# JAVA 8

12/09/2017

## * JAVA-

Java is Simple, High level, platform independent, Architecture-Neutral, Secure, Robust, Multithreaded, Distributed and object oriented programming language.

## * Characteristics / Features / Buzzwords of Java-

### 1) Easily understandable-

The set of rules and regulations provided to develop java application is given in human understand-able language (i.e- English)

The drawback of C/C++ like pointer and operator overloading are not provided in java.

### 2) Securable Language-

Because of the components of java, (Java Virtual Machine) JVM, it is more secured language.

Java execution is protected by a cover like password protection for any account in JVM component. So, inside the JVM component only memory will allocate for Java application execution and without JVM Java application will not be executed.

JVM is mandatory for the execution of Java application.

### 3) Robust-

Strong in nature. Java is robust because of following functionalities-

a) Strong Memory Management
b) No pointer
c) Type checking mechanism / parallel mechanism.
d) Platform independent
e) Automatic memory cleaning process
f) Multitasking
g) Distributed.

13/09/2017

4) Architecture-Neutral-

On any type of CPU configuration, JAVA application will be executed.

5) Platform Independent-

Because of JVM and bytecode / magic value, Java application is platform independent.

WORA -> Write One Run Anywhere

6) Portable-

Bytecode files can be easily transfered from one machine to another machine.

7) Multitasking-

It is the process of performing multiple task at a time or simultaneously or concurrently.

To perform this multitasking java vendor has provided some predefined API (Library) (multithreading) and developers, no need to define any implementation.

8) Distributed-

It is the process of handling all the remote machines with the single administrator machine

Distributed technology.

RMI (Remote Method Invocation)

EJB (Entity Java Beans)

9) Object Oriented programming language-

By using JAVA language if we are developing any type of application then the task has to be divided into multiple task and it can be handled by multiple object of the class by using Object Oriented programming language.

Time and performance of the application are reverse process.

## * Programming Languages-

# Program :-
Set of instruction provided to perform any type of task.

# Language:-
Medium of communication (i.e system and user has to be connected)

# Programming languages are classified into three types:-
1) Low Level programming language.
2) Middle level programming language
3) High level programming language.

### 1) Low level programming language-
a) Source code is written in 0's and 1's.
b) No conversion is required
c) It is platform dependent
d) It is very complex to modify piece of code

### 2) Middle Level programming language-
a) Secure code - By using predefined words
i.e - ADD, SUB, MUL, DIV, MOV etc
b) Conversion is required to convert source code to machine code by using DeAssembler.
c) Processor dependent language
eg:- Assembly language - On which ever processor the application is developed, same is required for executing application.

3) High level programming language-

a) Source Code - Should be in english language.

b) By using compiler and interpreter we are converting source code to the machine language.

c) It is platform and processor independent.

d) It is easily understandable and modifiable.

# NOTE-

1) Java application must and should be executed by using JVM component, if there is no JVM on the machine then it will not be executed.
(Without JVM, java application cannot run)

2) In java application execution, first of all it is converting source code to byte code byte code will be converted to machine code.
The conversion of source code to byte code is known as compilation and conversion of byte code to machine code is known as execution.

3) For compilation process, java compiler is converter and for execution process java interpreter is converter.

4) Java source code has to be followed with .java extension and it is developed by developer and byte code will follow .class extension and it will be understandable by JVM which is generated in compilation process and the native code extension does not know and it will be generated in execution process.

5) In JAVA execution process, we can see only byte code file which is generated but we cannot see the native code file. So, we don't know the extension and where it is generating and what is its memory location. We don't know. So, the native code of JAVA application is invisible code.

* Components of JAVA Application-

=> JDK/SDK - Java Development kit / Software Development kit
=> JRE - Java Runtime environment
=> JVM - Java Virtual Machine
=> JAVAC.exe - Java Compiler - Compilation
=> JAVA.exe - Java Interpreter - Execution

Only Java application is platform independent and every component required for java application is platform dependent.

14/09/2017

# Javac.exe -

Number of lines of code will be read by the java compiler and verifies whether the code is syntatically correct or not. If there is any syntatical error then compiler will not convert to the byte code. (Compilation error) If there is no compilation error then it will convert into byte code (JVM)

Java compiler is implemented by using C/C++ languages. It is platform dependent due to C and C++ implementation is there.

# JAVA.exe -

The process of converting byte code to machine code is known as execution. To perform this process, java interpreter will involve.

Byte Code → Native Code

100 lines of byte code → Native Code

# JIT compiler - (Just-In-Time compiler) -

Byte code - Native code

How much necessary part of code is there, then it will convert into native code by checking the errors (JVM

# JDK / SDK -

It is a software/installer file which consists of

→ Java Development tools

→ SRC file

→ JRE

It is installed in the system to run java application

# JRE -

- It is providing some environment inside the system to execute the java application in the form of JVM.
- JRE is the implementation of JVM
- JVM is an instance of JRE

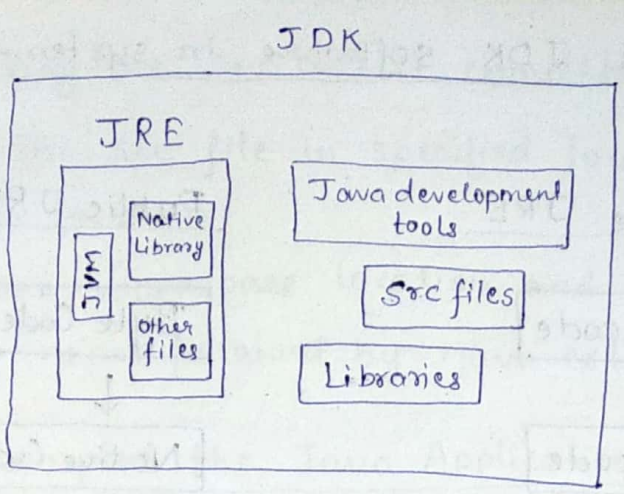
* Difference between JDK, JRE and JVM :-

- JDK is required as a java developer

src code → byte code → native code

- JRE is required as a java executer

byte code → Native code




=> When we install JDK software in file system then automatically JRE as well as JVM will be installed. Since JRE is a part of JDK, and JVM is a part of JRE

=> As a java developer when you are developing source code and performing execution then JDK is required, or if you are using the application which is already developed by some other user then we just need to convert byte code to machine code to perform execution. Further only JRE is enough so, we have JRE as seperate software and JDK as seperate software, i.e. JDK.exe and JRE.exe.

=> Every component of java application is implemented by using C/C++. So, they are platform dependent (Any application by using C/C++ then it is called as platform dependent application)

* HISTORY-

Editions-

SE - Standalone application (single system application) by using applet, swings.

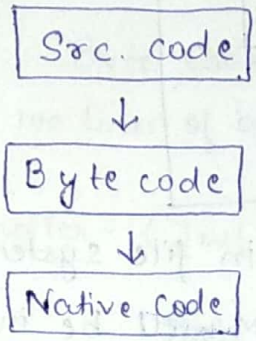
EE - Enterprise edition (client and server application) by using JDBC, servlets etc.

ME - Micro device edition (Smart phones, Remote, set up box)

ANDROID

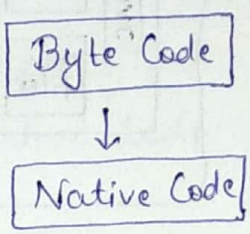
* How to install JDK software in system-

Private JRE




Any one JRE is enough either private JRE or public JRE

Public JRE




For using/running already developed application, public JRE is mandatory and we can't execute with private JRE.

15/09/2017

* How to develop the Java application-

# Requirement- Display some message on the screen/consol

"Welcome to Course Cube pvt. Ltd."

1) Take any text editor (i.e Notepad)

2) Java application is an OOPL so any java src code has to start the class definition.

```
class <class Name> { }
```

3) According to the requirement, for displaying the message on the console as it is, then java vendor has provided one statement.

```
System.out.println("msg");
```
→ After displaying the message the control will be transfered to the next line.

```
System.out.print("msg");
```
→ same line

4) Once writing the src code is completed then you need to save the src file in specified location called as workspace location.

5) Select some workspace location and save your src file with any name followed by .java extension.

## * How to compile the Java Application :-

→ Open system command prompt for execution of Java application.

→ The command used for compiling java application

javac <src filename.java>

javac Hai.java

# 'javac' is not recognized as an internal or external command, operatable program or batch file.

Above error will occures when system command prompt doesn't know the javac file location. So, we need to intimate about the location of javac file where it is located by setting the path.

set path = %path% <location of javac file> ;

set path = %path% E:\jdk1.8.0_78\bin

↳ It will add new path to the old path

set path = <location of javac file>

setpath = E:\jdk1.8.0_78\bin

↳ It will add new path without adding to the old path

→ Any java application starts the execution with standard method i.e main method.

public static void main (String args[])

While compiling, if there is any syntatical error then it will not convert into byte code. So byte code will be converted if there is no error and compilation is successful.

## * How to execute Java Application-

java.exe → Java interpreter is used for converting byte code to native code.

The command for using/executing java application

java <.classfile name without .class extension>

After compilation if the byte code is modified then JVM will verifies the byte code before converting to native code by using byte code verifier then if it is modified then following errors will be given.

ClassFormatError: Incompatible magic value

ClassFormatError: Truncated class file

## * To know the version of Java-

Compiler version → javac -version

JRE version → java -version

Compiling with lower version and executing in higher version is possible because lower version features are available in higher version features.

Compiling with higher version and executing in lower version is not possible because higher version features are not available in lower version features. It will throw error.

Unsupported class version error: Bad version number in class file

* Comment Line -

If we mark any statement with the comment line then that statement will not be executed.

Single line comments //

Multiple line comments /* to */

So, internally comment lines code is unnecessary part to execute the code.

WORKBOOK - PG-4

Q16. To execute java program without setting path.

To compile

Prompt > E:\jdk1.8.0_72\bin\javac Hai.java

location of java file

To execute

Prompt > E:\jdk1.8.0_72\bin\java Hello

Q17 - Yes it is possible to execute the java application without setting any path and without name on file.

Compilation -

javac .java → only one file is compiled.

javac *.java → Multiple java file will be compiled

Execution -

java .classfilename

Q16.

-> Compiling multiple source file can happen

-> We can only execute one program at a time.

When only JDK 7 is installed => higher version of java is installed

JDK 5 version is the lower version feature so java 5 features will be available in java 7 version.

| | | |
|---|---|---|
| compile | javac Hai.java | → java 7 compiler is used |
| compile | javac -source1.5 Hai.java | → java 5 compiler is used |
| execute | java classFileName | → java 8 / jre8 |

Compilation with lower version and execution with higher version is possible.

# NOTE-

Any higher version of jdk contains the features of lower version jdk features.

So, If higher version of jdk is installed, we can compile our application by using lower version features with -source option.

* JAVA Language Fundamentals- 18/09/2017

There are 10 fundamentals for developing a JAVA application.

1) Character Set
2) Keywords
3) Identifiers
4) Data types
5) Variables
6) Constants
7) Literals
8) Operators
9) Control Statement
10) Arrays.

1) Character Set -

There are 4 set of characters which can be used for developing the java application.

a) Digits => 0 to 9
b) Alphabets => A-Z or a-z
c) Symbols => All symbols (i.e * $ % # @ ! ( ) { } [ ]
d) White space character => \n, \t, \r, \f, \b (There will be no representation's for these characters.

| Digit/character | | Representation |
|---|---|---|
| 2 | → | 2 |
| A | → | A |
| * | → | * |
| \t | → | |
| \n | → | |

2) Keywords -

These are predefined words which are provided by java vendor and are also known as Reserved words because it cannot be modified.

Rules to implement keywords -

a) Any keyword cannot be used for classname, Variable name and Method name

b) All keywords are in lower case sensetive format.

c) True, False and Null are not a keyword but are reserved literals.

d) Upto JDK 1.4 there are 49 keywords but in JDK 1.5 a new keyword is added called enum. Now there are totally 50 keywords after JDK 1.5.

e) const and goto are unimplemented /unused keywords but these are not removed from the list of keywords.

3) Identifiers -

It is the set or specification of name which is provided by developer for the members of the java application. So, these are also known as user defined Identifiers.

When ever we need to use identifiers for the members of java application then we need to follow some rules.

a) Identifier should be the combination of digits, alphabets, and specific symbols (_ and $)

b) We should not start the identifier with digits.

c) Any keyword cannot be used as an identifier

d) Any reserved literals cannot be used as an identifiers.

e) By using case sensetive of keywords and reserved literals or by adding some other content, we can use as an Identifiers

| Keyword / Reserved word | Identifiers |
|---|---|
| Int | INT |
| char | charChoice |
| null | Null |

# Suggestions for using Identifiers-

a) While defining the class name as identifier, provide the starting letter of the word as capital and rest in small case.

ie
```
class Hello{};
class Test{}
class Student{}
```

b) While defining variable name, the total content should be either in lower case or upper case

```
int abc;
int XYZ;
int NOOFPERSON;
```

c) While defining the method name, we should provide the first word in lower case sensitive and from second word onwards, the starting letter should be upper case.

```
void showStudentValues(){}
void displayDetails(){}
void m1InCommunication(){}
```

# Other names to represent-

Keywords → Reserved words

Identifiers → User defined words

19/09/2017

## 4) Data types-

Whatever application we are developing, it will specified / allocate specified amount of memory.

jvm has the java memory allocation.

# Datatypes are used for 2 purposes.

a) To intimate about the type of input value

b) How much memory it is allocating for each input value.

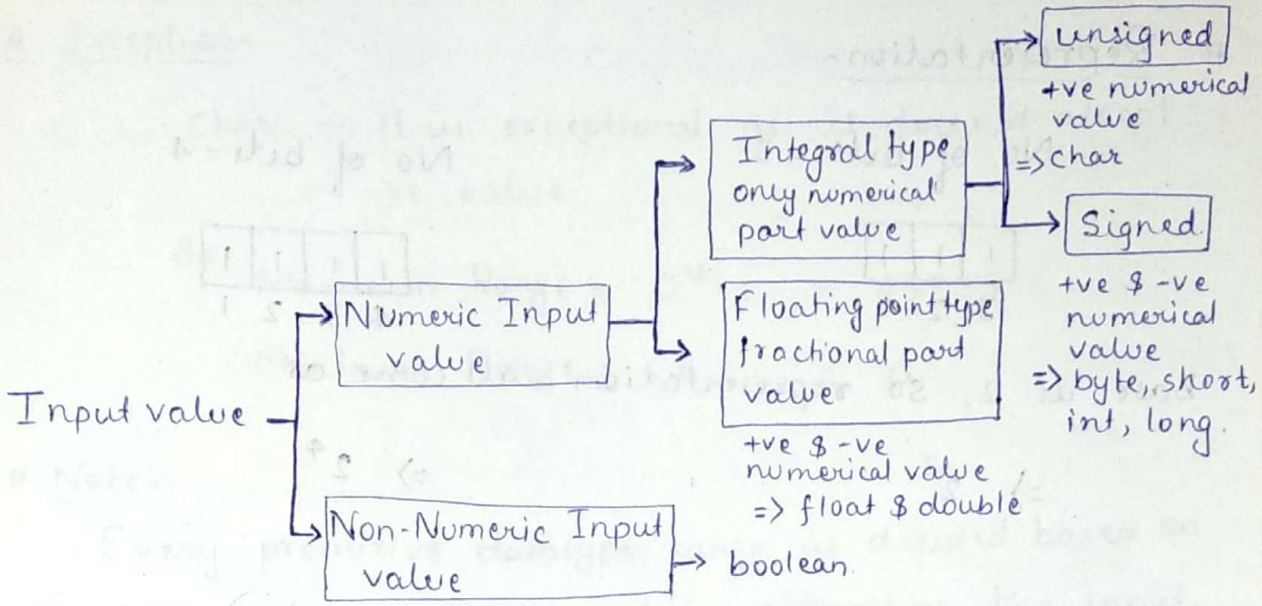
# NOTE-

As a developer, while developing any type of application we should estimate the memory of application,

When ever we need to estimate approximate memory location for an application, then we need to find out how many input values we are using for developing application and for each input value how much memory is required.

Then we can roughly estimate the memory required for running an application. So we need to find out each input value, how much memory is required and for the same the java vendor has provided the data type concept.

# The main need of datatypes is to estabise/estimate memory management.




# Classification of datatypes.

The datatyes are classified in two types.

a) Premitive datatype (Predefined)

b) User defined datatype (Reference)

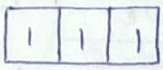
## a) Premitive datatypes -

Premitive datatypes are pre defined keywords whose implementation cannot be modified. Premitive datatype is provided for specific size, specific range and specific default value.

