// check the documentation in the code

Operators

Incrementation: increasing exiting value by 1

Post incrementation a++

pre incrementation ++a

Eg: Incrementation\_1

Eg: Increment

Eg:Increment\_Eg\_2

// if in print statement post increment is applied, first printed and then incremented

// even though if we use “-“ in increment the result will be positively incremented . but since it is written in print statement first “a” is printed then incremented. and “a” is printed which consists of incremented value.

Eg: Increment\_Eg\_3

Eg: Increment\_Eg\_4

// Here post increment is applied i.e in if condition 5 is loaded in i, and then condition check since 5<6 ,it enters into the loop

And after that in print statement again post increment i.e first 6 is printed and then incremented

Eg: Increment\_Eg\_5

// check documentation in code

Eg: Increment\_6

1. During the execution of line 3, since bracket is present the compiler first compiles the code in the bracket
2. Now evaluation is done for the code inside in bracket i.e ++x (pre increment is done and x will be incremented to 5)
3. Now outside the bracket, since x is incremented in braces to 5, we can’t increment literal, which throws compilation error.

Eg: Increment\_Eg\_7

//check documentation in the code

Eg: Increment\_Eg\_8

// go through the code

Eg: Increment\_Eg9

Eg: Increment\_Eg10

Since post increment first variable is fetched and then incremented. the condition is checked with the original value not with the incremented one.

% in arithmetic operator stores remainder.

Eg: Modulo\_Operator

Logical operator :

&& Short circuit AND operator -> all the conditions should be true only then result is true else false

Second operand evaluation will happen only if first operand evaluation is true

Eg: Short\_Circuit\_AND\_Operator

Eg: Short\_Circuit\_AND\_Eg\_2

|| Short circuit OR operator -> if one condition is true result will be true, it will not check for other condition. If both conditions are false result will be false.

Second operand evaluation will happen only if first operand evaluation is false

Eg: Short\_Circuit\_OR\_Operator

Eg: Short\_Circuit\_OR\_Operator\_Eg2

// go through the program

Eg: Short\_Circuit\_OR\_Operator\_Eg3

// first condition is true , so other condition is not evaluated .

Bitwise

& Logical AND operator (or) Bitwise And -> truth table is same as Short Circuit AND Operator

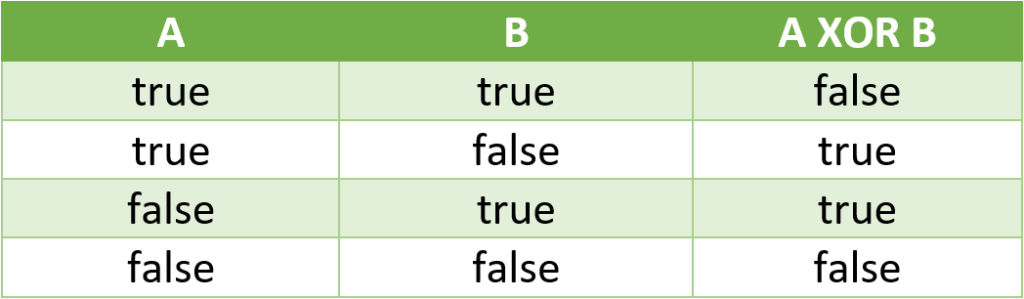
The only difference is Bitwise And checks both the conditions, even though first condition evaluates to false.

Eg: Bitwise\_AND

| Bitwise Inclusive OR -> truth table is same as Short Circuit OR Operator. The only difference is Bitwise OR checks both the conditions, even though first condition evaluates to false.

Eg: Bitwise\_Inclusive\_OR

^ Bitwise Exclusive OR ->



Evaluates both the conditions. Even though the first condition evaluates to true.

Eg: Bitwise\_Exclusive\_Or\_Not\_Compliment

// go through the code

Eg: Bitwise\_Exclusive\_Or

// go through the code

! -> NOT operator

!true -> false

!false -> true

~ -> bitwise compliment

-(n+1)

Eg : ~ 25 -> -(25+1) -> -26

Eg: ~ -36 -> -(-36+1) -> -(-35) -> 35

Relational operators:

It will return boolean either true (or) false, internally it makes use of or operator.

