Lecture 13

Bitwise operators

- These operators work on individual bits of operands.
- They work at binary levels and used in the scenarios where we need to manipulate data efficiently such as in system programming, cryptography and low-level operations.
- Java is not typically used for system programming, but languages like C & assembly heavily rely on low-level operations.
- Mostly used in embedded systems applications which work in resource constrained environments. As these things have limited memory and data has to be manipulated very efficiently and bitwise operators play an important role here.

→ There are following bitwise operators in Java:-

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise NOT (~)
4. Bitwise XOR (^) (exclusive OR)
5. Bit Shift Operators
   - → Left Shift (<<)
   - → Right Shift (>>)
   - → Unsigned Right Shift (>>>)

BitWise AND (&) :- it returns 1 if both input bits are 1.

eg:- a = 10 & b = 5

a & b = 0

```
00000000 00000000 00000000 00001010 (a)
00000000 00000000 00000000 00000101 (b)
00000000 00000000 00000000 00000000
```

② **Bitwise OR (|) :-** it returns 1 if any of the input bit is 1.

e.g:- a = 10 , b = 5;

a | b ⇒ 15

```
00000000 00000000 00000000 00001010 (a)
00000000 00000000 00000000 00000101 (b)
-----------------------------------
00000000 00000000 00000000 00001111
```

③ **Bitwise XOR (^) :-** it returns 1 if both bits are different, otherwise 0.

OR

it returns 1 if any of the input bit is 1, but not both.

e.g:- a = 10, b = 5

```
00000000 00000000 00000000 00001010 (a)
00000000 00000000 00000000 00000101 (b)
-----------------------------------
00000000 00000000 00000000 00001111
```

a ^ b = 15

We can apply these bitwise operators (&, |, ^) on integers & char values not on floating point numbers. These operators (&, |, ^) can also be applied on boolean values.

eg:- boolean b1 = true;
boolean b2 = false;

b1 & b2 → false
b1 | b2 → true
b1 ^ b2 → true

But the operators &, |, ^ are referred to as bitwise only when they are applied on integer operands

NOTE: There is no short-circuiting in Bitwise & and | → these are non-short-circuit operators

eg:-
```
int a = 10, b = 20, c = 30, d = 40;
if ((a<b) | (++c<d)){
    System.out.println("Both conditions are true");
}
System.out.println("c="+c);  → 31
```

Here even if (a<b) is true but still second condition would be evaluated & thats why c value becomes 31

④ Bitwise NOT (~) :- It inverts bits. 0 becomes 1 & 1 becomes 0.

e.g:-
```
int a = 10;
~a = -11
```

00000000 00000000 00000000 00001010 (a)
11111111 11111111 11111111 11110101 (~a)

↳ this is -11 in 2's complement

NOTE:- Bitwise NOT (~) can not be applied to boolean values.

Compound Bitwise Assignment :-

Combining bitwise operations with assignment.

eg:-
```
operand1 = operand1 operator operand2
operand1 operator= operand2
```

→
```
a = 10, b = 5;
a = a & b
```
Can be written as `a &= b` → this performs Bitwise AND on a & b, then assigns the result to a.

Same we can write:-

```
a |= b;
a ^= b;
```

Practice Time :-

```
char ch1 = 'A', ch2 = 'B';
```

'A' = \u0041 → 0000 0000 0100 0001 ⇒ 65

'B' = \u0042 → 0000 0000 0100 0010 ⇒ 66

⇒ System.out.println(ch1 & ch2) ⇒ 64
System.out.println(ch1 | ch2) ⇒ 67
System.out.println(ch1 ^ ch2) ⇒ 3
System.out.println(~ch1) ⇒ ~66

Answer this:-

```
float a = 10, b = 5;
System.out.println(a & b); = ?
```

output

Bit Shift operators :- Shifts the bits.

Left shift:- shifts the bits to the left by a specified number of positions. given on right.

eg:-

```
int a = 5;
a << 1
```

00000000 00000000 00000000 00000101
00000000 00000000 00000000 00001010 → 10

5<<1 00000000 00000000 00000000

→ This operation inserts 0s from the right and discards any bits that move out of the left side

Here $a = 5$

$a << 1 = 10$ (5×2^1)

⇒ So the answer is same as multiplication by powers of two (assuming no overflow occurs)

eg:- $5 << 2 = 5 \times 2^2 = 5 \times 4 = 20$

$5 << 3 = 5 \times 2^3 = 40$

IMP

Suppose:- int a = 1;

int result = a << 31 = ? = −2147483648 [because of overflow]

But:- long a = 1L;

long result = a << 31; ⇒ 2147483648 [No overflow here]

Unsigned Right shift:- shifts the bits to the right, by the specified number of positions.