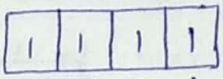
| Type | Size | Default value | Range |
|---|---|---|---|
| boolean | not defined | False | True / False |
| byte | 1 Bytes/8 Bits | 0 | $2^7-1$ to $-2^7$ |
| short | 2 Bytes/16 Bits | 0 | $2^{15}-1$ to $-2^{15}$ |
| int | 4 Bytes/32 Bits | 0 | $2^{31}-1$ to $-2^{31}$ |
| long | 8 Bytes/64 Bits | 0 | $2^{63}-1$ to $-2^{63}$ |
| char | 2 Bytes/16 Bits | 0 & \u0000 | $2^{16}-1$ to 0 |
| float | 4 Bytes/32 Bits | 0.0 | $2^{31}-1$ to $-2^{31}$ |
| double | 8 Bytes/64 Bits | 0.0 | $2^{63}-1$ to $-2^{63}$ |

# Representation-

No of bits = 3




No of bits = 4

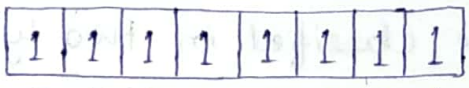



base is 2, so representation will come as

$\Rightarrow 2^3$

$\Rightarrow 2^4$

$\Rightarrow$ Byte = 1 Byte = 8 bits




$\Rightarrow 2^8$

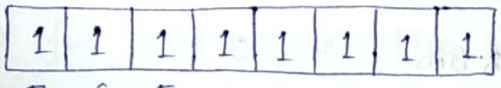
Since, considering a value 1000

$2^8$ - $1000 \times 2^1 = 2000$ → when its starting from 1 the initial value gets changed

$2^7$ - $1000 \times 2^0 = 1000$ → when its starting from 0 the initial value dosn't changes

So, that's why we take base power as 0 and not 1.

So, the bit representation will be as




So, insted of $2^8$, we are taking $2^7$.

as the last bit i.e signed bit is excluded bit so subtract it by 1.

So,

$2^8 \Rightarrow 2^7$

$\Rightarrow 2^7 - 1$

Maximum Range = $2^7 - 1 = (128-1) = 127$, Minimum Range = $(-2^7) = -128$

# Exception-

char -> it is exceptional as it does not accept -ve value.

So,

Maximum Range = $2^{16}-1$ = 65535

Minimum Range = 0 = 0

# Note-

Every premitive datatype range is decided based on numbers of bits, it is used for allocating the input value.

b) User defined datatype-

Upto JDK1.4, there are 2 data types which are used.

(i) Any class type
(ii) Any interface type

After JDK1.5, there are 4 datatypes which are used.

(i) Any class type
(ii) Any interface type
(iii) Any enum type
(iv) Any annotation

| Type | Size | Default value | Range |
|---|---|---|---|
| Any reference type | 8 Bytes/64 Bits | null | No specific Range. |

Memory allocation-

int 10 + int 20 = int 30

4B + 4B = 4B

So total memory allocation in 4B+4B+4B = 12 Bytes.

* Workbook Page 24 -

Q. Among the premetive datatypes short and char is allocating 2 Bytes of memory location (i.e same memory) then why the range's are different.

Ans. Short and char both are numeric data type but short is signed numeric datatype and char is unsigned numeric datatype.

Signed numeric datatype allows, both +ve & -ve values while unsigned numeric datatype allows only +ve numerical value and will not allow -ve value

This is the reason why short and char are of same size but different ranges.

# Requirement and size of memory calculation:-

Take 2 input value and perform addition, substraction division, multiplication and modular

declaration,

int 10 → [10] 4B
int 20 → [20] 4B
} 8 B

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 + 20 = 30 | → | [10] | + | [20] | = | [30] | 12B |
| 10 - 20 = -10 | → | [10] | - | [20] | = | [-10] | 12B |
| 10 / 20 = 0.5 | → | [10] | / | [20] | = | [0.5] | 12B |
| 10 * 20 = 200 | → | [10] | * | [20] | = | [200] | 12B |
| 10 % 20 = x | → | [10] | % | [20] | = | [x] | 12B |
| | | 4B | | 4B | | 4B | 60B |

So, here to memory usage is 60B (12x5)

This will cause unnecessary memory wastage So, to overcome this, we use variables a & b

## 5) Variables-

Variable acts like a storage container and is used for multiple time and reusability of same input value in the application.

Variables are used for storing any type of input value by specifying the datatype and a particular name.

Variable concept is given for reducing the memory wastage of same input value.

# To use variable we need to follow 2 steps.

a) Variable declaration

b) Variable initialisation.

a) * <modifier> <datatype> <variable_name>

b) <variable name> = <actual value>

ex:- int a;
a = 20;

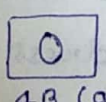
# Based on datatypes, variables are defined in 2 categories.

a) Premitive variable.

b) Reference variable

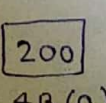
### Premitive variable

int a;




→ provided by jvm default value.

a = 200;

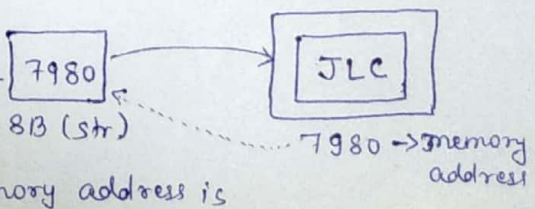



→ value assigned to a variable

### Reference variable

String str;


Str = "JLC";



Here 32 bit OS will have 32 bit format which is 4 Bytes

and 64 bit OS will have 64 bit format which is 8 Bytes

So, maximum memory allocation is provided for the Reference variable.

Q- Every premitive variable has different sizes of memory but why reference variable is required always 8 byte of memory?

Ans. Every premitive value/variable will assign the actual value with the allocated memory but in the case of reference variable, the actual content is assigned in different memory and wherever it is allocated, that allocated memory of reference variable is 8 byte of memory.

So, any type of reference variable will take memory address of either 32 bit or 64 bit based on the operating system, but the premitive variable will store which actual value, we dont know and actual value are allocated memory of premitive variable.

So, every premitive variable has different sizes and every reference variable will have same memory address since it is storing memory address but not actual value.

* Difference between doing declaration in one line and doing declaration in 2 lines.

20/09/17

int x;

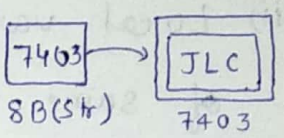

String str;


x = 99;


Str = "SRI";


int a=120;


String str="JLC";




Here, no defaults values are assigned, directly the actual values are allocated

# Based on Scope, variables are defined in 3 categories-

a) Instance variable-

Declaration of variable within the class defination without static modifier is known as Instance variable.

b) Static variable-

Declaration of variable within the class defination with static modifier is known as Static variable.

c) Local variable-

Declaration of variable inside the method of the class is known as Local variable.

```
class Hello {
    int x;              --> Instance variable
    static int y;       --> Static variable
    void show() {
        int c;          --> Local variable.
    }
}
```

# NOTES.

(i) Both instance and static variable can be accessed any where in the class definitation but local variable can be accessed only inside the method of the class

(ii) Local variable cannot be accessed in different methods of same class and instance and static variable cannot be accessed in another class defination.

(iii) Instance and static, if it is declared by not initializing then 'JVM' will provide the default values for those variables and local variables which are declared by not initialized, cannot be accessed. Since 'JVM' will not provide any default value for the local variables and before using local variable we need to initialize the actual value explicitely.

(iv) In case local variables are declared but not using then no need to initialize local variable.

(v) Restriction.

Instance variable cannot be accessed inside the standard main method because main method is declared as static.

# Various cases / syntaxes for using / declaring variables -

(i) Declaring the variable in 1 line.

Syntax => *<modifier> <datatype> <variable name>;

eg:-

```
int a;
char ch;
String str;
```

(ii) Initializing the variable with the actual value in next line.

Syntax => <variable name> = <value>;

eg:-

```
a = 36;
ch = 'A';
Str = "JLC";
```

(iii) Declaration and initialization of variable in same line.

Syntax => *<modifier> <datatype> <variablename> = <value>;

eg:-

```
int a = 99;
char ch = 'A';
String str = "JLC";
```

(iv) Declaration of multiple variable of same datatype in one line.

Syntax => *<modifier> <datatype> <var1>, <var2>, <var3> ... etc;

eg:-

```
int a, b, c;
char ch1, ch2, ch3;
String str1, Str2, Str3;
```

(v) Initialization of multiple variable of similar datatype in next line within the same value.

Syntax => <var1> = <var2> = <var3> = <var4> = 99

eg:-

```
int a, b, c, d;
a = b = c = d = 10;

Char ch1, ch2, ch3, ch4;
ch1 = ch2 = ch3 = ch4 = 'A';
```

(vi) Declaration and initialization of multiple variable of similar datatype in same line.

Syntax => *<modifier> <datatype> <var1> = val1 , <var2> = val2,
<var3> = val3 .... etc ;

eg:-

```
int a = 10, b = 20, c = 30, d = 40;
or  int a = 10, b = 10, c = 10, d = 10;
```

# Without declaration initialization will not happen.

NOTE-

(i) While declaring one variable, we need to initialize already declared variable with some specified value

eg:-

```
int a, b, c;
int d = c = b = a = 54;   -> valid

int d = c = b = a = 54;   -> invalid.
```

(ii) Duplicate variable declaration is not allowed in the same scope but it is allowed in different scope.

eg:-

```
int a, b, c;
int a=10, b=20, c=30;   } invalid

int a=10, b=20, c=30;   } valid.

int a=b, b=45;   } invalid because b is not declared $ initialized before initializing variable a
```

```
int a = a = 40;   } valid
```

only once variable a is allocating memory.

21/09/2017

## * Constant-

It is the fixed content of the variable that cannot be modified. In java, constant is also known as final variable.

```
int a = 80 ;        -> Not constant
final int a = 80 ;  -> Constant
```

If we use final, the the value cannot be modified.

eg:-

```
int a = 80;
System.out.println(a);
a = 20;
System.out.println(a);
a = 350;
System.out.println(a);
```

O/p will be:

```
80
20
350
```

When we use final, the same code will give error.

```
final int a = 80;
System.out.println(a);
a = 200;
System.out.println(a);
```

O/p will be some error.

=> Byte code display with final

```
System.out.println(120);
final int a = 50;
System.out.println(50);
```

Byte code display without final

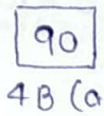
```
System.out.println(120);
int a = 50;
System.out.println(a);
```

# There are two ways of declaration.

a) Single line declaration

b) Two line declaration

int a=90; int b;




4B (a)


4B (b)

b=80;


4B (b)

final int x=99 final int y;


4B (x)

fixed

4B (y) → Blank final variable

y=77;

fixed ← 


4B (y)

In two line declarat
the value will not be
filled by the JVM
(i.e default value)
else it will be blank

## * Literals -

These are the actual value which we are initializing for the corresponding datatype based on some set of rules and regulations.

Literals are the constants and are used for any type of operations.

int x = 240;

datatype ← int

Identifier or (Variable) ← x

→ Literal or (Actual value) 240

int a = 10; Literal value of a = 10.

# Classification of Literals-

a) Boolean literals. → boolean
b) Character literals → char
c) String literals → string
d) Integral literals → byte, short, int, long
e) Floating literals → float and double
f) Null literals → reserved literals

## a) Boolean Literals-

These literals are provided for initializing boolean datatype variable and to provide which is valid literals and which is invalid literals.

boolean
(i) It is a Primitive datatype
(ii) It is a non-numeric datatype
(iii) Size → Not defined
(iv) Default → False
(v) Range → True/False

> Boolean is allowed for initializing either true or false which is considered as a reserved literals and that cannot be modified.
> Boolean datatype is a non-numeric datatype. So, internally. true and false means that machine will take it as 0 and 1. But we cannot initialized the boolean value with 0 and 1 explicitely.
> Internally jvm will convert true and false as 0 & 1
> So, the valid literals which we can initialize for boolean datatype is either true or false and the invalid literals are 1, 0, 1, 0, True, False, TRUE, FALSE etc.

b) Character Literals-

These literals are provided for initializing char datatype variable and to provide which is valid literal and invalid literal.

# char datatype variable can be initialized in 5 ways.

char
(i) unsigned numeric datatype which will not allow -ve numeric values and alphanumeric datatypes
(ii) Size → 2 Bytes (16 bits)
(iii) Default → ASCII (0) and UNICODE (\U0000)
(iv) Range → 0 to 65535

=> The 5 ways of initializing char datatype.

(i) Single character enclosed in single quotation marks.

```
char ch = 'Only one character';
char ch = 'A';
char ch = '5';
char ch = '*';
char ch = ' ';
char ch = '    ';
```

Valid terms.

```
char ch1 = 'ABC';
char ch2 = '';
char ch3 = ''';
Char ch4 = '\';
```

Invalid terms.

There are some characters which are enclosed in single quotes and it is not printing on the console like ', \ etc and returns error.

To overcome this we has a term called Escape sequence

(ii) Escape sequence character Literals-

Whenever we are printing some of the characters in single quotes, its not printing as it is on the console. So those characters can be printed as it is on the console by declaring with the help of escape sequence character literal (\)

```
char ch1= '';   -> empty character literal error
char ch2 = '\';  -> unclosed character literal error

char ch1 = '\'';  -> no error / valid
char ch2 = '\\';  -> no error / valid
```

valid escape sequence literals are

\n, \t, \r, \b, \f, \', \", \\, etc

for string " is required.

```
String str = "Welcome to \" CAPGEMINI\" ";
System.out.println (Str);

char ch1= '\+';  -> error / Invalid
char ch2 = '\p';  -> error / Invalid
```

Only specified escape sequence literals are valid not all or any.

iii) ASCII Representation -

ASCII representation is an unique numerical value which is assigned for each and every character literal.

There is an equivalent numerical representation which is used for each and every character literal

So, by using ASCII value of character, we can assign any type of character into the numerical datatype variable

ASCII defined range → 0 to 255

ASCII undefined character literals provided by JVM → 255 to 65535 (?)

eg:-

```
System.out.print("10");
char ch = 246;
System.out.print(ch);
System.out.print("2 = 5");
```

O/P 10 ÷ 2 = 5 → here the ASCII value of 246 represents ÷ symbol

Similarly $(a+b)^2$ can also be printed using ASCII

# ASCII representation can be used for 3 purposes-

- Whenever we are assigning character datatype variable to the numeric datatype variable, we require some numerical representation.

  The numerical representation can be given by using ASCII.

- Whenever we are performing any type of operation with a numerical value, it will easily perform with numerical content.

  But while performing operations with character values we need to use ASCII numerical values to perform any type of operation.

  '1' + '2' = 49 + 50 — when we use quotes, it acts as character hence takes ASCII value.

  = 99

  1 + 2 = 3

- Whenever we need to display some character which are not available in keywords, those characters are displayed by using ASCII representation.

eg:-

To display following representation, we need to use ASCII.

$10 \div 2 = 5$

$(a+b)^2$

```
class Lab {
    public static void main (string args []){
            System.out.print ("(a+b)");
            char ch1 = 178;
            System.out.println(ch1);
    }
}
```

O/P $(a+b)^2$

* Codes to display ASCII character.

```
class ASCII {
    public static void main (string arg[]) {
        for (int i=0 ; i<=255; i++) {
            char ch = (char)i ;
            System.out.println (ch + "\t" +i);
        }
    }
}
```

# Test code-

```
public static void main (string arg[]){
    char ch1 = 'A';
    char ch2 = 65;
    char ch3 = 'a';
    char ch4 = 97;
    int A = 99;
    int 65 = 99; -> invalid as variable starts with number.
```

# Limitations-

a) ASCII values can be used only in the initialization of the variable of char datatype and it is supported for only source code but not the byte code and native code.

b) ASCII value cannot be used anywhere in the application like for variable names, method names, class names, etc.

iv) UNICODE Representation-

To overcome the drawbacks of ASCII value, the UNICODE is introduced. It stands for Universal code Representation. It can be used anywhere in the application and it is given to overcome the limitation of ASCII values

=> Syntax

Starts with \u followed by 4 hexadecimal digits.