Assignment operator:

Single assignment

a = 10

chained assignment:

int a,b,c,d ;

chained assignment is not possible in method parameters

a = b = c = d =10; assignment is done from right to left , i.e first 10 to d and it is passed upto a.

Eg: Chained\_Assignment\_Eg1

// check documentation in code

compound assignment:

a +=10 => a = a+10

Eg: Assignment\_Operator

//go through the code

Eg: Assignment\_Operator\_Eg2

// Here j is being reassigned(compound assignment)

Unary operators:

If you are using any operators and it involves only one operand it is called unary operator

Eg: a = 10 here a is operand.

a+=20

a++

Eg: Assignment\_Operator\_Eg3

Eg: Assign

//

First R.H.S side is evaluated and then it is assigned to assignment operator example : x = 2

x \*= 2+3\*5

x = x\*25

x = 2\*25 => 50

number literal increment is not possible, leads to compile time error.

Example : 1+ ++x

Space should be given after first addition otherwise complier treats increment is for literal (1+++x) which leads to compile time error.

increment decrement, assignment operators are unary

Binary operators:

if you are using any operators and it involves more than one operand it is called binary operators

Eg: a+b

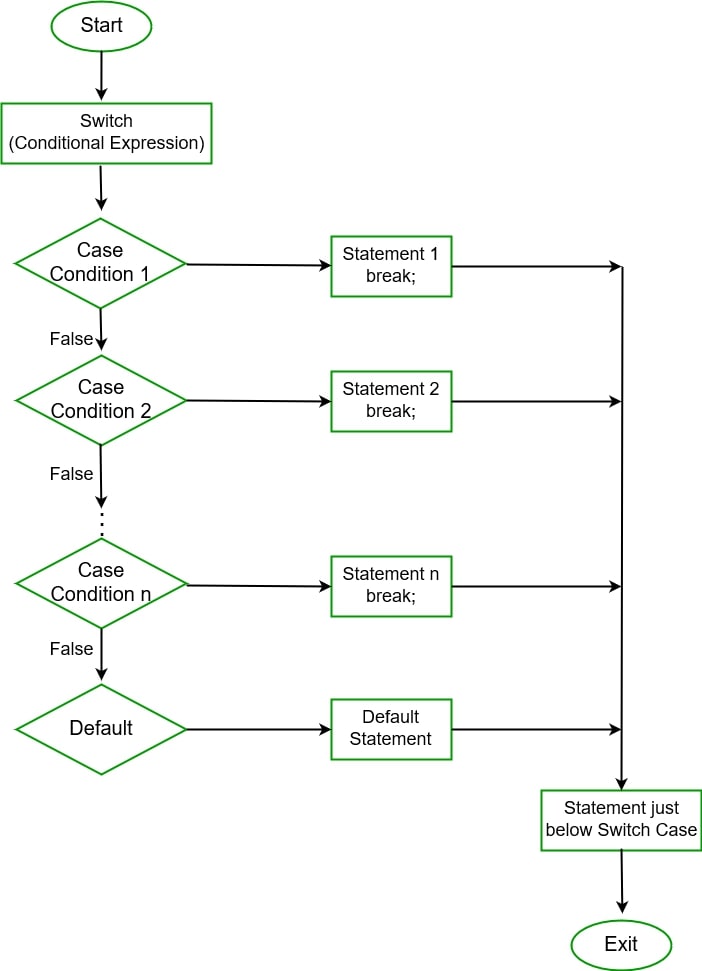
a-b

a>b

a==b

Switch cases

In switch case if certain case is matching it is executed and all the cases below the matching cases will be executed



To stop that use break

If all the cases are wrong you can use default and it is optional. If any case is matching and at last default is present It will not execute.

Eg: Switch\_Case\_Eg1

// go through the code

Eg: Switch\_Case\_Eg2

//statement is not a part of case label so compile time error

Eg: Switch\_Case\_Eg3

label in switch should be “compile time constants” the value should be known to compiler.

