9/2/24

⑤ Bitwise Operators ⇒ works on each bit of integers.

① & AND

② | OR

③ ^ XOR

④ ~ NOT

⑤ << Left shift

⑥ >> Right Shift

⑦ >>> Right shift zero fill

5 & 7 ⇒ Bitwise AND

a<b & 5 error

a<b & b<c ⇒ Boolean logical AND

Examples ⇒

① int a=10 & 12;

$(1010)_2$ & $(1100)_2$ = $(1000)_2$ ⇒ $\boxed{a=8}$

② int a, b, c;

a=25;

b=13;

c=a&b;

SOP (c); ⇒ c=9

25 = $(11001)_2$

13 = $(01101)_2$

01001

11101

10101

c=a|b;

SOP(c) c=29

c=a^b;

SOP(c) c=20

③ ~

⇒ {first bit of any integer no. is sign bit}

0 = +ive , 1 = -ive

⇒ To evalute -ive no. we have to take 2's complement of it.

⇒ 2's complement = 1's complement +1

{0→1 , 0→1}

Eg ⇒ ① int i= ~10;

SOP(i)

$(10)_b$ ⇒ $(1010)_2$

00001010

(~10) ⇒ 11110101 {negative}

⇓ 2's

00001010

+1

001011 ⇒ (11)

$\boxed{-11}$

② int i=-25

SOP(~i)

+25

⇒ $(11001)_2$

⇓ 2's

111 00110

+1

(111 00111) ⇒ $\boxed{-25}$

~(-25) ⇒ (0011000) ⇒ $\boxed{24}$

~(-25) = 24

④ << {0 will add on the Right side}

int i = 6 ⇒ $(6)_{10} \Rightarrow (--0110)_2$

SOP(i<<2);

0011000

⇒ 24

int i = -8;

SOP(i<<4);

$(-8) \Rightarrow (--0100)_2$

1's 2's

--1111011

+1

$(-8) = (--01111 1000)_2$

$(---11 11 10000000)_2$

1's 2's

$(---1011111)$

+1

$(10000000)_2$

(128)

$\boxed{-128}$

Java

Case-I

In Case of shifting more than 32 eg (x<<33) then actual shifting will be according to modulus 32

33 % 32 = 1

x<<1

12/2/2024

a << 32;

⇒ a << 32 % 32

⇒ a << 0

⇒ a

{32 because size of int}

Case II

a << -50

{In case negative value for shifting than +32 till value becomes +ive}

-50 + 32 = -18 + 32 = 14

(a << 14 == a << -50)

{If we are shifting long variables then we will use 64. We can perform shifting only on int, long}

⑤ >> — Right shift

int a = 10;

SOP(a>>2);

$(10)_{10} = (00001010)_2$

>>

$(00000010)_2$

$= (2)_{10}$

i = -20;

SOP(i>>3);

$(-20)_{10} = (00010100)_2$ = 20

11101011

+1

$(--11101100) = -20$

$(11111110 1)$

{In case of Right shift, addition of bits will be dependend on sign bit.}

$(---000010)$

$(---011)_2 = (3)$

$\Downarrow$

$(-3)_{10}$

$-23 >> 2$

$(-23)_{10} = (00010111)_2 = (23)_{10}$

11101000

$(11101001)_2 = (-23)_{10}$

$(11111010)_2 \Rightarrow (00000101)$

$(--0110)_2$

$(-6)_{10}$

(6) $>>>$ $\Rightarrow$ Right shift with 0 fill

$\Rightarrow$ Unsigned Right shift.

{Here, New bit that will added to the number will always be '0'.}

$a = -15$

$a >>> 2;$

$(00001111)_2 = (15)_{10}$

$(11110000)_2$

$+1$

$--11110001)_2 = (-15)_{10}$

$(0011 ------ 11110 0)_2$ = Very big number --

eg $\Rightarrow$

$a = -10$

$b = a >>> 2;$

SOP(b)

What will be value of 'b'.

a) very big -ive value

(b) very big +ive value

(c) 1

(d) 0

++ increment — Pre increment, Post ~~decrement~~ increment

-- decrement — Post increment, Predecrement

```
int a = 10;
int b;
b = a++;
SOP(b)
SOP(a)
b = ++a;
```

SOP(b)

SOP(a)

{10, 11, 12, 12}

```
int a = 10;
int b;

b = a--;
SOP(b);        (10, 9, 8, 8)
SOP(a);

b = --a;
SOP(b)
SOP(a)
```

13/2/24

Arithmetic Assignment Operators

```
a = a + 5;
```

```
a += 5;        a *= 10;
a -= 5;        a |= 10;
a *= 5;        a ^= 10;
a /= 5;        a >>= 2;
a % = 5;       a >>>= 2;
               a <<= 2;
```

Ternary operator :→

Three operands

⇒ ? :

```
a = 10; b = 15; c = 20;

max = a > b ? a : b;
```

$\boxed{\text{max} = 15}$

Arithmetic Expressions

Rule 1 ⇒ If byte, short, or char is available in any of the expression, that is converted into int before evaluation of expression. This is true even for Unary operators / expressions.

