch.
→ We can define several case at a time.
→ Code re-usebility
> switch(x){
    >case 1: >case 2 > case 3: Sop("Hello"); break;

→ We can use default atmost.
→ Can ~~declare~~ write anywher, recommend at last.

#27. Flow-Control 09|Apr|23

* Iterative Statements
① while
→ If we don't know no. of iterations in advance the we should go for while loop.
Syntax:
> while (rs.next()) { }
> while (b) { } Should be boolean type.
    Action?
> while(1) {
    Sopln("Hello"); } CE:
→ {} is optional.

> while(true)	while(true);	while(true)	wile(true){
✓ Sopln("Hello");	✓	✗ int x=10;	✓ int x=10; }

> while(true){
    Sop1n("Hello");}
Sop1n("Hi");
↳ CE:

> int a=10,b=20;
while(a<b){
    Sop1n("Hello");}
Sop1n("Hi"); o/p: Hello...

> final int a=10,b=20;
while(a<b){
    Sop1n("Hello");}
Sop1n("Hi");
↳ CE: Unreachable stat

while(false){
    Sop1n("Hello");}
Sop1n("Hi");
CE:

> int a=0,b=20;
while(a>b){
    Sop1n("Hello");}
Sop1n("Hi"); o/p: Hi

> final int a=10,b=20;
while(a>b){
    Sop1n("Hello");}
Sop1n("Hi"); ↳ CE:

① final int a=10;
int b=20;
Sop1n(a);
Sop1n(b);
→ After compilation →
Sop1n(10);
Sop1n(b);

② final int a=10, b=20;
int c=20;
Sop1n(a+b);
Sop1n(a+c);
Sop1n(a<b);
Sop1n(a<c);
→ After comp. ⇒
Sop1n(30);
Sop1n(10+c);
Sop1n(true);
Sop1n(a<c);

→ Every final var. will be replaced by value during compile time.

do-while

Syntax:
```
do {
    body
} while (b);
```
→ should be boolean type.
→ mandatory.

→ If we want to execute loop body atleast once then we should go for do-while.

→ {} are optional

> do
Sopln("Hello");
while(true); ✓

> do;
while(true); ✓

do
int x=10;
while(true); ✗

do {
int x=10;
}while(true); ✓

> do
while(true); ✗

> do while(true)
Sopln("Hello");
while(false); ✓

do
while(true)
Sopln("Hello");
while(false); o/p: Hello

> do {
Sopln("Hello");
}while(true);
Sopln("Hi"); CE: ✗

int a=10, b=20;
> do {
Sopln("Hello");
}while(false);
Sopln("Hi");
o/p: Hello, Hi ✓

int a=10; b=20;
do{
Sopln("Hello");
}while(a<b);
Sopln("Hi"); o/p: Hello...

> int a=10, b=20;
do {
Sopln("Hello");
}while(a>b);
Sopln("Hi"); o/p: Hello
Hi ✓

> final int a=10, b=20;
do{
Sopln("Hello");
}while(a<b);
Sopln("Hi");
CE: ✗

> final int a=10, b=20;
do{
Sopln("Hello");
}while(a>b);
Sopln("Hi"); o/p Hello
Hi ✓

31:00 min

#29. Flow-Control

09/Apr/23

* for loop

> for(int i=0; i<10; i++){
    Sopln("Hello");
}

→ It is most commanly used loop.
→ If we know no. of iteration then it is best choice.

Syntax:-
→ for(init-section, condition-check, inc-dec-section){
    loop body
}

→ {} is optional.

* Initilazation :-

> int i=0, j=0;
  int i=0, String s="durga"; ✗
  int i=0, int j=0; ✗

→ Executed only once.
→ Declare & initialize local variable.
→ Can declare any no. of var but should of same type.

→ int i=0;
  for(Sopln("Hello Ur sleeping"); i<3; i++){
    Sopln("NO, Boss U only sleeping");
  }

  o/p: Hell Ur sleeping
  NO ___ ___ ___ 3

→ We can take any valid Java statement

* Conditional check

→ Can take any valid Java ~~statement~~ expression but final should be boolean.
→ This part is optional, by default ⇒ true.

Inc. & dec. section

```
int i=0;
for(Sopln("Hello"); i<3; Sopln("Hi")){
    i++; }
```
o/p: Hello, Hi, Hi, Hi

→ Can take any valid java statement.

```
for(;;){
  Sopln("Hello"); }
```

```
for(;;);
```

→ all three parts are independent of each other & optional.

```
for(int i=0; true; i++){
   Sopln("Hello"); }
Sopln("Hi"); → C.E
```

```
for(int i=0, false; i++){
   Sopln("Hello"); }
Sopln("Hi"); CE:
```

```
for(int i=0; ; i++){
  Sopln("Hello"); }
Sopln("Hi"); CE:
```

```
int a=10, b=20;
for(int i=0; a<b; i++){
   Sopln("Hello"); }
Sopln("Hi"); o/p: Hello
```

```
int a=10, b=20;
for(int i=0, a<b; i++){
   Sopln("Hello"); }
Sopln("Hi"); CE:
```

```
final int a=10, b=20;
for(int i=0; a>b; i++){
   Sopln("Hello"); }
Sopln("Hi"); CE:
```

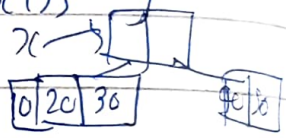
* for-each loop (Enhanced for loop) (1.5V)

→ It is for Arrays & collection.