Eg: Switch\_Case\_Eg4

// go through the code

Eg: Switch\_Case\_Eg5

// go through the program

Eg: Switch \_Case\_Eg6

// label in switch should be “compile time constants” meaning the value should be known to compiler otherwise compile time error. (case(label):)

Label value should be in the range of the args type ( byte x)

Eg: Switch\_Case\_Eg7

//go through the code

Eg: Switch\_Case\_Eg8

1) label in switch statement should be “compile time constants”, meaning the value should be know to compiler otherwise C.E

2) Label value should be within the range of the switch argument otherwise C.E

3) duplicate case values are not allowed.

In the line 11 label value is a ‘literal’ not a operand which satisfies the first condition.

char value we can store in int

what compiler do is

int x = ‘a’ the corresponding value of the char is 97, where 97 is already a label value, so it is compile time error.

Note: we can store int data in char and char data in int without any typecasting.

Eg: Int\_Char

Eg: Switch\_Case\_Eg9

// Here the case label which is matching is executed and the remaining labels below the matching label will also be executed, since there is no break statement

Eg: Switch\_Case\_Eg10

// default can be placed anywhere in the switch, but it will be executed only when no case labels are matching.

If we place “->” in default, we cannot use “ : ” in other cases.

When tried with x =1, case labels, 1 & 2 are executed since there is no break in case 1 and 2 . All the cases will be executed after the matching case labels, as there is no break statement

When tried with x =2 , case label 2 is executed since there is no case labels beyond only it is executed.

When tried with x =3 , so there is no matching case labels default will be executed.

Eg: Switch\_Case\_Eg11

//go through the program

## Eg: Switch\_Case\_Eg12

//control goes to case 3 executes and since there is no break statement default also executes.

Eg: Switch\_Eg13

// Note :

Switch(arg) can only be of type

byte -> Byte

short -> Short

int -> Int

char -> Character

String

enum

Eg: Switch\_Eg14

Case sensitive mango is different from Mango

Eg: Switch\_Eg15

default can be placed anywhere inside switch. the control flow will be like this

1. All the cases will be evaluated first, if no cases are matching then default will be executed

Here default is written first, since no cases are matching, default statement is printed, and after default as there is no break statement, all the remaining cases also gets printed ,irrespective of their the case condition.

Note: after the matching case (correct condition case ) all the below remaining cases will also get executed if there is no break.

Eg: Switch\_Eg16

After matching case if there is no break and default is below the matching case, default will also get executed.

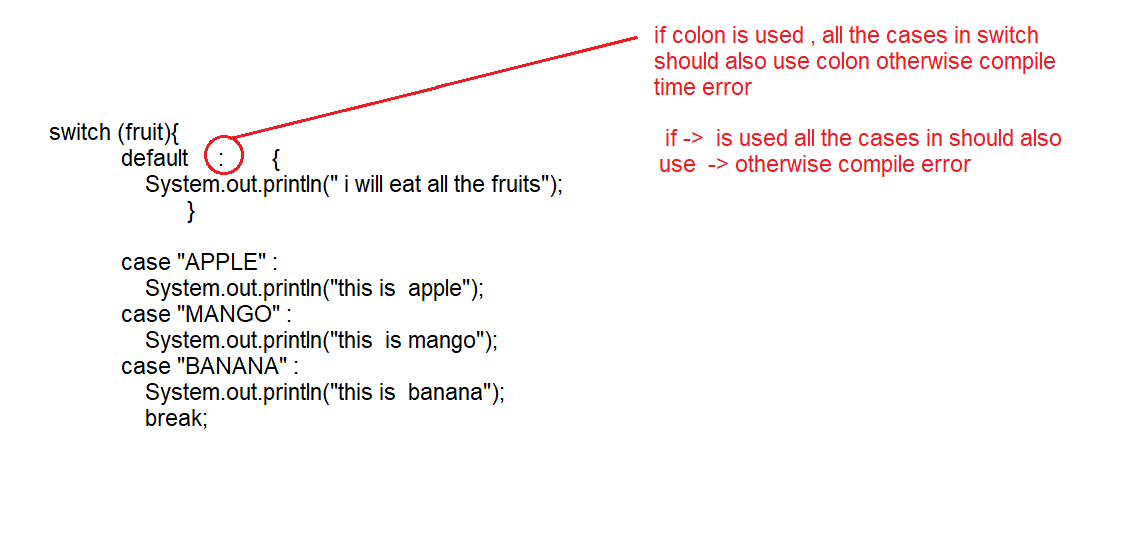
Eg: Switch\_Eg17

If any below case has break statement after the matching case, the execution stops /terminates at that break statement.