> Base of hexadecimal is 16

> Range of UNICODE character literal is not based on character range

| | | |
|---|---|---|
| 0 | to | 65535 ASCII |
| \u0000 | to | \uFFFF UNICODE |

| character | ASCII | UNICODE | OCTAL |
|---|---|---|---|
| A | 65 | \u0041 | \101 |
| a | 97 | \u0061 | \141 |
| 0 | 48 | \u0030 | \060 |
| | Decimal Base | Hexadecimal Base | Octal Base. |

Calculation =>

| 16 \| 65 |
|---|
| 16 \| 4 1 |
| 0 4 |

\u0041

| 8 \| 65 |
|---|
| 8 \| 8 1 |
| 8 \| 1 0 |
| 0 1 |

\101

| 16 \| 97 |
|---|
| 16 \| 6 1 |
| 0 6 |

\u0061

| 8 \| 97 |
|---|
| 8 \| 12 1 |
| 8 \| 1 4 |
| 0 1 |

\141

```
class Lab {
    public static void main (String args[]){
        char ch1 = '\u0041';
        char ch2 = '\u0061';
        char ch3 = '\u0030';
        System.out.println (ch1);
        System.out.println (ch2);
        System.out.println (ch3);
    }
}
```

O/P

A
a
0

```
class Lab {
    pu\u0062lic  st\u0061tic void main (String arg[]){
        ch\u0061r  ch1 = 99;
        System.out.println (ch1);
    }
}
```

# Note-

> Unicode character literals can be represented any where in application and it will be supported by source code or byte code or native code.

> whenever we are taking any unicode character literal, anywhere in the application, then compiler will replace with the actual character in the byte code conversion.

v) Octal Character Literals-

Syntax =>

Starts with \ followed by three octal digit.

Range

0 to 255

\0 to \377 → octal

```
char ch1 = 'A';
char ch2 = 65;
char ch3 = '\U0041';
char ch4 = '\101';
```

O/P

```
A
A
A
A
```

> It must and should be in single quotes.

> Whenever we are initializing the character literals with octal representation, then it should be enclosed in single quotation marks and it should be within the range.

# For the drawbacks of character literal, String Literal has been introduced.

c) String Literals-

These are provided for initializing the string datatypes variables which is an reference datatype and it will store in the content in different memory.

String is defined as collection of character enclosed within double quotation marks " " and it is known as character array.

# Syntax:

String str = " zero or more character";

String-

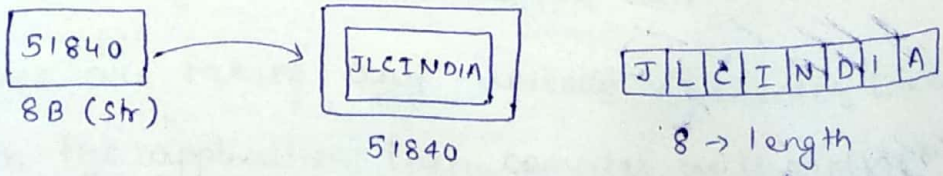
Size >> 8 bytes (64 bits)

default >> null

range >> No range

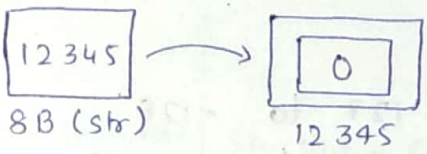
```
String str = "JLCINDIA";
System.out.println(str);
System.out.println(str.length()); → this line will give the count of characters i.e-8 for above Literal.
```

String str = "JLCINDIA";



# Condition -

String str = " ";

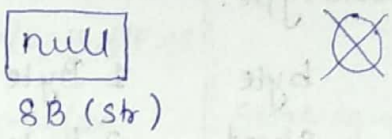



```
System.out.println (str)
System.out.println (str.length());
```

O/p

empty (nothing is displayed)
0 (count will be zero)

String str = null;




```
System.out.println (str);
System.out.println (str.length());
```

O/p

Null pointer exception error

here the allocation for length was not done as the null default value was assigned during declaration so, it will throw errow.

# We can use ASCII, UNICODE and OCTAL representation in string literals

```
class Lab {
    public static void main (String arg[]) {
        String str1 = "ASCII of A is 65";
        String str2 = "UNICODE of A is \\U0041";
        String str3 = "OCTAL of A is \\101";

        System.out.println (str1);
        System.out.println (str2);
        System.out.println (str3);
    }
}
```

d) Integral Literals. 25/09/2017

These literals has been provided for initializing the numeric data type. i.e only integral datatype.

| | | | |
|---|---|---|---|
| byte | 1 Byte | 0 | 127 to -128 |
| Short | 2 Bytes | 0 | 32767 to -32768 |
| int | 4 Bytes | 0 | 2147483647 to 2147483648 |
| long | 8 Bytes | 0 | ..... to ..... |

# Rules for using integral literals.

(i) Any literals integral datatype which is allowed for only numeric values and will not take any alphabets, symbols and white space character

(ii) While initializing the numerical value to the corresponding datatype, we need to provide the value within the range

(iii) By default any numerical constant is taking 4B of memory and the default datatype is int datatype

(iv) While initializing long datatype variable, if we are taking value upto the range then we can initialize directly (integer range) but in any case long size is more than integer in bytes then to exceed the integer range of value while initializing, we need to add suffix L.

=> Integral literals are classified in 4 types.

1) Decimal literal
2) Octal literal
3) Hexadecimal literal
4) Binary literal (From Java 7 onwards)

| Decimal Literal | Octal Literal | Hexadecimal Literal | Binary Literal |
|---|---|---|---|
| Base/radix = 10 | Base/radix = 8 | Base/radix = 16 | Base/radix = 2 |
| Digits = 0 to 9 | Digits = 0 to 7 | Digits = 0 to 9 | Digits = 0 & 1 |
| | | a/A = 10, b/B = 11, c/C = 12, d/D = 13, e/E = 14, f/F = 15 | |
| Starting → no initial starting required | Starting → 0 | Starting → 0x/0X | Starting → 0B/0b |
| ex - 101 | ex - 0101 | ex → 0X101 | ex → 0B101 |
| Solution on base 10 | Solution on base 8 | Solution on base 16 | Solution on base 2 |
| int a = 101 (2 1 0) | int b = 0101 (2 1 0) | int c = 0X101 (2 1 0) | int d = 0B101 |
| => $1\times10^2+0\times10^1+1\times10^0$ | => $1\times8^2+0\times8^1+1\times8^0$ | => $1\times16^2+0\times16^1+1\times16^0$ | => $1\times2^2+0\times2^1+1\times2^0$ |
| => $10^2+0+1$ | => 64 + 0 + 1 | => 256 + 0 + 16 | => 4 + 0 + 1 |
| => 101 | => 65 | => 257 | => 5 |
| It is the actual value & within the range of integer | It is the actual value & within the range of integer. | It is the actual value & is within the range of integer | It is the actual value and is within the range of integer |

c) Floating_point Literals-

It is given for initializing fractional part of value which includes decimal point. These are used for initializing the numerical datatype, (i.e floating datatype)

ex:-

| | | | |
|---|---|---|---|
| float | 4 Byte | 0.0 | Infinite |
| double | 8 Byte | 0.0 | Infinite |

=> Floating_point literals can be initialized in 3 ways.

(i) Standard Notation

(ii) Exponential Notation or Scientific Notation.

(iii) Hexa decimal Notation (Java 5 onwards)

(i) Standard Notation-

By default any fractional part or value will be considered as the double datatype.

```
float f;
f = 547.2638f  -> when using float, suffix f is mandatory
double d;
d = 7481.3695  -> when using double, suffix d is optional.
                  This is also the default datatype
```

Float → calculates upto 7 decimal points.

Double → calculates upto 15 decimal points

(ii) Exponential Notation-

Exponential notation can be used for initializing float and double datatype to convert standard value as either higher value or lower value. Just for the representation to the user.

Q- How any long datatype can be stored in float datatype with respect of size.?

Ans. Long datatype is a numerical datatype, an integral datatype which cannot allow decimal values but float is a fractional part of datatype which is used for storing fractional part of values. So, long maximum value can be stored in float datatype by converting into fractional part of values by using exponential notation. So, Double and float datatypes can be represented by using exponential notation. But it will not affect the byte code or native code.

byte > short > int > long > float > double.

1B > 2B > 4B > 8B > 4B > 8B

Conversion using c

26/09/2017

```
float f = 129.8763e2F;
double d1 = 129.8763e+2;
double d2 = 12987.63e-2;
```

O/P

f = 12987.63
d1 = 12987.63
d2 = 129.8763

} There will be no loss of data.

# double d = 0X101.145; → It will give error because Hexadecimal literal is malformed literal.

double d = 0101.123; → Starting 0 value is ignored and content (101.123) is considered as decimal floating point literal.

# Note-

If we initialize the double and float datatype with the hexadecimal floating point literal then if starting 0 is ignored but the alphabet in the content cannot be valid in decimal floating point literals. So, it throws error as Malformed floating point literal by the compiler.

This error is upto Java 1.4 and was overcomed by introducing Hexa decimal Floating point Notation in Java 5 onwards

(iii) Hexadecimal floating point Notation- (Java 5 onwards)

To initialize floating datatype with the hexadecimal floating point Notation, we need to convert base 16 to base 10 by using p notation in the fractional part content.

p0, p1, p2, p3, p4, ..etc.

```
double d1 = 0X101 ;      → valid
double d2 = 0X101.123 ;  → invalid (Malformed literal)
double d3 = 0X101.123p0  → valid syntex
```

```
int x = 0X101;

   101
  210

=> 1x16^2 + 0x16^1 + 1x16^0
=> 256 + 0 + 1
=> 257
```

```
double d = 0X101.145p0;

   101.145
  210-1-2-3

=> (1x16^2 + 0x16^1 + 1x16^0 + 1x16^-1 + 4x16^-2 + 5x16^-3) * 2^0

=> (257 + 1/16 + 4/16^2 + 5/16^3) x 1

=> 257.07934.....
```

Similarly,

```
{
    double d1 = 0xa.0p0;
    double d2 = 0x a.0p1 ;
    double d3 = 0x a.0p2;
    System.out.println (d1);
    System.out.println (d2);
    System.out.println (d3);
}
```

```
0xa.0p0

  a.0
 10.0 * 2^0
 ---------
  0 .-1

=> 10x16^0 + 0x16^-1 * 2^0
=> 10.0 x1

0xa.0p1

=> 10 * 2^1 = 20.0

0xa.0p^2

=> 10 * 2^2 = 40.0
```

We have only 3 ways.

There is no octal method and no binary method.

```
int x = 0B010 ; → invalid
int x = 0B101 ; → valid.
```

f) null literals-

It is a reserved literal which cannot be modified and it is used as the default value for any type of reference variable.

null literal will represent that the reference variable is not allocating any actual memory if it is initialized with the null literal or value. So, it will assign null value in the allocated memory. i.e 8 Bytes

* Underscore in Numerical Literal- (Java 7 onwards)

Underscore is a specified character which is used for just representing the count of a number easily while initializing to the numerical datatype. But it will not affect the byte code and native code.

#Following rules to follow while using underscore

```
int a = 123_45_67_890 ;   -> valid
int a = 1234_ ;           -> invalid
int a = _1234 ;           -> invalid
```

## 8) Operators -

They are symbols which are provided to perform any type of operations.

```
        10 + 20 = 30
Operands ←┘ └→ Operators
```

10, 20 → Operands (Value required for performing operation)

\+ → Operator (symbol)

=> Based on number of operands required to perform the operation, Operators are classified into three types:-

a) Unary operator → only one operand required
b) Binary operator → two operands required
c) Ternary operator → three operands are required.

=> Based on operation or application requirements Operators are classified into 10 types.

(i) Arithmetic Operators
(ii) String Concatination Operators
(iii) Assignment operators
(iv) Increment and Decrement operator
(v) Relational operator
(vi) Logical operator
(vii) new Operator
(viii) instanceof operator
(ix) Conditional or Ternary operator
(x) Bitwise operator.

(i) Arithmetic Operators-

Arithmetic operators are used to form arithmetic expression to perform the arithmetic operation. The operands used in arithmetic expression can be of numeric type or char type. Result of arithmetic expression is always int or wider than int type.

=> It can be classified into two types.

a) Unary Arithmetic operator

b) Binary Arithmetic operator

Operands which can be used = Any numeric datatype {byte, short, int, long, float, double} and char datatype also.

Resultant will be = int or wider than int {int, long, float, double}

a) Unary Arithmetic operator (+, -) -

+ => Unary plus => It returns same operand value with same sign

- => Unary minus => It returns same operand value with opposite sign.

ex-

```
int a = 10;
int res1 = +a;
int res2 = -a;
System.out.println(res1);
System.out.println(res2);
```

O/p

10
-10

```
int a=20
int res1= +(+a);
int res2 = -(-a);
System.out.println(res1);
System.out.println(res2);
```

if ++a is used instead of +(+a) -> it acts as incremental

if we use byte instead of int we will get error.

```
int a = 20
int res1 = - +a;    -> valid
int res2 = +-a;     -> valid
```

In the case of unary Arithmetic operator, if we are applying multiple same unary operator to the single operand side by side, then it will be considered as an incremental and decremental operator.

So, when we are using multiple same unary operator for single operand side by side then we need to seperate either by using space character or parenthesis. But it is not required for different unary operator for single operand.

b) Binary Arithmetic operator-

To perform this operation, we require two operands. So, here two operands must and should be numeric datatype.

+ → Addition
- → Substraction
* → Multiplication
/ → Division
% → Modular

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

```
long res1 = 250 + 560L ;      → valid
double res2 = 'A' + 48.89 ;   → valid
float res3 = 345L + 12.34f ;  → valid

int x = 50;
byte b = x ;  → invalid, int cannot be stored in byte.

50;
System.out.println(b1);
final int x = 50;
  byte b = x;
  System.out.println(b);  → valid; as the final keyword is used which will fix the variable and then it can be stored in byte (upto the range of byte i.e 127)
```

(ii) Whenever we want to assign integer datatype into lower datatypes (i.e byte, short, char) then we need to declare integer variable that should be declared as final and initialize in the same line.

But the final variable content should be within the range. If it is out of range then we cannot initialize.

(iii) The final concept is applicable only till integer datatype. and not for wider datatype than int.

```
final byte b1 = 10;
final byte b2 = 20;
byte b3 = b1+b2;
```
} → valid.

```
System.out.println (13/3);      //int     => O/p = 4
System.out.println (13/3.0);    //double  => O/p = 4.333333333
System.out.println (13/3.0f);   //float   => O/p = 4.3333333
```

```
System.out.println (13/0) ;  //int    -> invalid -> Arithmetic exp. error
System.out.println (13/0.0); //double -> valid  -> Infinite range
System.out.println (13/0.0f); //float -> valid  -> Infinite range.
```

(↳ Finite range)

# Double and float can give + Infinity and - Infinity.

```
System.out.println( 13/0.0)    -> + Infinity
System.out.println(-13/0.0)    -> - Infinity
System.out.println ( 13/-0.0)  -> - Infinity
System.out.println (-13/-0.0)  -> + Infinity.

System.out.println (0/0.0) -> NaN -> Not a number.
```

# Note-

(i) While performing division operation with zero value, then if resultant datatype is integer then then it give the error (i.e Arithmetic exception.) because it is having finite range (i.e int). which allows the maximum value and it cannot take more than that maximum value and if we perform the division operation with zero then it will return fractional part of value and these values cannot be accept by integer datatype.

(ii) While performing division operation with zero, if resultant datatype is double and float, then both are Infinite range of datatype. Then in that case java vendor has provided 3 constance for float and double datatype. i.e + Infinity, - Infinity, and NaN.

(iv) Any numerical value/0.0 gives +Infinity and -Infinity depending on sign of the value.

0.0/0.0 → NaN

0/0.0 → NaN

0.0/0 → NaN

* On the basis of priority evaluation, Operators are of two type :

(a) Operator Associativity-

(b) Operator Precedence-

(a) Operator Associativity-

Among the multiple operator in the single expression, same priority of operators is there, then it will perform the evaluation from left to right.

(b) Operator Precedence-

Among the multiple operator in the single expression, different priority of operators is there, then it will perform the high priority operator first, and remaining will be done after that.

# When parenthesis is involved, first parenthesis will be evaluated then others.

```
int a = 10;
int b=20;
int c= 5;
int res1= a+b+c;  → Operator Associative (same priority)
int res2 = a+b*c;  → Operator Precedence (different priority)
           first evaluation
int res3 = (a+b)*c;  → Operator Precedence (different priority with parenthesis)
           first evaluation.
```

(ii) String Concatination Operator- (7) 28/09/2017

It is a binary operator. Two operators are required to perform operation.

opt1 + opt2

# (+) operator is used for 2 purposes-

(a) Case I -

If the operands of + operator or numeric data type then it will be considered as an arithmetic addition operator.

10+20 = 30 ; (Numeric datatype)

Result of Arithmetic addition operator is numeric datatype

(b) Case - II -

If both the operands of + operator or any one operand is string datatype, then it will be considered as a string concatination operators.