- Right shift the left operand by the specified number of positions given on right.

eg:- int a = 10;

a >>> 1

10 → 00000000 00000000 00000000 00001010

10>>>1 ⇒ 00000000 00000000 00000000 00000101 ⇒ 5

- inserts zeros at leftmost bits (higher order bits) & ignores the sign bit.
- This is useful for unsigned operations
- same as division by powers of 2

eg:- 10 >>> 1 ⇒ $10/2^1 = 5$

10 >>> 2 ⇒ $10/2^2 = 2$

The >>> operators shifts the bits to the right & fills the leftmost bits with 0, regardless of the sign of the orginal number.

→ This is why it is called "unsigned" – it does not preserve the sign bit

- The sign bit (MSB) is replaced with 0, treating the number as positive after the shift.

e.g:-
```
int a = -10;
int result = a >>> 2;
System.out.println(result); ⇒ 1073741821
```

Signed Right-Shift Operators (>>) :→

↳ shifts bits to the right while preserving the sign of the orginal value. It means that if the value is negative, the sign bit (MSB) is filled with 1s and if the value is positive it is filled with 0s.

e.g:-
```
int a = 10;
a >> 1
```
⇒ Case-I (number is +ve)

10 → 00000000 00000000 00000000 00001010

10>>1 ⇒ 00000000 00000000 00000000 00000101 ⇒ 5

↑ inserted 0

```
int a = -10
a >> 1
```
Case II:- (number is -ve)

-10 ⇒ 11111111 11111111 11111111 11110110

-10>>1 ⇒ 11111111 11111111 11111111 11111011 ⇒ -5

↑ inserted 1.

⇒ We can say that >> is same as >>> but padded with MSB

Compound Bit shift Operators :-

Combining bit shift operators with assignment.

eg:- `int a = 10;`

~~`int result =`~~

`a <<= 1;` same as `a = a << 1;` = 20

`int a = 10;`

`a >>= 1;` same as `a = a >> 1` ⇒ 5

`int a = 10;`

`a >>>= 1;` same as `a = a >>> 1` ⇒ 5

Applications :-

1. Embedded Systems | Embedded Programming
2. Games Programming
3. Performance optimization (Faster Computations)
   - ↳ Instead of using * or /, shifting bits left or right can speed up the calculation.
4. Network Programming (IP Address Manipulation)
   - ↳ used to manipulate & compare IP addresses & subnet masks
5. Cryptography
   - ↳ Encryption algorithms (AES, DES) use bitwise operations
6. Graphics Programming (Color Manipulation)
7. Huffman Encoding - Data Compression Algorithms

→ Bitwise operators have lower precedence than Arithmetic operators.

Operator Precedence & Associativity Chart :-

Precedence	Operator	Type	Associativity
1	( )	Parantheses	Left-to-Right
	[ ]	Array Subscript	
	.	Member Selection	
2.	++	Unary Post increment	L-R (Left-to-Right)
	--	Unary Pre increment	
3	++	Unary Pre-increment	R-L (Right-to-left)
	--	Unary Pre-decrement	
	+	Unary plus	
	-	Unary minus	
	!	Logical Not	
	~	Bitwise NOT	
	(type)	Unary type Cast	
4.	*	Multiplication	L-R
	/	Division	
	%	Modulus	
5.	+	Addition	L-R
	-	Subtraction	
6.	<<	Bitwise Left Shift	L-R
	>>	Signed Right Shift	
	>>>	Unsigned Right Shift	
7.	<	Relational less than	L-R
	<=	less than equal to	
	>	greater than	
	>=	greater than equal to	
	instanceof	Type Comparison (objects only)	

8.	== !=	Relational is equal to is not equal to	L-R
9	&	Bitwise AND	L-R
10	^	Bitwise XOR	L-R
11	|	Bitwise OR	L-R
12	&&	Logical AND	L-R
13.	||	Logical OR	L-R
14	?:	ternary operator	Right-to-Left
15.	= += -= *= /= %=	Assignment Addition Assignment Subtraction Assignment Multiplication Assignment Division Assignment Modulus Assignment	Right-to-Left

NOTE:- Smaller number means higher precedence.