eg: To print 1-D array

```
> int [] x = {10, 20, 30, 40};
```

Normal for loop	Enhanced for loop
`> for(int i=0; i<x.length; i++){`	`> for (int x1 : x){`
`>  Sop ln(x[i]); }`	`>  Sop ln (x1); }`



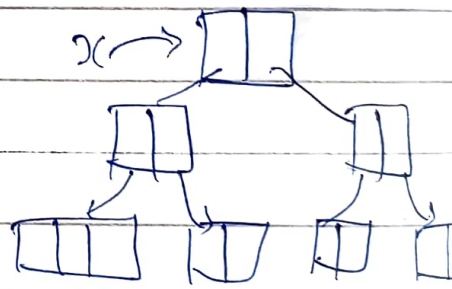
eg: To print 2-D Array

```
int [][] x = {{10,20,30}, {40,50}}
```

`for(int i=0; i<x.length; i++){`	`for (int[] x1 : x) {`
`  for(int j=0; j<x[i].len; j++){`	`  for (int x2 : x1) {`
`  Sop ln(x[i][j]); }`	`  Sop ln (x2) ;`
`}`	`}`

eg: To print 3-D Array

```
for(int[][] x1 : x) {
    for(int[] x2 : x1){
        for (int x3 : x2){
            Sop ln(x3); }
    }
}
```



→ It is only for array & collection ~~but its limitation~~

→ Normal loop ⇒ Any order but

for-each loop ⇒ only in original order.

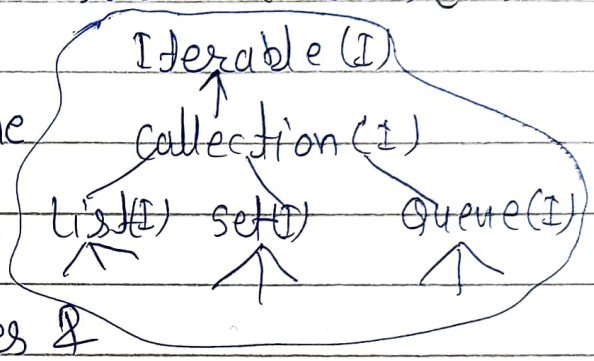
* Iterable (I)

```
for (eachitem x : target)
{
   ≡
}
```

target → Array / collection
target → Iterable object

only one method Iterator

J.1. Iterable (I) (J.5)



→ Iterable(I) only contain one method.

public Iterator iterator()

→ All array related classes & collection implemented classes alread implem-ent iterable interface.

* diff. Iterator (I)	Iterable (I)
① It is related to collection	① It is related to for-each lo
② We can use to retrieve the ele. of coll. one by one.	② The target element in for-each loop should be iterable.
③ Java.util pkg	③ Java.lang pkg.
④ 3 methods (i) hasNext() (ii) next() (iii) remove()	⑤ 1 method iterator()

40 min

#29. Flow-Control

* Transfer statement :-
* break

1) Inside switch :-

```
int x=0;
switch(x) {
    case 0 : Sop(0);
    case 1 : Sop(1);
             break }
```

2) Inside loops;

```
for (int i=0; i<10; i++) {
  if (i==5) { break;
    Sop(i); } o/p. 0 1 2 3 4
```

3) Inside labeled blocks:

```
Class Test {
    P S V m(S[] args) {
        int x=10;
        l1: { Sop("begin");
            if (x==10) break l1;
              Sop("end"); }
        Sop("Hello"); }
    }
```

→ Only 3 places we can use break inside switch to stop fallthrough.
→ To break loops based on some condition
→ Inside labeled blocks to break based on some condition.

* Continue

```
for (int i=0; i<10; i++) {
    if( i%2 ==0){
        continue; }
    Sop(i);
}
```

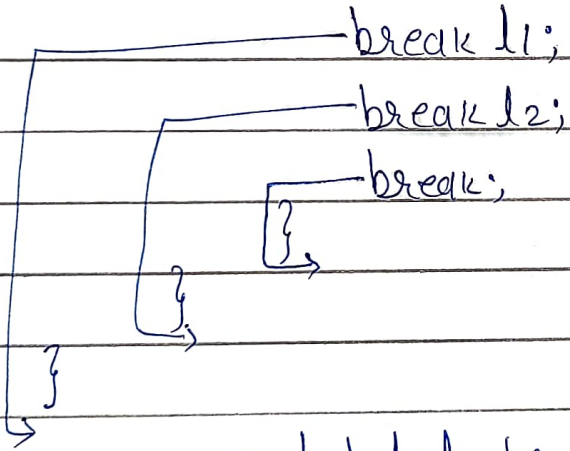
o/p: 1 3 5 7 9

→ We can use continue inside loop to skip current iteration & continue for next iteration.

→ we can use continue only inside loop.

* labeled break & continue

```
l1:
for(_____){
    l2:
    for(_____){
        for(_____){
            break l1;
            break l2;
            break;
        }
    }
}
```



→ we can use labeled break & continue to break or continue tha particular loop in nested loops.

```
l1:
for(int i=0; i<3; i++){
    for(int j=0; j<3; j++){
        if(i==j){ break ;}
        Sop(i+"---"+j);
    }
}
```

break;	break l1;
1--0	no o/p
2--0	
2--1	

continue;	continue l1;
0--1	1--0
0--2	2--0
1--0	2--1
1--2	
2--0	
2--1	

* do-while() Vs continue (dangerous comb.)

```
> int x=0;
  do {
     x++;
     Sop ln(x);
     if(++x<5) {continue;}
     x++;
     Sop ln(x);
  } while (++x<10);
```

o/p
1
4
6
8
10

x=0 1 2 3 4 5 6 7 8 9 10 11

22 min