"10"+"20" = "1020"; → String

Result of string concatination operator is string datatype

"JLC" + 99 = "JLC99";

true + "INDIA" = "true INDIA";

er. int a = 10;
int b = 20;
System.out.println (a+b); O/p = 30.

```
int a = 10;
int b = 20;
System.out.println("Result is :"+a+b);  → Result is: 1020
System.out.println("Result is:"+(a+b)); → Result is: 30
System.out.println("a+b+"is the Result"); → 30 is the Result
System.out.println("Result is :"+a-b);  → error
System.out.println(a-b+"is the Result"); → valid → -20 is the Result
```

# String reference datatype will perform the evaluation only for (+) operator and it will not work for any other operator.

So, if we are taking string datatype for other operator symbol except + operator, JVM will throw bad operand datatype error.

```
System.out.println("Result is :"+(a-b));  → Result is: 23
System.out.println("Result is:"+a+(-b));  → valid → concatinate
System.out.println("Result is:"+a+-b);    → valid → concatinate
                                            Result is: 10-20
```

Q- Why Reference content is not saved in Premitive?

Ans.

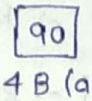
```
int a = 90; → valid
int x = a;  → valid
```

```
int a = 90;  → valid
String str = a; → invalid
```

In the case of string reference variable we cannot assign any premitive variable content because reference variable always require memory address but premitive variable has the actual content within the allocated memory.

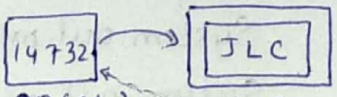
So, there is no chance of assigning premitive variable content to the string reference variable but there is a chance to assign any premitive variable to reference variable by using concatination operator.

int a = 90;



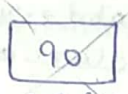

4B (a)

String str = "JLC";




8B (str) 14732

String str = a;




8B (str)

↓

not possible as it will store memory address not the content.

We can resolve this issue by using string concatination.

int a = 90;

String str = "" + a; → valid

empty string literal

String str = a + ""; → valid

# example-

```
int a = 35;
int b = 90;
int res = a+b;
System.out.println("Sum of 35 & 90 is:" + res); → valid
```

but when we will change the value of a & b, then still 35 and 90 is there, it should also get change. So, for the same we can use string concatination.

```
int a = 752;
int b = 85;
int res = a+b;
```

(i) System.out.println("Sum of" + a + "and" + b + "is:" + (a+b));

+ res

This can also be given using format specifies.

(iii) System.out.printf("Sum of %d and %d is: %d", a, b, res);

# The (i)st is not preferable as it uses lots of memory i.e memory wastage.

So, (ii)nd option can be used. instead.

(iii) Assignment Operators (=) -

It is a binary operator and it is used for assigning or initializing or storing the actual value to the corresponding operand datatype.

Syntax:-

```
Op1 = Op2 ;    Operands
```

# Rules for Assignment operators -

(a) Op1 must and should be a variable only.

(b) Op2 may be value of or any expression or any other variable, which should be compatible to the op1 datatype variable.

# Classification -

(a) Simple assignment

(b) Compound assignment.

```
int a;
a = 90;  -> valid
b = 70;  -> valid
```

```
int a;
a = ABC;  -> invalid
b = *^$$;  -> invalid
```

10 = 25; -> not valid, because operand should be variable not constant.

```
int a = 10 > 20;   -> invalid   (boolean type)
long L = 567.890;  -> invalid   (fractional type)
String str = 123;  -> invalid   (cannot take content directly)
```

but when,

```
boolean a = 10 > 20;  -> valid
double l = 567.890;   -> valid
int x = 123;          -> valid
```

## * TYPE CASTING-

It is the process of converting one datatype to another datatype.

destination variable = source value

# If destination and source are different so in that case we require conversion.

byte > short > int > long > float > double

char > int > long > float > double

# Whenever, while assigning source and destination datatype are not same, we need to provide the conversion.

# Conversion is not required when source and destination are same.

=> TYPE CASTING conversion is of two types-

(i) Implicit type casting → internal conversion

(ii) Explicit type casting → external conversion

### (i) Implicit typecasting-

This conversion is responsible by JVM.

```
byte b = 100;
int a=b;
```

} no syntax is introduced so JVM will do the conversion.

### (ii) Explicit type casting-

This conversion should be initiated by the developer

```
int x = 300 ;
byte b= (byte) x;
```

} here syntax is introduced (byte) so, o/p value will be converted to byte datatype instead of default value that is integer.

```
int x = 120;
byte b = x; → Invalid
```

```
int x = 120;
byte b = (byte) x; → valid
```

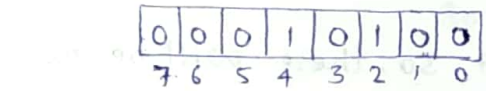
# Conversion

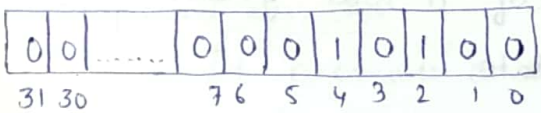
```
byte b = 20
int a = b;
```

| | |
|---|---|
| 2 | 20 |
| 2 | 10 0 |
| 2 | 5 0 |
| 2 | 2 1 |
| 2 | 1 0 |
| | 0 1 |

⇒ 10100

1 Byte = 8 bits.



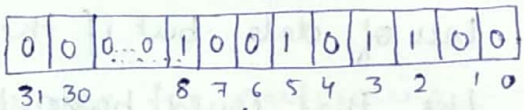

Since source bits are 8 and destination bits are 32. hence data can can stored without any data loss.

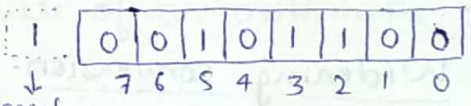
```
int a = 300;
byte b = (byte) a;
```

| | |
|---|---|
| 2 | 300 |
| 2 | 150 0 |
| 2 | 75 0 |
| 2 | 37 1 |
| 2 | 18 1 |
| 2 | 9 0 |
| 2 | 4 1 |
| 2 | 2 0 |
| 2 | 1 0 |
| | 0 1 |

⇒ 100101100

4 Byte = 32 bits.




loss of data

Since source bits are 32 and destination bits are 8 so, there will be some data loss.

# Calculation for data loss.

00101100

$= 0\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0$

$= 2^5 + 2^3 + 2^2$

$= 32 + 8 + 4$

$= 44$

Our value for a was 300

but the value which was stored is 44

So, it shows loss of data.

The loss will happen only when value is out of range of datatype

=> There are two types of conversion in type casting-

(i) Narrowing conversion

(ii) Widening conversion.

(i) Narrowing conversion-

It is the process of converting higher datatype to lower datatype.

32 bit  int a = 120;

8 bit  byte b = a; → invalid syntax (implicit)

so byte b = (byte) a; → valid (explicit)

as value of a is less than 127 so, there will be no loss of data. but if the value of a was greater than 127, the that could have loss of data.

(ii) Widening conversion-

It is the process of converting lower datatype to higher datatype

```
byte b = 50;
int a = b; → valid (implicit)
int a = (int) b; → valid (explicit)
```

There is no data loss.

"SAFE CONVERSION"

```
byte b1 = 10;
byte b2 = 20;
byte res = (byte)(b1+b2); → valid
byte res = (byte) b1+b2; → invalid.
```

boolean b = (boolean) 1 ; → not in compatable order

String str = (String) 120 ; → not in compatable order

int a = (int) "JLC";

# Instead of declaring datatype explicitely, we can use shortcuts

" Compound operator-

a* = 2 ; // a = (byte) (a*2)

int a = 10;
a* = 3+4; // a = a* 3+4 ← => here the answer will vary as the evaluation will happen from right to left. not lefttoright

So - result = 70
not = 34

* Compound operator-

It provides a shorter syntax for performing operations and assigning the results of an arithmetic or bit wise operator

| Compound assignment operators | Description |
|---|---|
| + = | Add right operand to left and assign result to left operand |
| - = | Subtract right operand to left and assign result to left operand |
| * = | Multiply right operand to left and assign result to left operand |
| / = | Divide right operand to left and assign result to left operand |
| % = | Calculate modulus using two operand and assign result to Left operand |

iv) Increment and Decrement Operator- (++,--) 03/10/20

Increment operator is used for increasing the value of the operand always by one value.

Decrement operator is used for decreasing the value of the operand always by one value.

++ operand
-- operand

# Increment and decrement operators are unary operator

ex- int a = 10 ;
a++ ; → valid

final int a = 10;
a++ ; → invalid, because final fixes the variable and makes it constant

int x = 15;
(x++)++ ; → invalid, because it is converted as constant first and second operation is not allowed.
(16) ++
(16)++

# Multiple increment operator for single operand is not allowed

=> It is classified into two types-

a) Prefix increment or decrement

b) Postfix increment or decrement

Prefix Syntax
<operator> <operand>

++ op
-- op

Postfix syntax
<Operand> <operator>

op ++
op --

## Prefix increment or decrement

First priority will go for evaluating increment / decrement operation and then it will perform other operation with the incremented/decremented value of operand.

```
int x = 15;
++x;
```

for ++x

x = x+1

increment before operation

```
int x = 15;
int y = ++x;
System.out.println(x);
System.out.println(y);
```

O/p

=> x = x+1 => 15+1 => 16 , x = 16

y = x => 16 , y = 16

=> int a = 10;

int res = a++ + ++a + a--;

So, O/p will be

a = 10,

a => 10 → 11 → 12 (a++) + 12 → 12 → 12 (++a) + 12 → 11 (a--)

res => 10 + 12 + 12 = 34

So,

a = 11

res = 34

# We cannot take 4 operand at one time.

## Postfix increment or decrement

First priority will go for ~~increment~~ operation and after the operation is done, the value will be incremented / decremented as per input.

```
int a = 15;
a++;
```

for a++

a = a+1

increment after operation

```
int a = 15;
int b = a++;
System.out.println(a);
System.out.println(b);
```

O/p

=> a = a+1 => 16 , a = 16

=> b = a; => 15 , a = 15

↳ this operation will be performed the increment.

```
int a = 10;
int b = a+++a;
System.out.println(a);
System.out.println(b);
```

So, O/p will be

a++ + a

b => 10 → 11 → 11

10 + 11 = 21

a => 11

So, a = 11

b = 21

```
int a = 10;
int b = ++a;
```

b = ++a → prefix
= a+1
b = 11
a = 11

first increment of a then operation, then assignment

```
int a=10
int b= a++
```

b = a++ postfix
b = 10
a = 11

first operation and assignm
then increment to a

# Postfix operation internally.

```
int a = 10;
b = a++;
```

```
{ int temp = a++
    a = temp
int temp = a++;
int temp = a      => temp = 10
a++ => a+1 = 10+1 => 11
a = temp
a = 10
```

hence o/p
a = 11
b = 10

# Prefix operation internally

```
int a = 10;
b = ++a;
```

```
{ int temp = ++a;
    a = temp;
int temp = ++a;
int temp = a 11
++a => a+1 = 10+1 => 11
a = temp
a = 11
```

hence o/p
a = 11
b = 11

# If

```
byte b = 127;
b++;
System.out.println(b)
```

o/p
-128

because when the data type value range is increased then it will perform cyclic action. i.e again starts from lower value for that datatype. i.e -128 onwards.

v) Relational Operators. (<, >, <=, >=, ==, !=)

These operators are binary operators.

By using these operators we can make as a relational expression.

=> Relational operator is classified into 2 types:-

a) Comparasion Operator. (<, >, <=, >=)

b) Equality Operator.

a) In comparasion operator any numeric datatype is allowed.

i.e. byte, short, int, long, float, double including char.

b) In equality operator any numeric datatype, boolean and reference datatype is allowed.

i.e byte, short, int, long, float, double including char

Resultant type for Relational operator is = true / false

i.e boolean type.

ex-

```
int a = 10;
int b = 20;
System.out.println(a>b);
System.out.println(a<b);
System.out.println(a>=b);
System.out.println(a<=b);
System.out.println(a==b);
System.out.println(a!=b);
```

O/p

```
False
True
False
True
False
True.
```

ex.

```
byte b = 65;
System.out.println('A'==b);
```

When comparing, the datatype is not necessary. When

'A' = char

b = byte

both are different datatype but still the comparasion will happen.

ex-

double d1 = 13/3.0f ; → valid statement

double d2 = 13/3.0 ; → valid statement

Both o/p should be equal but still it will return "false" while comparing because float will evaluate upto 7 decimal place and double will evaluate upto 15 decimal place.

Internal Conversion will happen using Type casting. So decimal value will get changed.

#Note-

When P1 = P2

1) Then in this comparasion, we are comparing two premitive variables p1 and p2 by using equals operator then internally JVM is performing comparasion. i.e it doesn't matter about the datatype. It will just compare the data.

2) Whether we are taking compatable datatypes or not. internally JVM is checking about its contents even though both operands are of same datatype of different datatypes and even in fractional part. If content is different then it is returning false or else it will return true.

```
System.out.println (true == false); → valid
System.out.println (true != false); → valid
```

but

```
System.out.println (true >= false); → invalid
System.out.println (true <= false); → invalid

System.out.println (true == 1); → invalid
System.out.println (false != 0); → invalid.
```

# Premitive Variable -

$P1 = P2$

Directly it will compare the actual content and it doesn't verify about the datatype.

# Reference Variable -

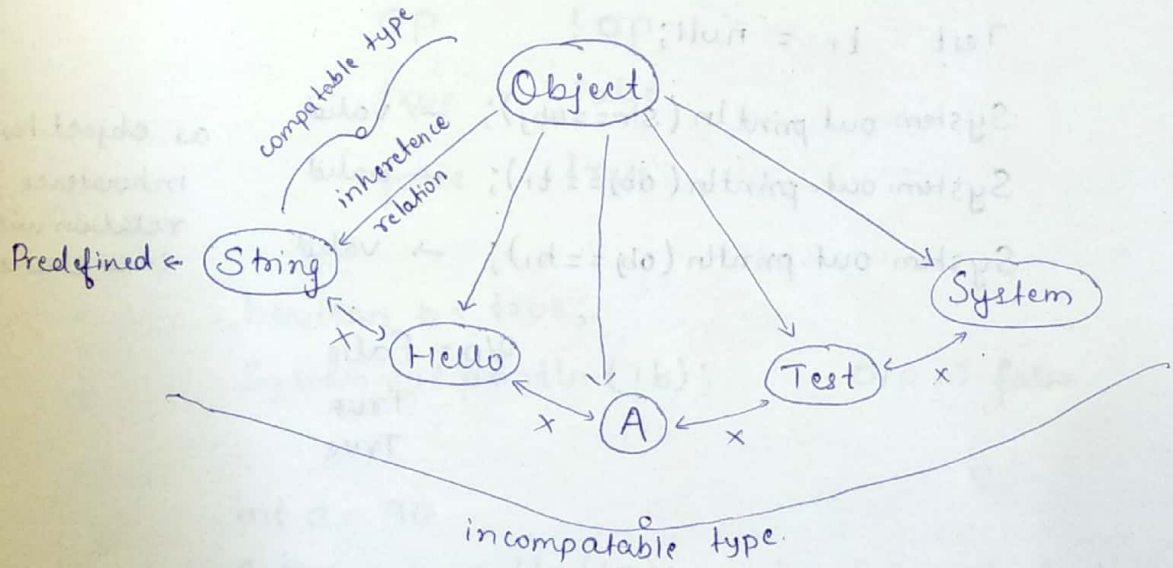
$R1 == R2$

=> For reference variable, it follows 3 points.

a) Firstly it verifies, both reference variable type are compatable type or not.

b) If its compatable type then internally it will compare the address of reference variable .ie same or not. It will never compare the actual value.

c) If they are incompatable types then compiler will throw error.




For every class, object is predefined class. ie called as inheretence relation.

ex.

```
{
    String str = "JLC";
    Hello h1 = null;
    Test t1 = null;
    System.out.println(str==h1);   -> invalid
    System.out.println(str==t1);   -> invalid
    System.out.println(t1==h1);    -> invalid
}
class Hello{};
class Test{};
```

the statement is incompatable

no inheritance relation b/w Classes.

So, for compatibility, we need to use object type.

```
{
    Object obj = null;
    String str = "JLC";
    Hello h1 = null;
    Test t1 = null;
    System.out.println(str==obj);   -> valid
    System.out.println(obj==t1);    -> valid
    System.out.println(obj==h1);    -> valid
}
```

as object has inheritance relation with each classes.

O/p - False
True
True

## vi) Logical Operator- 04/10/2017

These operators are used to perform logical operations.