Eg: Switch\_Eg18

// go through the code

Eg: Switch\_Eg19



Eg: Switch\_Eg20

// check Eg13

Eg: Switch\_Eg21

// go through the code

default value of all the wrapper classes is null.

## Java Ternary Operator

## Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming.

## It is the only conditional operator which takes three operands.

Eg : Teranary\_Operator

Eg: Nested\_Ternary\_Operator

// In the above program first type checking is done. all of them are of int type, so there is no compilation error.

Eg: Ternary\_Operator\_Eg2

// if literals are involved in ternary operator compiler will directly do the computation before jvm.

Eg: Ternary\_Operator\_Eg3

// go through the code

Eg: Ternary\_Operator\_Eg4

// go through the code

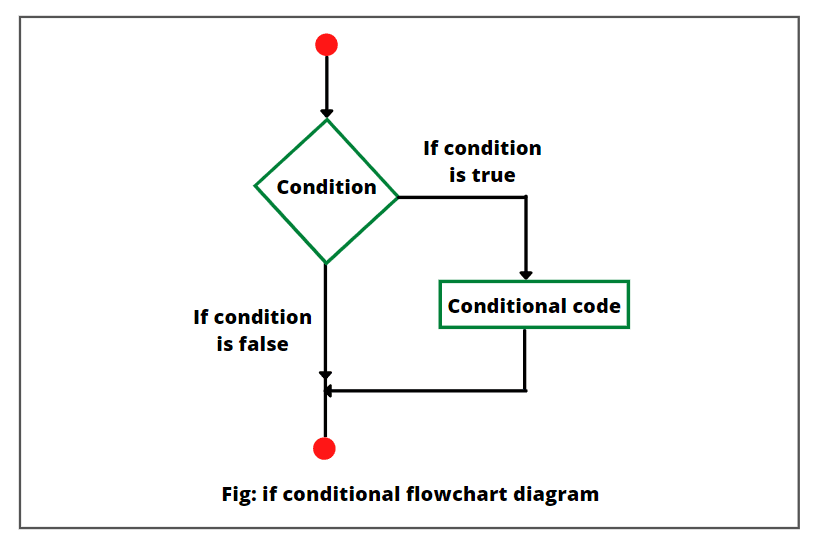
// expression result is of same data type, as of variable

There is no compilation error.

Eg: Ternary\_Operator\_Eg5

// go through the program

If Conditional statement:



Eg: If\_Else\_1

Eg: If\_Else\_If\_Ladder\_2

Eg: If\_Else\_3

// If works only on boolean, does not work for other datatypes. but int is given

Eg: If\_Else\_4

// If works only on boolean, does not work for other datatypes.

Eg: If\_Else\_5

//go through the code.

Eg: If\_Else\_6

// go through the code

Eg: If\_Else\_7

// here we are checking whether boolean false is equal to true are not.

Since false is not equal to true it moves to else block.

Eg: If\_Else\_8

//If there is one statement which needs to be part of if then { } braces can be omitted , but that statement should not be a declarative statement.

Eg: If\_Else\_9

// semicolon is also a valid java statement

Eg: If\_Else\_10

// go through the code

Eg: If\_Else\_11

// we can use declarative statement only if we use flower brackets, since we use flower brackets here it is valid

Eg: If\_Else\_12

// go through the code for documentation

Eg: If\_13

//go through the code

Eg: If\_14

// break can be used only in switch, loop and labeled block

If we try to use in other places it leads to compile time error

Eg: If\_15

//continue can be used only in the loops.

We can use them in labeled blocks but only if there are part of the loops. you cannot use in the switch.