eg ⇒

```
byte b = 10;
b = b + 1;   /* will give us error because before evaluating addition b will be converted to int then int + int & we would not be able to assign int value to byte. */

b = -b   // error
```

Rule 2 ⇒ If one of the operand is Long then convert int to long.

```
byte b = 10;
long l = 5;
b + l;
```

b → Int, l → Long → long + long

Rule 3 ⇒ If one of the operand is of type float, other one will be converted to float.

```
byte b1 = 10;
float f1 = 2.5f;
b1 + f1;
```

float ← int ← byte

```
int i = 10;
float f1 = 2.5f;
i + f1;
```

int → float, float

long l1 = 10;
float f1 = 2.5f;
l1 + f1;
↓
long
↓
~~float~~

byte / short / char / Int / long + float
↓
float.

⇒ Float with double, float will be converted to double.

float f1 = 2.5f;

double d1 = 35;

f1 + d1;
↓
float
↓
double

Integer Expression

where both operands are of integral type {Byte, short, int, char, long}.

Rule! ⇒ If integer is divided by 0, you will get Arithmetic exception. {Runtime error}

Underflow → Over the min value ← 0 → Over the max value ← Overflow

In both cases, truncation will happen

We cannot directly assign a value bigger then integer, and this will only happen in arithmetic exp. → ex ⇒ int i = 500 * 500 * 500 * 5000;

size of int = 32 bit

whatever bits are outside or over 32 bit, it will ignore those bits and give output according to 32 bit.

Floating Point Expression

To get max / Min value in int we have
Integer.MAX_VALUE;
Integer.MIN_VALUE;

We have three special values here :→

① double d1 = 10.0;
d1 = d1/0;
SOP (d1);
⇒ we give us output 'infinity'

② double d1 = -10.0;
d1 = d1 / 0
SOP(d1)
⇒ '-Infinity'

(3) double d1;

d1 = 0.0/0;

SOP (d1);

Output ⇒ NaN;

{Not a Number}

(4) double d1 = Math.sqrt(-25);

SOP (d1); Complex Number

Output : NaN;

underflow

-1×10^{1000} -1×10^{-1000}

↓ ↓

−Infinity 0

Overflow

1×10^{1000} 1×10^{-1000}

↓ ↓

+Infinity 0

Conversion / Casting

→ Implicit Conversion {Automatic}
↳ widening

→ Explicit Conversion
↳ Narrowing

Implicit Conversion

{widening} — Automatic

byte
↓
Short Char
↓ ↓
int
↓
long
↓
float
↓
Double

Example ⇒

byte b1=10; short s1=10;

int i=100 long l1= 10; Size

float f1=10.0f ; double d1= 2.5;

✓ i = b1 ; i = s1 ;

✓ l1=i; f1= i ; f1 = l1;

✓ f1=s1; d1 = f1; d1 = i;

Explicit Conversion {Higher to Lower}

short s1= 10;

byte b1= 10;

int i = 10;

long l1 = 10;

float f1= 2.5f;

double d1= 25;

f1= d1 {Error}

f1 = (float) d1; ✓

i = (int) l1;

i = (int) d1;

Three exception are there in this case :→

(1) increment / decrement case, autocasting happens.

Example ⇒

byte b1 = 10;

b1 = b1+1; //error

b1 = (byte)(b1+1); ✓

or

b1 = b1++ ✓

or

b1= --b1 ✓

② In case of Arithmetic Assignment, auto casting happens.

ex → byte b1 = 10; ✓
b1 += 1; ✓

③ If int literal is in range then auto casting happens.

eg → byte b1 = 8; ✓

b1 = 130 // error because out of range.

byte b1 = 10+5; ✓ → literal

byte b1 = 125+100; → Error because Out of range

b1 = b1+5; ✗ → variable

14/2/2024

Input from User

⇒ JDK 5 we got Scanner class.

example

```
import java.util.*; OR import java.util.Scanner;
class Add2Numbers{
    PSVM(String[] args){
        int a,b,c;
        a=10; b=20;
        c=a+b;
```

Output required = 30

Sum = 30

Sum of 10 & 20 = 30

SOP(sum); // 30

SOP(a+b); // 30

SOPC "sum = "+c); // Sum = 30

SOP("Sum = "+a+b); // Sum = 1020

SOP("sum of "+a+"&"+b+"="+c);

Sum of 10 & 20 = 30.

```
        Scanner sc = new Scanner(System.in);
        int a,b, sum;
        S.OP("Enter a number: ");
        a = sc.nextInt();
        S.OP("Enter 2nd number: ");
        b = sc.nextInt();
        Sum = a+b;
        S.O.P(sum);
        OR
        S.OP("sum = "+sum);
```

Concatination

+ works from left to right

"sum = "+a+b

⇒ Step1 = "sum = a"+b;

→ Step2 = "sum = a b"

Arrays

⇒ Collection of elements of same type.

⇒ Dynamic Size arrays

⇒ To use array, we have three steps.