=> Logical operators are classified into 3 types.

a) Logical NOT !

b) Logical AND &&

c) Logical OR ||

Operands => Only boolean datatype (true and false)

Resultants => Boolean datatypes (true and false)

### a) Logical NOT-

It is an unary operator It is used for reversing the value of operand.

| OP | !OP |
|---|---|
| true | false |
| false | true |

ex-

```
boolean b = true;
System.out.println(!b);        O/P => false
```

```
int a = 90
System.out.println(!a); -> invalid, only boolean is accepted.
```

```
System.out.println(!a==90); -> invalid
System.out.println(!(a==90)); -> valid    O/P => false
System.out.println(!(a!=90)); -> valid    O/P => true
```

b) Logical AND.

c) Logical OR

These two operands are used for combining two relational operators expression.

b) Logical AND- (&&)

It will return true when both operands values are true or else it will return false.

| op 1 | op 2 | op 1 && op 2 |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

ex-

```
int a = 88;
long b = 31903290L;
boolean res = a>50 && b == 31903290;
```

O/p => true

```
int a = 50;
boolean res = a>30 && a++<10;
System.out.println(res);
System.out.println(a);
```

O/p =>
res = false
a = 51

```
int a = 50;
boolean res = a<30 && a++>10;
System.out.println(res);
System.out.println(a);
```

→ Time consuming process.

O/p =>
res = false
a = 50

# here a=50 because first evaluation is false so no need to evaluate second. operand

c) Logical OR - (||)

It will return false when both the operands value are false or else it will return true.

| op1 | op2 | op1 \|\| op2 |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

ex-

```
int a = 10;
float f = 12.34f;
boolean res = a==10 || f>45;
System.out.println(res);
```

O/p => true

if `boolean res = a != 10 || f>45;` O/p => false

ex-

```
int a = 10;
boolean res = a>35 || a++<50;
System.out.println(res);
System.out.println(a);
```

O/p => true
a = 11

ex-

```
int a = 10;
boolean res = a<40 || a++<5;
System.out.println(res);
System.out.println(a);
```

O/p => true
a = 10

# here a=10 because first evaluation is true. so no need to do second evaluation. i.e operation will terminate at the point of true condition.

NOTE-

In the case of logical AND operator, if first operand value is false then no need to evaluate the second operand. so, it is a time consuming process.

In the case of logical OR operator, if first operand value is true then no need to evaluate the second operand, so it also called as time consuming process

So, java vendor has considered Logical AND and Logical OR as "SHORT CIRCUIT OPERATORS"

(vii) new Operator-

It is used for creating new memory location always whenever we specify new operator while using in the application.

It is mainly used in creating object of the class new Operator returns the memory address of memory.

```
Hello h1 = null;
Test  t1 = null;

        OR

Hello h1 = new Hello();   → it will assign
Test  t1 = new Test();      new memory
                            address
```

In java if we are using new operator then every time seperate memory location has to be allocated then in that case there is no chance that it will take already allocated memory location. Everytime, if we are using operator, for the same class also, a seperate memory location has to be allocated.

## viii) instanceof Operator-

By default, for each and every class, object class is the default parent class then we can assign any type of reference variable into the object class, object reference variable then in that case, which class it is belonging, we can not find out and if we want to verify manually, what is the last assignment then it is a time taking process. So, in that case, they have given instance of operator which is used for verifying any reference variable of specific class belongs to which class, then the instance of operator will return true. If it belongs to specified class or else it will return false if it doesn't belongs to specified class.

ex-

```
String str = "JLC";
Hello h1 = new Hello();
Test t1 = new Test();
Hai ref = new Hai();

Object obj = null;
  obj = str;
System.out.println(obj instanceof String);  -> valid
System.out.println(obj instanceof Hello);   -> valid
System.out.println(obj instanceof Test);    -> valid
System.out.println(obj instanceof Hai);     -> valid
System.out.println(obj instanceof Object);  -> valid
```

when, defined along side after obj = str ↓ the previous obj = str will collapse and new obj = h1 will take over.

```
obj = h1;
System.out.println(h1 instanceof Hello);  -> valid  i.e same class
System.out.println(h1 instanceof Object); -> valid
System.out.println(h1 instanceof Test);   -> invalid i.e class are diff.
```

Syntax-

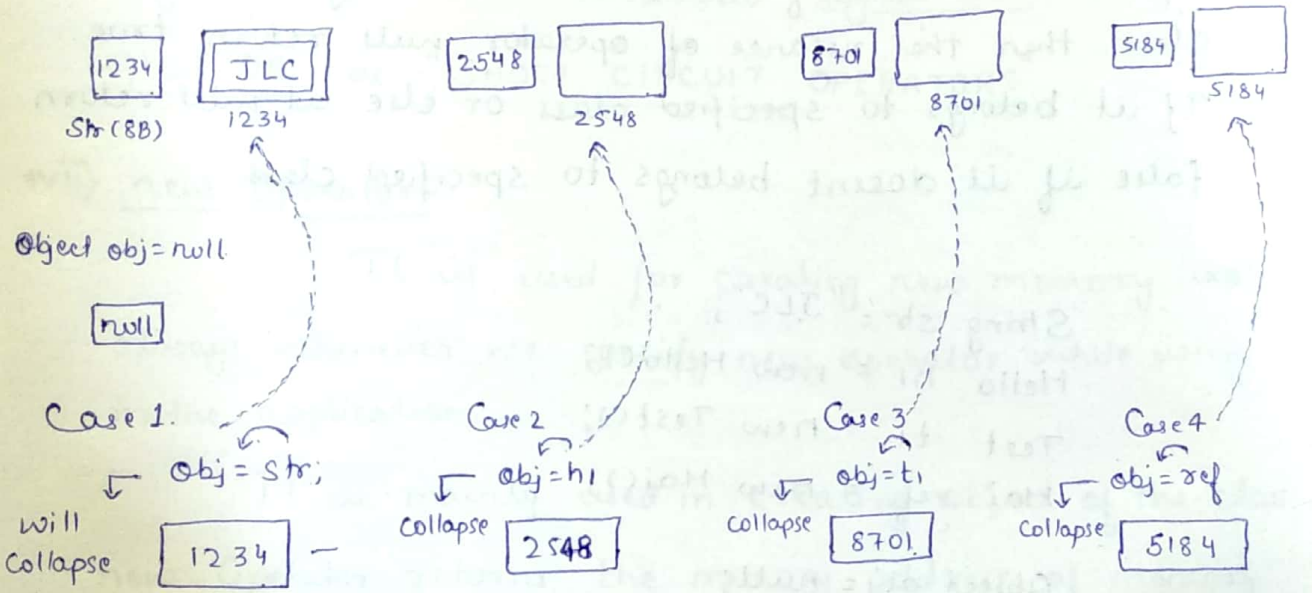
<any reference variable> <instance of className> ;

NOTE-

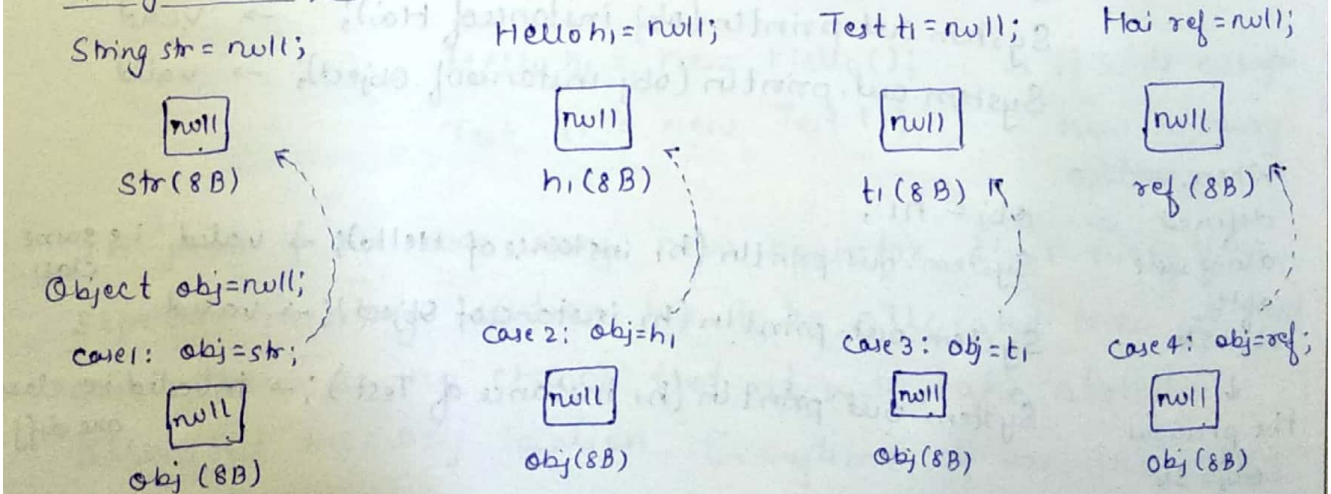
Reference variable classtype and className must and should be compatable.

# Assignment flow-

String str="JLC"  Hello h1 = new Hello();  Test t1 = new Test();  Hai ref= new Hai();




# Assignment for null




for any null, instanceof will return false. Even for object class it will return null.

while verifying any reference variable with instanceof operator, if it is null reference at that time no actual memory representation and no pointing to specified class. So, while refering null referencevariable with instanceof operator, it will return always false even for same class name also and object class also.

#NOTE-

It is also called as type comparasion operator.

ix) Conditional Operator / Ternary Operator-

Syntax.

```
op1 ? op2 : op3
```

#Rules-

op1 => must and should be boolean type

op2 & op3 => may be value, expression or variables which should be compatable to the resultant type.

true ? op2 : op3 => op2 value will be returned

false ? op2 : op3 => op3 value will be returned.

ex-

```
int res = true ? 10:20 ;    -> valid    O/P => 10
int res = false ? 10:20 ;   -> valid    O/P => 20

int a = 135;
int b = 87;
int max = a>b ? a:b;
int min = a<b ? a:b;
System.out.println ("Max value is:"+max);   -> O/P = 135
System.out.println ("Min value is:"+min);   -> O/P = 87
```

If we has 3 values then we should use Nested Conditional operator.

```
int a = 365;
int b= 96;
int c = 85;

int max = a>b?(a>c?a:c):(b>c?b:c);

System.out.println(max);
```

Upto 3 input value conditional operator is recomended or else its not recomended to use if more than 3.

*

X) Bitwise Operator-

They are used for performing the operation among the binary bits representation.

| | |
|---|---|
| NOT | ~ |
| AND | & |
| OR | \| |
| XOR | ^ |

1st Complement

↳ 00001010
=> 11110101 → Final value

2's complement

↳ 00001010
=> 11110101 → First to 1's complement
1 → After adding 1
11110110 → Final value

a) NOT - (~)

It is a unary operator

It is used for returning 1's complement of the operand value.

It will allow only numeric datatype operands.

ex-

```
int a = 10;
int res = ~a;
System.out.println(res)
```

10 → 00001010

1's → 11110101

If we have the first digit as 1 then first generate 2's complement

```
  00001010
         1
 ---------
- 00001011
```

$\Rightarrow 1\times2^3+0\times2^2+1\times2^1+1\times2^0$

$\Rightarrow -11$

Shortcut Notation-

~OP => -(OP+1)

~10 => -(10+1)

=> -11

* Difference between bitwise and logical-

| Bitwise | Logical |
|---|---|
| → Operand: Boolean and Numeric | → Operand: Boolean only |
| → Resultant: Boolean and Numeric | → Resultant: Boolean only. |
| → What ever op1 value is there either true or false, it must and should evaluate the op2 value | → In logical AND: if operand 1 is false then no need to evaluate operand 2 value.<br>→ In logical OR: if operand 1 is true then no need to evaluate operand 2 value. |
| → Bitwise operator are long-circuit operator | → Logical operator are short-circuit operator. |

TRUTH TABLE-

AND &

| op1 | op2 | op1 & op2 |
|---|---|---|
| T (1) | T (1) | T (1) |
| T (1) | F (0) | F (0) |
| F (0) | T (1) | F (0) |
| F (0) | F (0) | F (0) |

OR |

| op1 | op2 | op1 \| op2 |
|---|---|---|
| T (1) | T (1) | T (1) |
| T (1) | F (0) | T (1) |
| F (0) | T (1) | T (1) |
| F (0) | F (0) | F (0) |

Here the truth table is same as Logical OR and Logical AND

```
int a = 10;
boolean res = a<5 && a++ <30;
// boolean res = a<5 & a++ <30;
System.out.println(res);
System.out.println(a);
```

O/p for 1st => res = false
a = 10

for 2nd => res = false
a = 11

b) AND-(&)

ex- for numeric value evaluation

```
int a=10;
int b= 20;
```

AND

10 → 00001010
20 → 00010100
00000000

So, o/p => 0 . so it will return false.

OR

10 → 00001010
00010100
00011110 → $2^4+2^3+2^2+2^1$ => 30

So, o/p = 30 . so it will return true

c) XOR- (^)

| op1 | op2 | op1 ^ op2 |
|---|---|---|
| T (1) | T (1) | F (0) |
| T (1) | F (0) | T (1) |
| F (0) | T (1) | T (1) |
| F (0) | F (0) | F (0) |

* Shifting operator-

With the use of shifting operator the value can be doubled without using any Arithmetic operators.

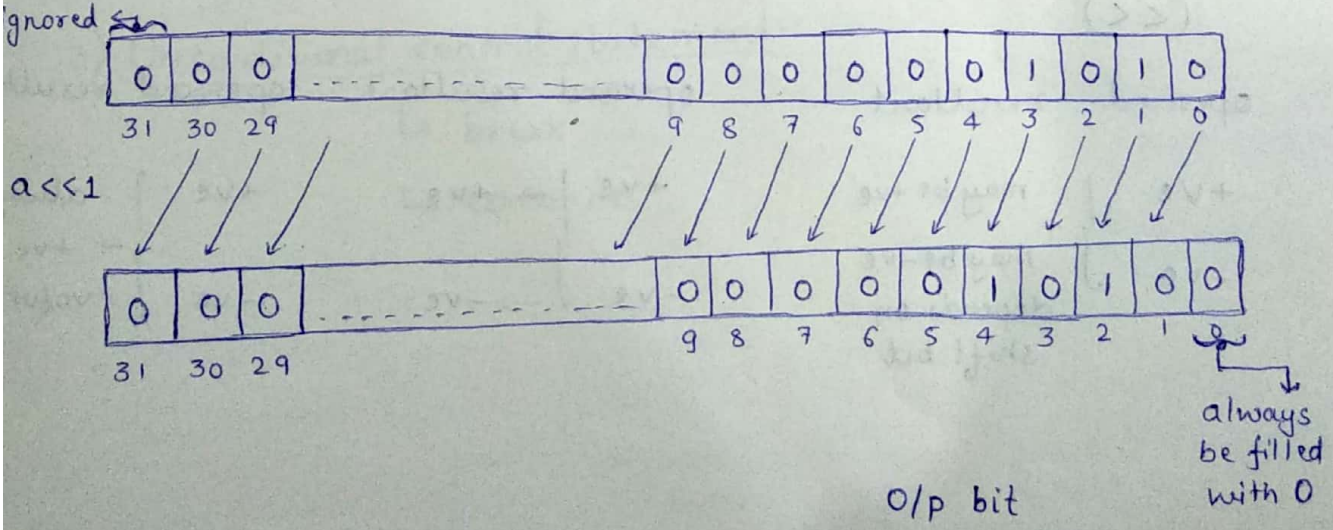
=> Types of shifting operator.

a) Left Shift operator.
b) Right shift operator
c) Unsigned right shift operator.

a) Left shift operator-

To shift the bits toward left (<<). It will double the value

ex. int a = 10; 10 → 00001010




O/p bit

10100 => 20

i/p bit

1010 => 10

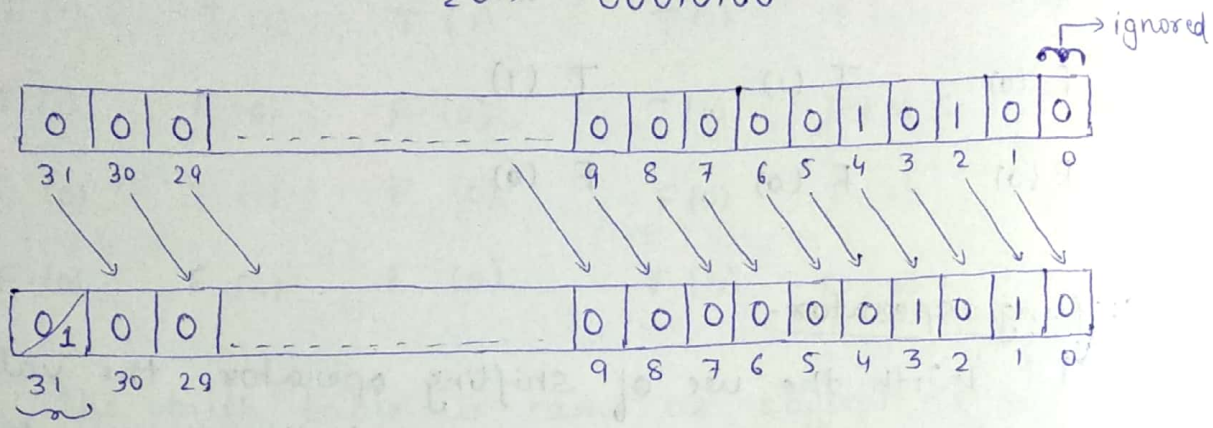
hence value was doubled.

b) Right Shift operator-

To shift the bits towards right (>>). It will half the value.

```
int a = 20;
```

20 → 00010100




if the i/p value is negative it will fill with 1 and if the i/p value is positive it will fill with 0

c) Unsigned right shift operator-

The last bit always fills with 0.

