March 3rd :

→ Operator

*):- operand is any object which ~~that~~ are effected by operator

Ex:-
```
fun main () {
    var x = 12;
    val y = 3
    ~~dat~~
    val result = x+y;
     println("$result");
 }
```

→ The operands here are 'x' and 'y' because those variables are effected by operator '+'

*) Without 'result' variable, output the result in console, But How

A):- println("x+y = ${x+y}"); // with the help of placeholder we can achieve it.

→ In Kotlin when we perform arithmatic operation involving a 'Double' & 'Int', the result is automatically promoted to double This behaviour is due to Kotlin's type promotion rules, which promote the smaller data type to larger data type in order to maintain precision and prevent loss of data.

Ex:-
```
fun main() {
    val x = 12.0
    val y = 3
    val result = x+y;
     println("$result"); // 15.0
```

→ result= result+2 ; (result = result *2 → result *=2;).

$\underbrace{\text{result= result+2}}$

this can be written as result+=2

at first right hand side will execute & that assign value to the left-side variable.

→ operator precedence:

*) code:-

```
println("3+3*4=${3+3*4}"); BODMAS
```

⇒ output:- 3+3*4= $\boxed{15}$

Multiplication have more precedence than addition

3*4=12

⇒ 3+12= $\boxed{15}$

→ look at the precedence table carefully:-

Note:- elvis operator:-("?:").

This operator is used for null safety checks and has right to left precedence. It provides consise way to handle nullable values by providing default if the expression on the ~~fet~~ left side evaluate to null.

(?!) → used to check null safety checks & uses right to left precedence.

x++ & ++x → increment & assign ⇒ [x= x+1]

↓

assign & Increment

left to Right:-

Precedence	operators	Description
Highest	()	Grouping
	[ ]	Array access
	.	Member access
	++, --	Increment & decrement
	!, +, -	Unary plus, unary minus logical negation.
	*, /, %	Multiplication, division, remainder
	+, -	addition, Subtraction
	..	Range
	in	Membership
	is, !is	Type checks
	&&	logical AND
(?)	\	
	?:	Elvis operator
	=, +=, -=, *=, /=	Assignment operator
lowest.	==, !=, >, <, >=, <=, ===, !==	Comparison operator

Right to left

*)

Precedence	operators	Description
Highest	=, +=, -=	Assignment operator
	?:	Elvis operator
	&&	logical AND
	\	
	is, !is	Typechecks
	in	Membership
	..	Range
	+, -	Addition, subtraction
	*, /, %	Multiplication, division, remainder
	!, +, -	unary plus, unary minus, ~~unary~~ logical negation.
	++, --	Increment & decrement
	(), [], .	Grouping, array access, member access
Lowest	==, !=, >, <, >=, <=, ===, !==	Comparison operator.

*) Operators at the top have higher precedence, meaning they are evaluated first. operators at the bottom have lower precedence and are evaluated last. In case where operators have same precedence evaluation occurs from left to right.

→ left to Right precedence:-

*) This is the most common precedence order in programming languages and reflects the natural reading order for many human language.

*) More arithimatic & logical have left to right precedence.

Right to left precedence

*) ~~It~~ this precedence order is less common but is used for specific operation & scenario's where right to left evaluation makes sense.

*) :- It's often employed with assignment operators & operators with right-associativity, where the operations is conceptually applied from right to left.

→ left to Right precedence:-

*) This is the most common precedence order in programming languages and reflects the natural reading order for many human language.

→*) More arthimatic & logical have left to right precedence.

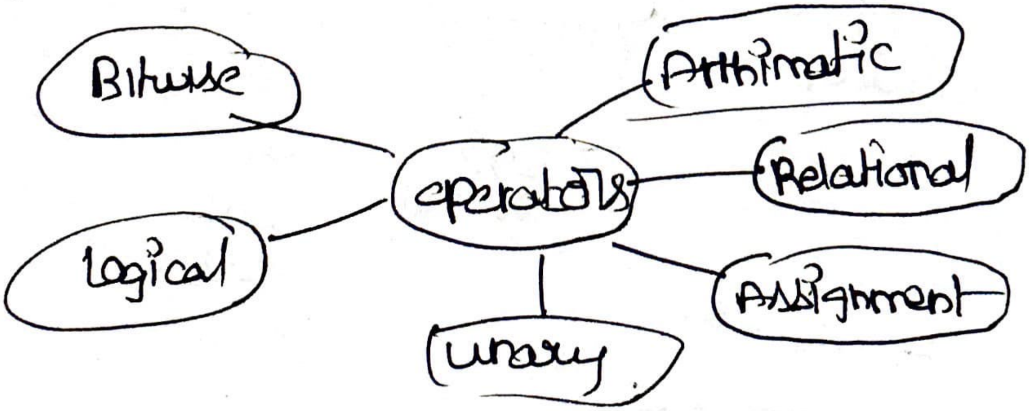
Right to left precedence

*) This precedence order is less common but is used for specific operation & scenario's where right to left evaluation makes sense.

*):- It's often employed with assignment operators & operators with right-associativity, where the operations is conceptually applied from right to left.

Variable

✓ Note:- Immutable variable is not a constant because it can be initialized with the value of a variable. It means the value of immutable variable doesn't need to be known at compile time, and if it is declared inside a construct that is called repeatedly it can take on different value on each function call.



Bitwise operators

operator	Meaning	Expression.
1) shL	signed shift left	a.shL(b)
2) shr	signed shift right	a.shr(b)
3) ushr	unsigned shift right	a.ushr()
4) and	bitwise and	a.and(b)
5) or	bitwise 'OR'	a.or()
6) xor	bitwise XOR	a.XOR()
7) inv	bitwise inverse	a.inv()

→ do 2's complement

=) 36 bitwise and 22

=)

④

bitwise'i

2|36
2|18 − 0
2|9 − 1
2|4 − 0
2|2 − 0
2|1 − 1

110010

2⁴

16

00001110

→ 11110001

+ 1

11110001

1010 by 2bit

1011 − 14

100 → ②.

②

2¹2⁰

= 2+1 → ③.

00 − 0
01 − 1
10 − 2
11 − 3
100 − 4
101 − 5
~~01~~

1010

$2^3 2^2 2^1 2^0$

=) 8 0 2 1

=) ⑪

2|14 − 0
2|7 − 1
2|3 − 1
2|1 − 0

10110

2^4 2^2 2^1