Note : break statement moves the control out of the current working loop

continue skips the current iteration and further iterations work normally

Eg: If\_Else\_16

// concept of unreachability is only for loops (for, while, do-while) compiler will ignore unreachability for if else syntax.

Compiler will just give a warning, will not throw compile time error.

Eg: Label\_break

{  
} // block

Labelled block in java can be given as labelname:{

}

If we doesn’t use labeled block and used a normal block, When break statement is used , compiler doesn’t know which one to terminate , it leads to compile time error ,so we have used label block and mentioned label name after block , so that compiler can identify that it should terminate the block.

Eg: If\_Else\_Eg17

else block should always come immediately after the if block. no other declarations or statements, should be in between them.

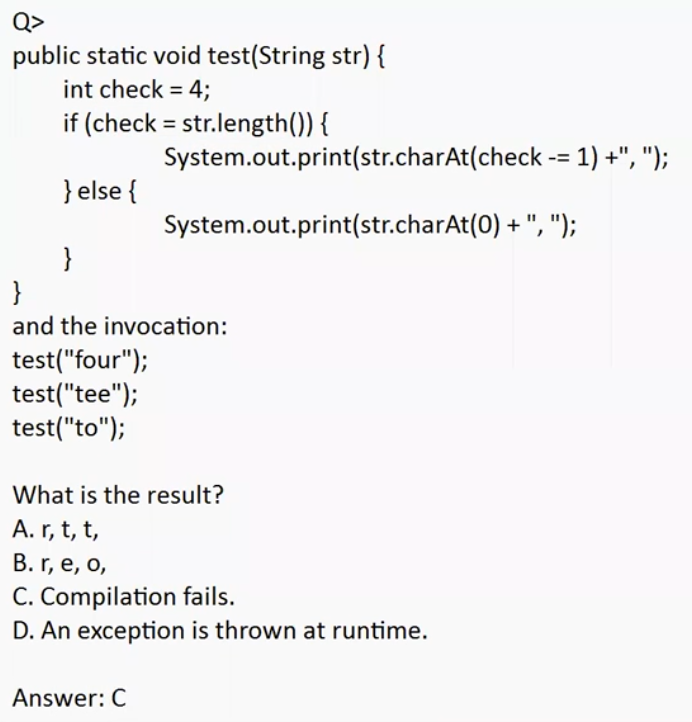
Eg: If\_Else\_Eg18

// go through the code

Note: unreachable won’t be checked by compiler for if and else logic

Eg: If\_Else\_Eg19

// go through the code



In the if the condition should be boolean.

Eg: Operators

// go through the program

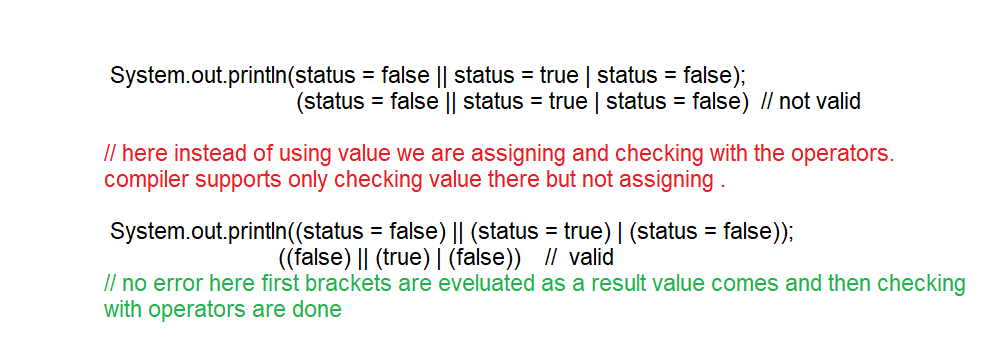
Eg: Operators\_Eg2

// go through the code

Eg: Operators\_Eg3

// go through the code

Eg: Operators\_Eg4



Eg: Operator\_Eg5

// go through the program

Eg: Operator\_Eg6

// go through the program

Eg: Operator\_Eg7

// go through the code