# Interview question-

| For Left Shift (<<) | | For Right Shift | | For unsigned | |
|---|---|---|---|---|---|
| operand | resultant | operand | resultant | operand | resulta |
| +ve | → may be +ve | +ve | → +ve | +ve | → always +ve |
| -ve | → may be -ve depends on shift bit | -ve | → -ve | -ve | value |

## * Control Statement-

Control statement are the set of statements which is used for controlling the flow of application as per the client requirement. Mainly these control statements will be used for

1) To reduce the lengthy source code
2) To reduce the time factor
3) Easy flow of application can be maintained

=> Control Statement is classified into 3 types-

1) Conditional control statement.
2) Looping control statement.
3) Unconditional control statement.

### 1) Conditional control statement-

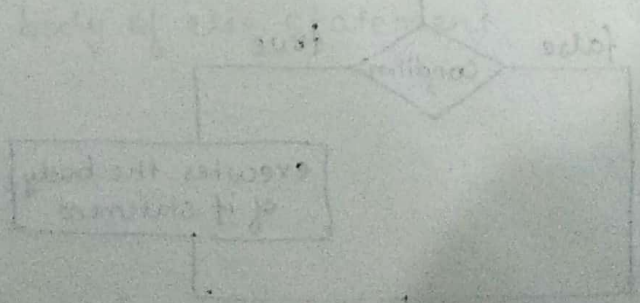
- if statement
- switch statement

### 2) Looping control statement-

- for loop
- enhanced for loop (java 5)
- while loop
- do-while loop.

### 3) Unconditional control statement-

- break
- continue



1) Conditional Control statement-

These are also known as decision making statement or selection statement. The control flow of conditional statement depends on the conditional statement.

Conditional expression can be variable or value or any constant or hardcoded value or any method etc. which should return the boolean value.

If boolean value is true then condition is satisfied or else if boolean value is false then the condition is not satisfied.

=> It is of 2 types-

a) if statement-

It is classified into 3 types.

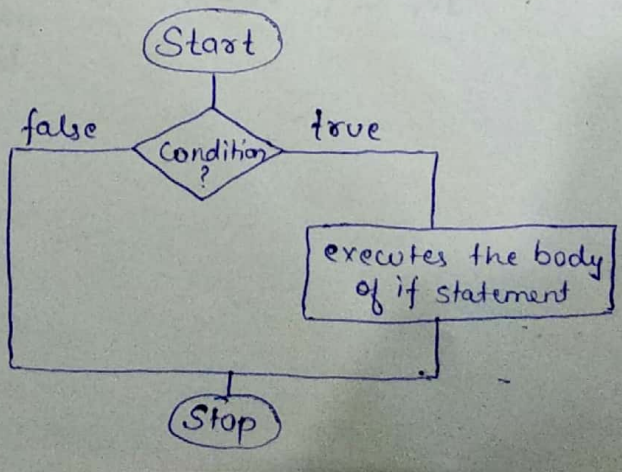
(i) Simple if statement
(ii) If-else statement
(iii) If-else if statement

(i) Simple if statement-

Syntax

```
if (conditional statement){
 // body of if statement
 }
```

Flow diagram.

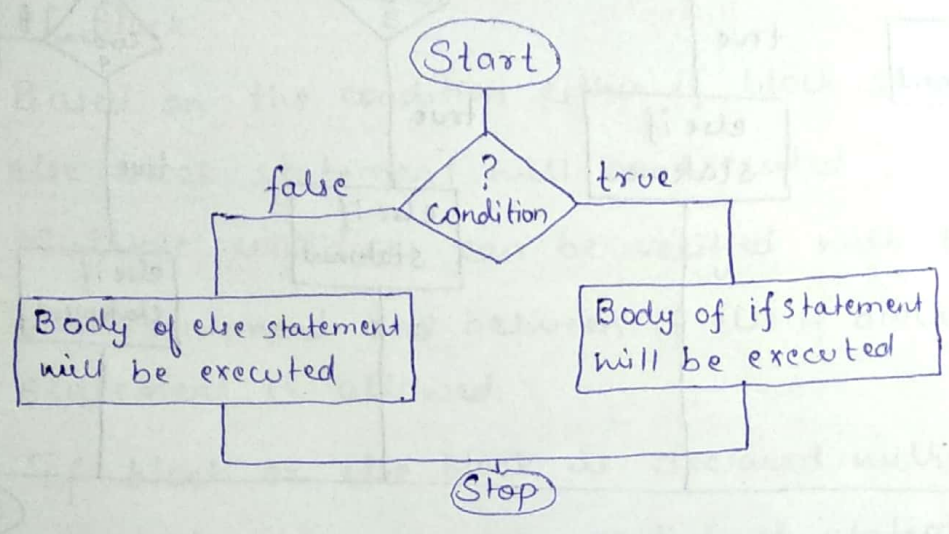


(ii) If-else statement-

Syntax-

```
if (conditional Statement) {
// body of if statement
} else {
// body of else statement
}
```

Flow diagram-

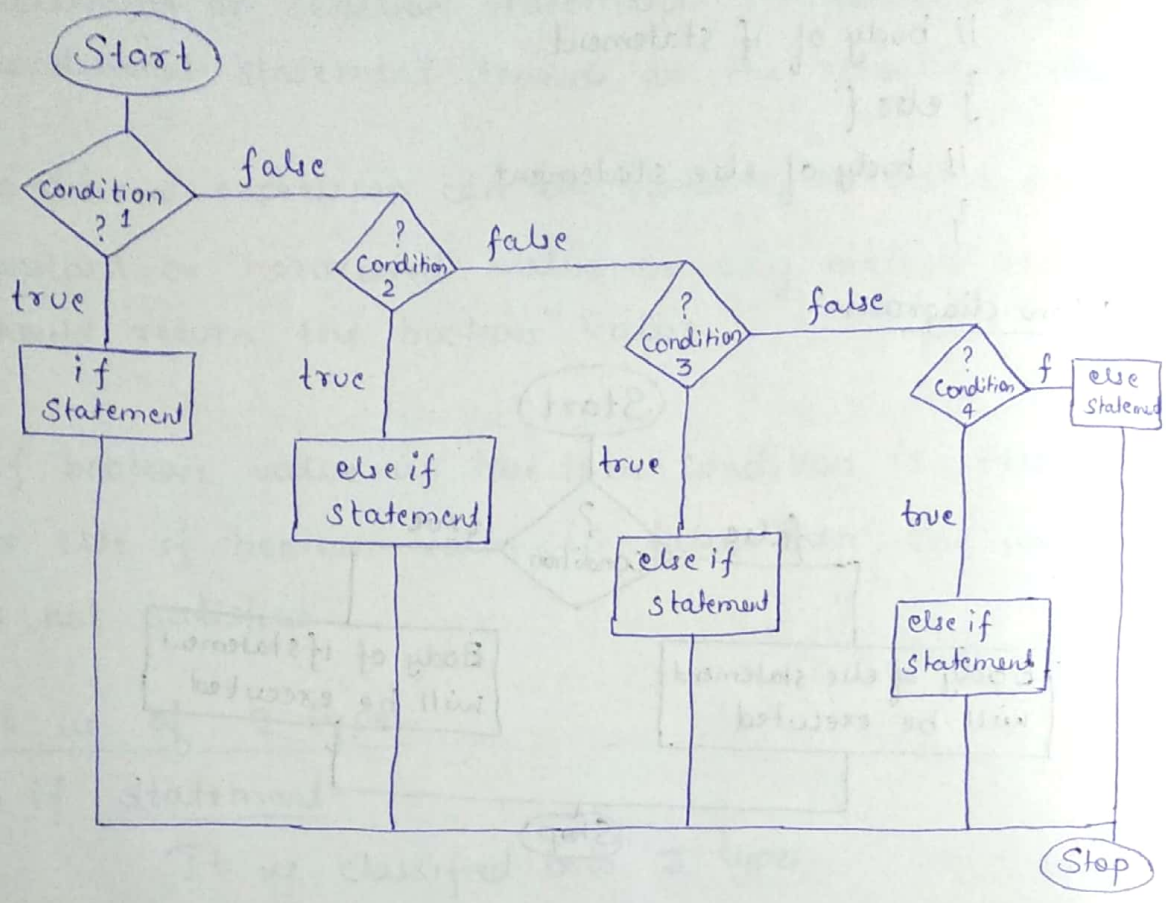



(iii) If-else if statement-

Syntax

```
if (conditional statement1) {
// body of if statement
} else if (CS2)
// body of if-else statement
} else if (CS3)
// body of if-else statement
} else if (CS4)
// body of if else statement
} else {
// body of else statement
}
```

## Flow diagram.




# NOTE.

Whenever the body if block and else block is having single statement then parenthesis {} is not mandatory.

```
if (true)
System.out.println("IF Block 1");   → if body statement without parenthesis
System.out.println("Hello Guys");  → normal o/p Statement
```

Only the first line after if statement is considered. not afterwards.

* Rules for defining if statement-

1) Statement must and should be declared with the conditional statement only.

2) Between if else statement no other statement is allowed. i.e println statement.

3) Else block should be followed with if block only. There is no chance to declare the else block without if block.

4) Based on the condition either if block statement or else block statement will be executed.

5) Multiple conditions can be verified with the if elseif block statement and between if else if block no other statement is allowed.

6) If block or else block is declared without any block of parenthesis then next first statement will be treated as the block of statement.

7) When if block or else block is declared without the block of parenthesis then next first statement should not be variable declaration statement.

i.e
```
if (true)
    int x=99;   -> Invalid.
```

8) Whenever we are declaring any variable inside the if block or else block, then block of parenthesis is mandatory.

i.e
```
if (true) {
}
```

Q- Write a java program to display your name, your college name, your email id, your date of birth, your throughout percentage, your mobile number using variable concept.

Code-

```
class PersonalDetails{
    public static void main (String arg[]){
    String Name = "Ankit Kumar";
    String clg = "Galgotia's collage of Engg & Tech";
    String email = " ankitkumargcet@gmail.com";
    String DOB = " 27-Feburary-1991";
    double per = 65;
    long mob = 9015099115L;
    System.out.println (" Name :"+ Name);
    System.out.println (" Collage Name :"+ clg);
    System.out.println (" Emaid ID:"+ email);
    System.out.println (" Date of Birth:"+ DOB);
    System.out.println (" Percentage :" + per);
    System.out.println (" Mobile No.:" + mob);
    }
}
```

Q- Write a java program to display days of a week based on input values starting from 0 to 6 as Sunday to Sat. For other than 0-6. Invalid option should display.

Code-

```
class DaysOfWeek {
    public static void main (String arg[]){
    int a=0;
    if (a==0){
    System.out.println ("SUNDAY");
    } else if (a==1){
    System.out.println ("MONDAY");
    } else if (a==2){
    System.out.println ("TUESDAY");
    } else if (a==3){
    System.out.println ("WEDNESDAY");
    } else if (a==4){
    System.out.println ("THURSDAY");
    } else if (a==5){
    System.out.println ("FRIDAY");
    } else if (a==6){
    System.out.println ("SATURDAY");
    } else {
    System.out.println("INVALID INPUT TYPE");
    }
    }
}
```

09/10/2017

## b) Switch Statement-

It is used for sequential block of statement execution continuously.

Syntax:

```
switch (choice value){
    case 1 : st1;
    case 2 : st1;
    case 3 : st1;
    case 4 : st1;
    case 5 : st1;
    default : statement;
}
```

* Rules for defining switch statement-

1) Switch choice value must and should be integer compatable datatype till JDK 1.4. (i.e byte, short, int, char)
2) From java5 onwards, we can specify enum type as the switch value.
3) From java7 onwards, we can specify string type as the switch value
4) Case label value must and should be constant values and it should be compatable to the switch value datatype
5) Duplicate case labels are not allowed.

ex-

```
Class Lab {
    public static void main (String arg[]){
        int x = 2;
        switch (x){
        case 0: System.out.println ("Sunday"); break;
        case 1 : System.out.println ("Monday"); break;
        case 2 : System.out.println ("Tuesday"); break;
        case 3 : System.out.println ("Wednesday"); break;
        case 4 : System.out.println ("Thursday"); break;
        case 5 : System.out.println ("Friday"); break;
```

```
        case 6 : System.out.println ("Saturday"); break;
        default : System.out.println ("Invalid input"); break;
      }
    }
  }
```

# If we need sequential block of information then break syntax can be removed from every line.

# Break statement terminates the execution.

```
int x=10;
switch (x) { }   → valid   → block of body is not necessary.

int x=10;
switch (x)       → invalid → block of parenthesis is mandatory.
```

# long, double and float are not not compatable type. but can be used by the help of typecasting.

```
class Test {
  // long l=67L;          →
  //   switch (l) { }     →  } invalid, but can be used by converting using typecasting
  // double d= 67.45;     →
  //   switch (d) { }     →
    float f = 67.453f;   →
      switch ((int)f) { } →  } valid, typecasting. switch((int)f) { }
  }
```

# enum declaration-

```
class Test {
    public static void main (String arg[]){
    color c = color.red;
    switch (c){
    case pink : System.out.println ("PINK"); break;
    case red : System.out.println ("RED"); break;
    case Black : System.out.println ("BLACK"); break;
    }
  }
}
enum color {
   pink, red, Black; }
```

## if else if

1) In if-else if, the sequential block of statement execution is not possible

2) In if-else-if, the condition is mandatory which return boolean value.

3) In this, the block of parenthesis is optional if body contains only one statement

4) In this, the body should contain atleast one statement

5) In this, the else block should declared always at the last.

6) In this, the condition is framed by using any operator (relational, logical etc)

## Switch

1) In switch, the sequential block of statement execution is possible

2) In switch, the parameter must be integral compatable data type or enum or string.

3) In this, the block of parenthesis is mandatory.

4) In this, we can declare with empty case labels

5) In this, the default case label can be provided anywhere (first, last or middle of case label)

6) In this, JVM internally takes the == (~~assig~~ comparator) operator to verify the switch value with the case label value

## * Looping Control Statement-

These are also known as 'Iteration statement or repetative statement and these are used for enclosing the piece of code to execute multiple times based on starting point and ending point values.

=> Types of Looping control statement.

a) for loop
b) enhanced for loop
c) while loop
d) do-while loop.

# Looping for loop statement flow is dependent on conditional statement only.

### a) for Loop-

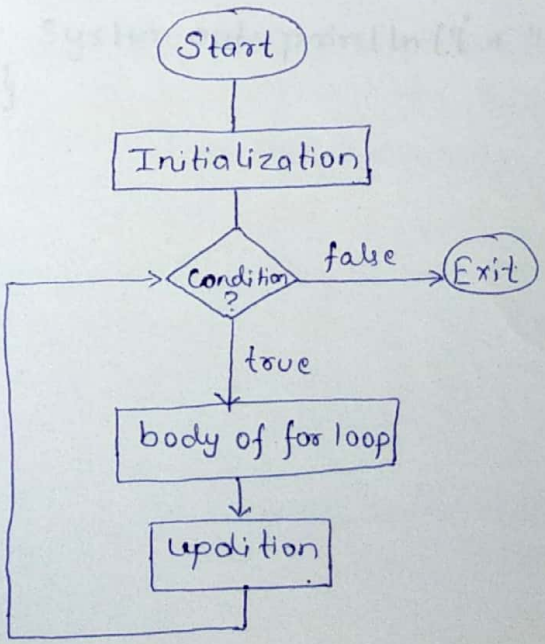
Syntax-

```
for (initialization ; condition ; updation) {
  // body of for loop statement
}
```

(1 → initialization; 2 → condition; 3 → body of for loop statement; 4 → updation)

=> in, for loop, initialization, condition and updation are optional.