Step1 ⇒ Declaration

1D array ⇒

```
int arr1[];
   OR
int [] arr2;
```

Step2 ⇒ Memory Allocation

```
arr1 = new Int[5];
   OR
arr2 = new Int[10];
```

Step3 ⇒ Initialisation.

int [] arr1, arr2
arr1, arr2 = int array.
	int arr1[], arr2;
	arr1 = int array
	arr2 = int variable
		int a = 5;
		int arr[]
		arr = new Int[6]
		arr[0] = 5;
		arr[1] = 10;
		arr[2] = 20;
		arr[3] = 30;
		arr[4] = 40;

⇒ Once memory allocation is done for all elements of array, then default allocation happens.

eg

```
int arr[] = new Int[5]
SOP (arr[0]); // Output = 0.
```

eg

```
int arr[] = new int[5];
// index range 0 to 4
arr[5] = 10;
arr[6] = 20;
```

⇓

will compile successfully ⇒ But at runtime will give us exception 'java.lang.ArrayIndexOutOfBoundException

byte = 0	
Char = '\0'	
Short = 0	
int = 0	Default Values
long = 0	
float = 0.0f	
double = 0.0	
boolean = false	
Object = null	

⇙ int arr[] = {5, 10, 20, 30}

⇙

Declaration + Memory allocation + Initialisation {4}.

OR

⇙ int arr[] = new int[]{5, 10, 20, 30};

16/2/24

```
Class A {
    Public Static Void main (String [] args){
    }
}
```

Compile => javac A.java
⇓
A.class

Run => java A

If we want to pass arguments at run time then they will be stored in string array 'args'. These are known as Command Line argument.

⇓

=> java A 10 20 30

```
# SoP ( args[0] + args[1] + args[2])
```

=> Output => 102030

{size of the array (args) depends upon arguments passed at run time, for example, in above case it will be 3}

```
class addNumbers{
    PSVM (String [] arr){
        int num [], int sum=0;
        num = new int [arr.length];
```

{array is an Object and length is a property of array object.}

```
for (int i=0; i<arr.length; i++){
    num[i] = Integer.parseInt (arr[i]);
    sum = sum + num[i];
}

So.P (sum);
}
}
```

=> java addNumbers 10 20 30

:> Output => 60

Java arrays can have dynamic size

=> Size of array in java is decided at runtime.

PSVM (String[] arr)
⇓
OR
String arr[]
OR
String ...arr => Variable length arguments {Varargs}

```
int a=5;
int arr[];
arr = new int [a];
```

2-D array

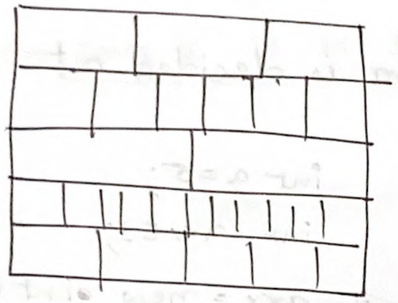
int arr[][];

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

In C/C++ 2-D array we use to say it is a matrix in java That is not necessary in, where we Can also have zic-zac array

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]
arr[2][0]	arr[2][1]	arr[2][2]

⇓



In Java each row of 2-D need not to have same no. of Column

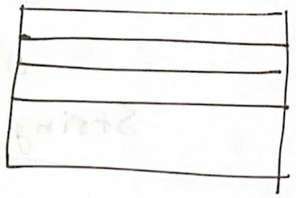
OR

Java can have zic-zag type of array.

⇓

int arr[][];

arr = new int[4][];

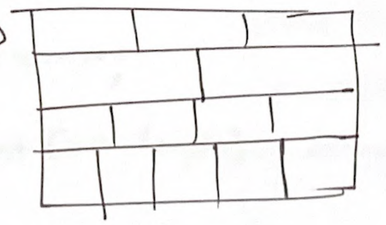


arr[0] = new int[3];

arr[1] = new int[2];

arr[2] = new int[4];

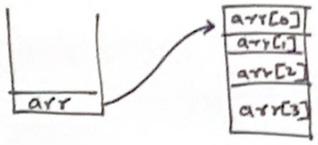
arr[3] = new int[5];



In Java, 2-D array is strictly array of arrays.

int arr[][]; ⟶ Memory will be allocated to arr.

arr = new int[4][];

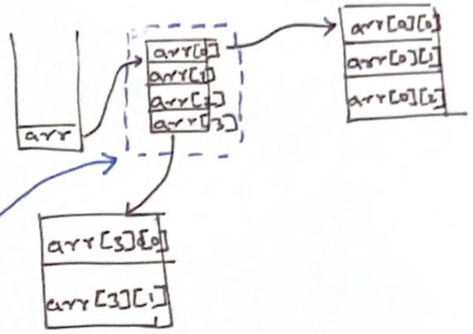


arr[0] = new int[3] ⇒

arr[1] = new int[2];

arr[3] = new int[2];

arr[2] = new int[4];



Sop (arr.length);

⇓

output = 4

arr[0].length ⇒ 3

arr[1].length ⇒ 2