=> Flow diagram-



```
for (int i=1 ; i<=5 ; i++){
    System.out.println(i);
}
```

condition checking.

for i = 1

1 <= 5 (T) → 1 → 1++ (2)

2 <= 5 (T) → 2 → 2++ (3)

3 <= 5 (T) → 3 → 3++ (4)

4 <= 5 (T) → 4 → 4++ (5)

5 <= 5 (T) → 5 → 5++ (6)

6 <= 5 (F) → Exit for loop

So, O/p will be

1
2
3
4
5

# To display even number between 1 to 100.

```
Class Test{
    public static void main(String arg[]){
        for (int i=1; i<=100; i++){
            if (i%2==0){
                System.out.println(i);
            }
        }
    }
}
```

O/p

2
4
6
8
.
.
.
96
98
100

# To represent the Table of 5-

```
class Test{
    public static void main (String arg[]){
        int n = 5;
        for (int i=1; i<=10; i++){
            System.out.println(n + "*" + i + "=" + n*i);
        }
    }
}
```

O/p

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
 :
5 * 10 = 50
```

# If a value is declared locally inside for loop then the increment will not happen as the same loop will do the same operation every time the loop occurs.

```
class Test{
    public static void main (String arg[]){
        for (int i=1; i<=5; i++){
            int a = 98;
            a++;
            System.out.println(i + "\t" + a);
        }
    }
}
```

O/p

```
1    99
2    99
3    99
4    99
5    99
```

# Infinite loop:-

```
class Test{
    public static void main(String arg[]){
    for (;;){
        System.out.println("Infinite Loop");
    }
  }
}
```

We cannot use ( ) only while using for loop. It has to be in syntax i.e (;;).

## * Unreachable Statement-

Inside the application whenever the scenario, if there is no chance to execute any piece of code, in case of infinite looping statement that also if condition is fixed or else if condition is not fixed then there is no error. So, in that case, if 100% gurantee is there that some statement will not be executed, then it will give unreachable statement as the compilation error.

```
class Test{
    public static void main (String arg[]){
        final boolean b=true;
        for (;b;){
            System.out.println ("Hello");
        }
        System.out.println ("Main Completed");
        }
    }
```

O/P

=> Unreachable statement error.

# Multiple variable declaration in for loop-

```
class Test {
    public static void main (String arg[]){
        for (int i=1, a=99, c=199; i<=5; i++, a++, c++){
            System.out.println(i+"\t"+a+"\t"+c+"\t");
        }
    }
}
```

O/p

| | | |
|---|---|---|
| 1 | 99 | 199 |
| 2 | 100 | 200 |
| 3 | 101 | 201 |
| 4 | 102 | 202 |
| 5 | 103 | 203 |

# Different datatypes are not allowed in same initialization statement and declaration statement. So, that has to be done seperately.

ex.

```
class Test {
    public static void main (String arg[]){
        int i;
        char ch;
        float f;
        for (i=1, ch='A'; f=12.34; i<=10; i++, ch++, f--){
            System.out.println(i+"\t"+ch+"\t"+f);
        }
    }
}
```

O/p

| | | |
|---|---|---|
| 1 | A | 12.34 |
| 2 | B | 11.34 |
| 3 | C | 10.34 |
| 4 | D | 9.34 |
| 5 | E | 8.34 |

ex.

```
class Test{
    public static void main(String arg[]){
      for(byte b=120; b>0; b++){
          System.out.println(b);
        }
    }
}
```

O/P

```
120
121
122
123
124
125
126
127
```

here, the datatype used is byte, so instead of going into infinite loop, it will go upto the datatype's maximum value (i.e 127) After 127, the datatype will start cyclic order, i.e (-128, -127, -126, .... till 0) so it will come out of looping statement.
The same will happen for integer datatype. (2147483647, -2147483648) and loop exists.

ex. printing pattern.

```
class Test{
    public static void main(String arg[]){
      for(int i=1; i<=5; i++){
          System.out.println("*");
        }
    }
}
```

O/P

```
*
*
*
*
*
```

# If we need pattern as below then we need to use nested for loop.

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

```
Class Test {
    public static void main (String arg[]){
        for (int i=1; i<=5; i++) {
            for (int j=1; j<=5; j++){
                System.out.print ("*");
            }
            System.out.println();
        }
    }
}
```

∴ Body -

rows (i.e i) = 1

| | | | | |
|---|---|---|---|---|
| 1 <= 5 (T) | col(i.e j) = 1 ; | 1 <= 5 ; (T) | 11 (1) | : j++ (2) |
| | | 2 <= 5 ; (T) | 1 2 | : j++ (3) |
| | | 3 <= 5 ; (T) | 1 3 | : j++ (4) |
| | | 4 <= 5 ; (T) | 1 4 | : j++ (5) |
| | | 5 <= 5 ; (T) | 1 5 | : j++ (6) |
| | | 6 <= 5 : (F) | Loop exists. | |
| 2 <= 5 (T) | col (j) = 1 ; | 1 <= 5 ; (T) | 2 1 | : j++ (2) |
| | | 2 <= 5 ; (T) | 2 2 | : j++ (3) |
| | | 3 <= 5 ; (T) | 2 3 | : j++ (4) |
| | | 4 <= 5 ; (T) | 2 4 | : j++ (5) |
| | | 5 <= 5 ; (T) | 2 5 | : j++ (6) |
| | | 6 <= 5 ; (F) | loop exists. | |

similar will happen till row 5 so the o/p will be as

| | | | | |
|---|---|---|---|---|
| 11 | 21 | 31 | 41 | 51 |
| 12 | 22 | 32 | 42 | 52 |
| 13 | 23 | 33 | 43 | 53 |
| 14 | 24 | 34 | 44 | 54 |
| 15 | 25 | 35 | 45 | 55 |

So, the code will be.

```
class Test{
    public static void main (String arg[]){
        for (int i=1; i<=5; i++){
            for (int j=1; j<=5; j++){
                System.out.print (i+""+j+" ");
            }
            System.out.println();
        }
    }
}
```

Similarly "*" pattern can also be printed.

```
class Test{
    public static void main (String arg[]){
        for (int i=1; i<=5; i++){
            for (int j=1; j<=5; j++){
                System.out.print ("*");
            }
            System.out.println();
        }
    }
}
```

this can be achived by replacing j<=5 as j<=i

O/p

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

```
*
* *
* * *
* * * *
* * * * *
```

Q- Write a program to check whether the number is even or odd.

```
class Test {
    public static void main (String arg[]){
     int num= 5;
      if (num %2 == 0){
          System.out.println ("The number is even");
        } else {
          System.out.println ("The number is odd");
        }
    }
}
```

b) Enhanced for loop:-

This is added from java 5 onwards. In this the loop control will be executed without initialization, condition and updation. Mainly the enhanced for loop is used for accessing the elements of Array or Collections object It is also known as "for each loop"

Syntax-

```
for (datatype variable : Array Name/Collection Name) {
  // body of statement;
  }
```

c) While Loop-

To over come the infinite looping issue, while loop is introduced.

Syntax-

```
initialization;
while (conditional expression){
    body of statement
}
```

# In while loop condition is mandatory.

=> Types in which the statements / codes can be written.

```
{
  int i = 1;
while (i<=5){
 S.o.p (i);
 i++;
}
```

```
{
 int i=1;
 while (i<=5){
  i++;
  S.o.p (i);
 }
```

```
{
 int i=1;
 while (i<=5){
  S.o.p (i++);
 }
```

```
{
 int i=1;
 while (i++ <=5){
  S.o.p (i);
 }
```

# Unreachable statement-

```
Class Test{
   public static void main (String arg[]){
      while (true){
        System.out.println ("Hello");
      }
      System.out.println ("Main Started");
   }
}
```

error. unreachable statement.

ex. nested while loop-

```
class Test{
    public static void main (String arg[]){
        int i=1;
        while (i<=5){
            int j=1;
            while (j<=5){
                System.out.print ("*"+" ");
                j++;
            }
        System.out.println ( );
        i++;
    }
  }
}
```

O/p

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

# for loop and while loop are known as "entry control looping statement" because if for the first time, the condition fails then it will not be executed.

d) Do-while loop-

It is also known as "exit control looping statement" because if the first time without verifying the condition the body of loop statement has to be executed and then only the condition will be verified. In case the condition fails, it will stop the execution or else the looping will continue.

Syntax-

```
initialization
  do {
     // body of statement;
       updation;
     }
  while (condition);
```

ex-

```
class Test{
   public static void main (String arg[]){
       int i=1;
       do {
         System.out.println(i);
       i++;
       }
       while (i<=5);
       }
   }
```

O/p

1
2
3
4
5

# Unreachable statement-

```
Class Test {
    public static void main (String arg[]){
        do {
            System.out.println ("Hello");
        }
        while (true);
        System.out.println ("Main Completed");
    }
}
```

O/p

error → unreachable statement

# infinite and finite looping.

ex-
```
Class Test{
    public static void main (String arg[]){
    do {
        System.out.println ("Hello");
    }
    while (true);
    }
}
```

O/p

Infinite loop when condition is true.

ex-
```
Class Test{
    public static void main (String arg[]){
    do {
        System.out.println ("Hello");
    }
    while (false);
    }
}
```

O/p

Hello

One time execution will happen either the condition is true or is false. Parenthesis is not mandatory.

3) Unconditional Control Statement- 11/10/2017

These are also known as transfer statements. These are used whenever we need to control the flow of the application without any condition.

=> Types of unconditional control statement-

a) break

b) continue

a) break-

It is a predefined keyword which is used to terminate the current block or loop of statement. break keyword can be used within any looping control statement or switch control statement.

eg

```
class Test{
    public static void main (String arg[]){
        for(;;){
            System.out.println("Hello");
            break;
            }
            System.out.println("Main Started");
        }
    }
```

ex-

```
{ int a= 10;
  if (a==10){
  break;
  System.out.println("Main Completed");
  }
```

→ invalid (can be used only on looping statement or switch statement)

ex-

```
for (int i=1; i<=10; i++){
System.out.println(i);
if (i==5)
break;
}
```

→ Valid (As per req")

ex-

```
Class Test {
    public static void main (String arg[]){
    for (int i= 1; i<= 3; i++) {
            for(int j= 1; j<=3;j++){
                System.out.println (i+"\t"+j);
        if (i==2)
        break;
        }
    }
}
}
```

O/P

```
1   1
1   2
1   3
2   1
3   1
3   2
3   3
```

ex-

```
Class Test {
    public static void main (String arg[]){
    for (int i=1; i<= 3; i++){
            for (int j=1; j<=3; j++) {
                System.out.println (i+"\t"+j);
            }
        if (i==2)
        break;
        }
    }
}
```

O/P

```
1   1
1   2
1   3
2   1
2   2
2   3
```

=> break is classified into two types-

a) break without label

b) break with label

a) break without label-

Syntax:

```
break;
```

It is used to terminate the current loop of statement.

ex-

```
Class Test {
    public static void main (String arg[]){
        for (int i=1; i<=3; i++){
            for (int j=1; j<=3; j++){
                System.out.println(i+"\t"+j);
                break;
            }
        }
    }
}
```

# NOTE-

break label without can be used outside the switch and loop statement.

b) break with label-

Syntax:

```
break labelname;
```

It is used to terminate the specific loop of statement and that specified loop has to be marked with labelName. The specified loop should be within the nested loop statement

ex-

```
Class Test {
  public static void main (String arg[]){
    JLC:
      for (int i=1; i<=3;i++){
          System.out.println ("The value of i is:"+ i);
      SD:
        for ( int j=1; j<=3;j++){
          System.out.println (" i+"\t" +j);
        if (i==2)
         break JLC;
         }
      }
  }
}
```

O/P

```
The value of i is:1
  1    1
  1    2
  1    3
The value of i is: 2
  2    1
```

```
class Test {
    public static void main (String arg[]){
    int x=1;
    JLC:{
        System.out.println("JLC label Started");
        SD:{
            System.out.println("SD label started");
            SRI:{
                System.out.println("SRI label Started");
                    if (x==3)
                    break JLC;
                System.out.println("SRI label stopped");
                }
            System.out.println("SD label stopped");
            }
        System.out.println("JLC label Stopped");
        }
    }
}
```

# NOTE-

break without label cannot be used outside the loop statement and switch statement but break with label can be used outside the loop statement or switch statement or inside the loop statement or switch statement but label declaration is mandatory.

b) Continue -

Continue is a predefined keyword which is used for continuing the current loop of statement and after continue statement, which ever the statements are there, that will skip the execution.

continue keyword has to be used inside the looping control statement.

ex-

```
Class Test{
    public static void main (String arg[]){
        for (int i=1; i<=5; i++){
                System.out.println ("i);
                if (i==3)
                continue;
                System.out.println ("Hello Guys");
                }
        }
    }
```

O/p

```
1
Hello Guys
2
Hello Guys
3
4
Hello Guys
5
Hello Guys.
```

# The statement Hello Guys was skipped at i=3 and others are executed.

# Continue statement cannot be used out side the looping or switch statement.

```
int a=10;
if (a==10)
continue;   -> invalid syntax.
```

```
class Test {
    public static void main (String arg[]){
        for (int i=1; i<=3; i++){
            for (int j=1; j<=3; j++){
            System.out.println( i+"\t"+j);
            if (i==2)
            continue;
            System.out.println("Inside j loop");
            }
        System.out.println("Inside i loop and outside loop");
        }
    }
}
```

O/P

```
1   1
Inside J loop
1   2
Inside j loop
1   3
Inside j loop
Inside i loop and outside loop
2   1
2   2
2   3
Inside i loop and outside loop
3   1
Inside j loop
3   2
Inside j loop
3   3
Inside j loop
```

=> continue is classified into two types.

a) continue without label

b) continue with label.

a) continue without label-

It is used for continuing the current loop statement.

b) continue with label-

It is used for continuing the specified loop of statement and the specified loop is marked with label declaration and it should be within the nested loop of statement.

ex- infinite while loop using continue-statement-

```
class Test{
    public static void main (String arg[]){
        int i=1;
        while(i<=5){
        System.out.println(i);
        if (i==3)
        continue;
        i++;
        }
    }
}
```

O/P

```
1
2
3
3
3
.
.
.
.
.
```

infinite loop

# make sure to use updational statement before continue statement.

# When we are using continue keyword in the loop statement (i.e inside the for loop). continue keyword will make the control to the updational statement and in the case of while and do while loop the continue statement will make the control to the conditional statement. So, whenever we are specifying continue keyword inside the while and do while loop, make sure that continue keyword has to be provided after updational statement only. In case if you provide updational statement after continue keyword then it will skip the updational statement and will make finite loop as infinite looping statement

Both continuous with label and continuous without label must and should be used inside the looping control statement only.

Q. Write a program to display whether character is alphabet or digit or special character symbol.

Ans.

```
class Test {
    public static void main (String arg[]){
        char ch = 'A';
        if (ch >= 'A' && ch <= 'Z'){
            System.out.println("It's capital Alphabet");
        } else if (ch >= 'a' && ch <= 'z') {
            System.out.println("It's small Alphabet");
        } else if (ch >= '0' && ch <= '9'){
            System.out.println("It's a Number");
        } else
            System.out.println("It's a Symbol");
    }
}
```

12/10/2017

10) Array:-

→ It is a reference variable.
→ It is used for collecting multiple values of similar datatype
→ It is a homogenous data structure.
→ We can represent the array by using dimensional symbol []
→ Based on dimensional symbol, array is classified into two types
  i) Single Dimensional array [1D Array]
  ii) Multi Dimensional array [2D, 3D, 4D, ... Array]
→ If we want to use array in the application, we need to follow three steps
  i) Array Declaration.
  ii) Array Construction.
  iii) Array Initialization.

# NOTE-

1) Premitive variables can hold only single values but if we want to store collection of value in premitive variable, it is not possible and whenever we are reassigning the values then older value will replace with new values. So, whatever we have initialized the last initialization then that value will be available in premitive variable and in case if you want to store some collection of values then we need to declare multiple variable of same datatype but with different name and there is a complexity while accessing the variables for collection of values and we need to remember the name of variable.
2) So to overcome the complexity of premitive variable, java vendor has provided a new concept (i.e Array). Array is considered as reference variable which stores the actual value in different memory and array reference variable will store the address of actual value.

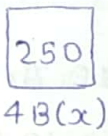
## => Syntax of Premitive variable-

### ① declaration-

int x; <datatype> <variable name>;


### ② initialization-

x = 250; <variable name> = <actual value>;

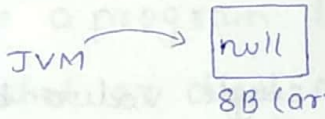



## => Syntax of Array-

### ① Array declaration-

<modifiers>* <datatype> <ArrayName> [];

int arr[];

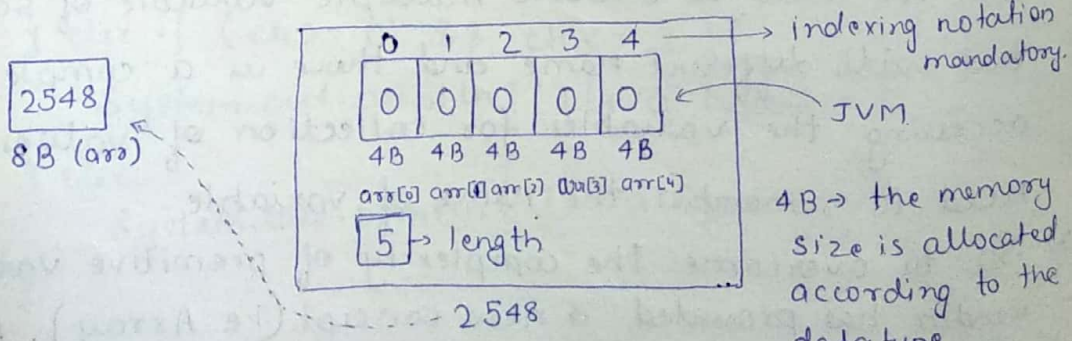



### ② Array construction-

<ArrayName> = new <Datatype> [size];

↳ mandatory.
(No of values we are collecting)

arr = new int [5];



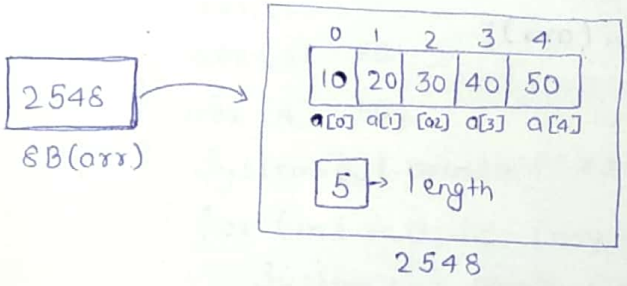
③ Array Initialization-

Initializing actual values for array can be represented by using indexing notation.

Range of index = 0 value to size-1

```
arr [0] = 10;
arr [1] = 20;
arr [2] = 30;
arr [3] = 40;
arr [4] = 50;
```

arr[0] .... arr[4] is called as indexing notation.




The zero's will be replaced with the actual values and will be stored in different memory location.

=> Array declaration-

Syntax-

```
int arr[];
int []arr;
int[] arr;
```

=> Array construction-

```
arr = new int[5];   → valid
arr = new int[];    → invalid (size is mandatory)
```

=> Array declaration-

ex-

```
Class Test{
    Static int arr[];                    → valid
    public static void main (String arg[]){
        System.out.println(arr);
    }
}
```

O/p

null

ex-
```
Class Test{
    Static int arr[5];
    public static void main (String arg[]){
        System.out.println (arr);
    }
}
```
→ invalid, size should not be provided during declaration

=> Array construction-

ex-
```
Class Test{
    public static void main (String arg[]){
        int arr[];
        arr = new int[5];
        System.out.println (arr);
    }
}
```
O/p
address will be displayed

ex-
```
Class Test{
    public static void main (String arg[]){
        int arr[];
        arr = new int[];
        System.out.println (arr);
    }
}
```
→ invalid, dimension missing

ex- Array length checking and elements checking:
```
Class Test{
    public static void main (String arg[]){
        int arr[];
        arr = new int[5];
        System.out.println (arr.length);
        System.out.println (arr[0]);
        System.out.println (arr[1]);
        System.out.println (arr[2]);
        System.out.println (arr[3]);
        System.out.println (arr[4]);
    }
}
```

ex- array element assignment-

```
class Test{
    public static void main (String arg[]){
    int arr[];
    arr = new int [5];
    System.out.println(arr.length);
    System.out.println("**********");
    for (int i=0; i<=(arr.length-1); i++){
        System.out.println ("arr["+i+"]"+"\t"+arr[i]);
        }
    arr[0] = 10;
    arr[1] = 20;
    arr[2] =30;
    arr[3] = 40;
    arr[4] = 50;
    System.out.println ("**********");
    for (int i=0; i<=(arr.length-1); i++){
        System.out.println ("arr["+i+"]"+"\t"+arr[i]);
    }
  }
}
```

O/p

```
5
**********
arr[0]    0
arr[1]    0
arr[2]    0
arr[3]    0
arr[4]    0
**********
arr[0]    10
arr[1]    20
arr[2]    30
arr[3]    40
arr[4]    50
```

# NOTE -

Declaration and construction can be done in the same line but we need to follow some rules:-

1) Size will be provided at construction side.
2) Size should be int compatable datatype only. (i.e. byte, short, int or char only)
3) It should be positive value.

ex-

```
Class Test {
    public static void main (String arg[]) {
    int arr [] = new int [5];
    System.out.println (arr.length);
    }
}
```

* dimension legal values

[5L] → invalid

[85.25] → incompatable

[6f] → invalid

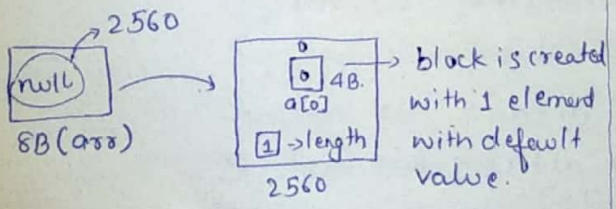
[true] → invalid

[-23] → runtime error, it cannot be supported. Negative array size exception.

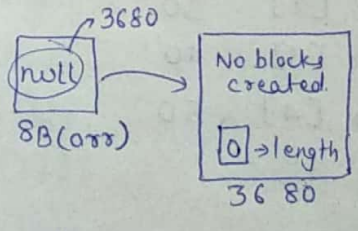
[2147483647] → runtime error, Requested array size exceeds VM limit.

[0] → it can be take but no elements can be accessed.

int arr [] = new int [1];




int arr [] = new int [0];




No element representation.

```
{ int arr [] = null;
    System.out.println (arr.length);
      System.out.println (arr [0]);
    }
```

No actual memory is created, so null point exception.
So, no length can be accessed hence no element can be accessed.

NOTE-

1) While declaring the array, we should not provide the size value in the dimension.
2) In case if you provide at the time of declaration then it is an illegal start of expression.
3) Always we need to provide the size value at the time of construction. In case if you donot provide the size value at the time of construction, then it will give "array dimension is missing" error.
3) While accessing the array element, we need to take indexing notation but the index range starts from 0 to (size-1) and if you try to access any index out of range then it will throw error "array index out of bounds exception" and we need to access the index value always within the range only.
4) Whatever size we are providing for array value should be int compatable datatype (i.e - it will allow byte, short, int and character datatypes) and in case if you provide other datatypes like long, double, float, boolean these are incompatable datatype.
5) Array size value must and should be +ve value, in case if you are constructing the array with -ve value then it will throw error as "Negative array size exception".

6) The maximum value we can provide for the size of array is int maximum value i.e 2147483647 to 0 but as per the integer range value if you provide the size then if sufficient memory is not there to allocate multiple block of memory then it will give out of memory error.

7) We can assign null value to the array reference variable and in this case actual memory will not be created. So, if you try to access any length value or element value, it will throw "null pointer exception" error. Since memory allocation is not there for length variable.

8) We can construct the array with size value as 0 also but no element will be allowed / allocate the memory and only length variable will take the memory and assign with zero value. So, if you try to access any index notation then it will throw. "array index out of bound exception.

13/10/2017

# If we need to change arr.length then it is not possible because internally it is becoming as constant.

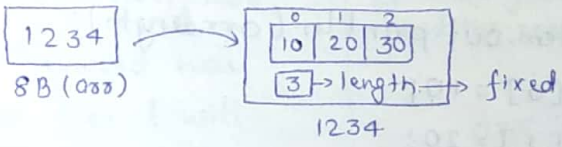
So,

array length and size is fixed value.

```
{
    int arr [] = new int [3];
    System.out.println (arr.length);
        arr [0] = 10;
        arr [1] = 20;
        arr [2] = 30;
    // arr.length = 5  → not possible (i.e. invalid).
        arr = new int [5]; → valid
    }
}
```

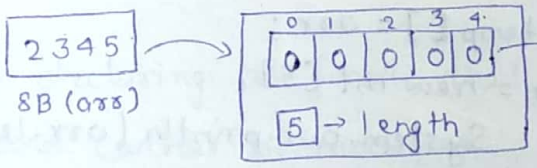
# memory allocation.

int arr [] = new int [3];




⇒ after using new int []. we can change the length with a new memory allocation.

arr = new int [5];




→ default value provided by JVM so we need to reinitialize the value/elements.

⇒ To load value again.

```
arr [0] = 10;
arr [1] = 20;
arr [2] = 30;
arr [3] = 40;
arr [4] = 50;
```

It's complex because its going to new memory with the default value as per the datatype.

# So, to take the older value, we can take some temp memory.

interview question-

```
{
    for (int i=0; i<=temp.length; i++){
        arr[i]= temp[i];
    }
```

So,

```
arr[0] = temp[0];
arr[1] = temp[1];
arr[2] = temp[2];
```

ex-

```
Class Test {
    public static void main (String org[]) {
        int arr[] = new int[3];
        System.out.println(arr.length);
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        for (int i=0; i<=(arr.length-1); i++){
            System.out.println(arr[i]);
        }
        int temp[] = arr;
        arr = new int[5];
            System.out.println(arr.length);
        for (int i=0; i<=(temp.length-1); i++){
        arr[i] = temp[i];
        }
        arr[3] = 40;
        arr[4] = 50;
        for (int i=0; i<=(arr.length-1); i++){
            System.out.println(arr[i]);
        }
    }
}
```

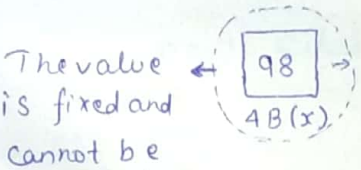
# NOTE-

Array's are static in nature because the length of array is fixed and we cannot modify the length of array within the allocated memory location.

In case if you want to modify length of array then we can modify in the different memory location but not in the allocated memory of array reference variable.

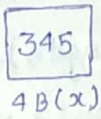
## ⇒ Use of "final" in array-

```
final int x = 98;
```

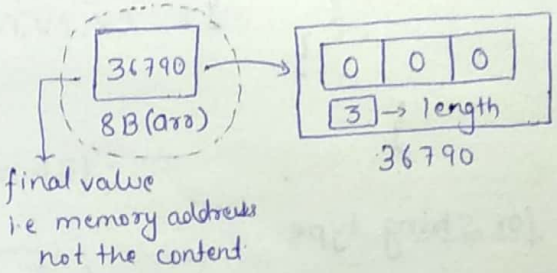



x = 345; → not possible

if final keyword was not used then it will be modified as




```
final int arr[] = new int[3];
```




arr = null → invalid

arr = new int[10]; → valid

When we are declaring as final, that is considered as fixed content and cannot be modified

- If you declare premitive variable as final then if you try to assign any different actual value, then it will give the error. because premitive variable will store the actual content within the allocated memory. So, it is a fixed value.
- If we declare any reference variable as final then we can modify the values of the element of array but we cannot modify the address of the reference variable and we cannot modify the length of the array because array reference variable allocated memory contains the address of actual value's memory. So. allocated memory address is fixed but not its actual element values.

=> Array declaration, construction and initialization in same line.

<Datatype> <Array Name> = {V1, V2, V3, V4, V5, ....};

ex-

```
Class Test{
    public static void main (String arg[]){
        int arr[] = {10,20,30,40,50};
            System.out.println(arr.length);
            for(int i=0; i<=(arr.length-1); i++){
              System.out.println(arr[i]);
            }
        }
    }
```

ex- for string type-

```
Class Test{
    public static void main (String arg[]){
        String arr[] = {"SRI","NIVAS", "DANDE");
            System.out.println(arr.length);
            for(int i=0; i<=arr.length-1; i++){
              System.out.println(arr[i]);
            }
        }
    }
```

ex-

int arr[] = {}; -> valid, but the count will zero for length.

```
int arr[];
arr = {10, 20,30, 40}; -> invalid syntax.
```

# whenever we are expecting reference, we need to provide address but not the content

using temp while increasing the length of the array to save the elements value is a wastage of memory.

To overcome this we have Anonymous Array.

* Anonymous Array:-

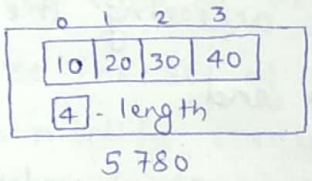
Array declaration without any name

Syntax

```
new <datatype> [] { V1, V2, V3, V4, V5, .... };
```

ex-

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




ex-

```
class Test{
    public static void main (String arg[]){
        int arr[] = null;
        arr = new int [] {10,20,30};
        for(int i=0; i<arr.length; i++){
            System.out.println (arr[i]);
        }
    }
}
```

→ We should not provide the size for Anonymous array.

If we initialized and provide size then how we can provide not the actual value.

```
new int[6]; → invalid
```

# using show method for Anonymous Array.

```
Class Test {
    public static void main (String arg[]){
        show(new int[]{10,20,30,40,50});
        }

    static void show(int arg[]){
        for(int i=0; i<arr.length; i++){
            System.out.println(arr[i]);
        }
    }
}
```

* Enhanced loop-

It is used for accessing the element of array from starting to end.

It is for accessing start value to end value but not in reverse order.

ex.

```
Class Test {
    public static void main (String arg[]){
        int arr[]={10,20,30,40,50};
        for(int x: arr){
            System.out.println(x);
        }
    }
}
```

O/p

```
10
20
30
40
50
```

16/10/2017

# While initializing the input values for variables we have two ways:-

1) Static initialization
2) Dynamic initialization.

## 1) Static initialization-

Within the source code if we initialize the value and if we want to modify the input value and after modifying we need to recompile and execute the application. So, it is a time taking process to compile the application everytime to same piece of code.

## 2) Dynamic initialization-

At the time of execution of application, we need to initialize the set of input values and in case if we want to modify then no need to recompile directly, we can modify at the time of execution.

## * Command line argument-

It is specified as the array parameter for the standard main method which is used for collecting dynamic input values for executing the application.

=> It it provided in standard main method only.

```
javac Lab.java
java Lab V1 V2 V3 V4 V5 .... etc
```

=> By default CLA is 1D array.

ex-

```
class Test {
    public static void main (String args[]) {
        if (args.length >= 2) {
            String str1 = args[0];
            String str2 = args[1];
            int a = Integer.parseInt (str1);
            int b = Integer.parseInt (str2);
            int res = a + b;
                System.out.println ("Result is :" + res);
        } else {
            System.out.println ("Provide CLA i/p value");
        }
    }
}
```

=> To execute type on cmd.

```
> javac Test.java
> java Test 10 20 ↵
```

O/p

```
Result is : 30
```

# When we provide the value then internally.

```
String args[] = new String[5];
    args[0] = "10";
    args[1] = "20";
    args[2] = "30";
    args[3] = "40";
    args[4] = "50";
    args[5] = "60";
```

ex-

```
Class Test {
    public static void main (String jlc []){
        System.out.println ("Main Started");
        System.out.println ( jlc.length );
        for( String st:jlc){
            System.out.println (st);
        }
    }
}
```

execution-

=> java Test *.java

It will show all .java files of the workspace folde

=> java Test *.xml

If .xml file is there, it will display the file or else it will display "*.xml" with "1".

=> java Test "*.java"

It will display "*.java" with "1".

=> Command line arguments are by default declared in String datatype which is an array representation then the need of string datatype is at the time of execution which type of input values will provided by user is don't know so to take any type of dynamic input value, string type is compatable. So, inside the main method they provide as String array declaration and array is given to collect multiple values.

* Array is defined on dimentional basis-

1) Single Dimensional Array - 1D Array

2) Multi Dimensional Array - 2D, 3D, 4D Array

→ 1D Array-

Collection of values

→ 2D Array/3D/4D -

Array of Arrays (collection of Arrays)
Nested Arrays.

1D Array → Collection of Array.
2D Array → Collection of 1D Array.
3D Array → Collection of 2D Array.

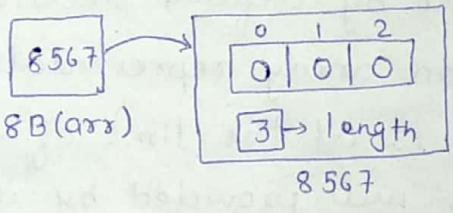
*-1D Array-

Collection of elements.

```
int arr[];
```


declaration.

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




construction.

```
arr [0] = 10;
arr [1] = 20;
arr [2] = 30;
```

initialization.

## 2 D Array -

collection of 1 D Array

```
int arr [][];
```

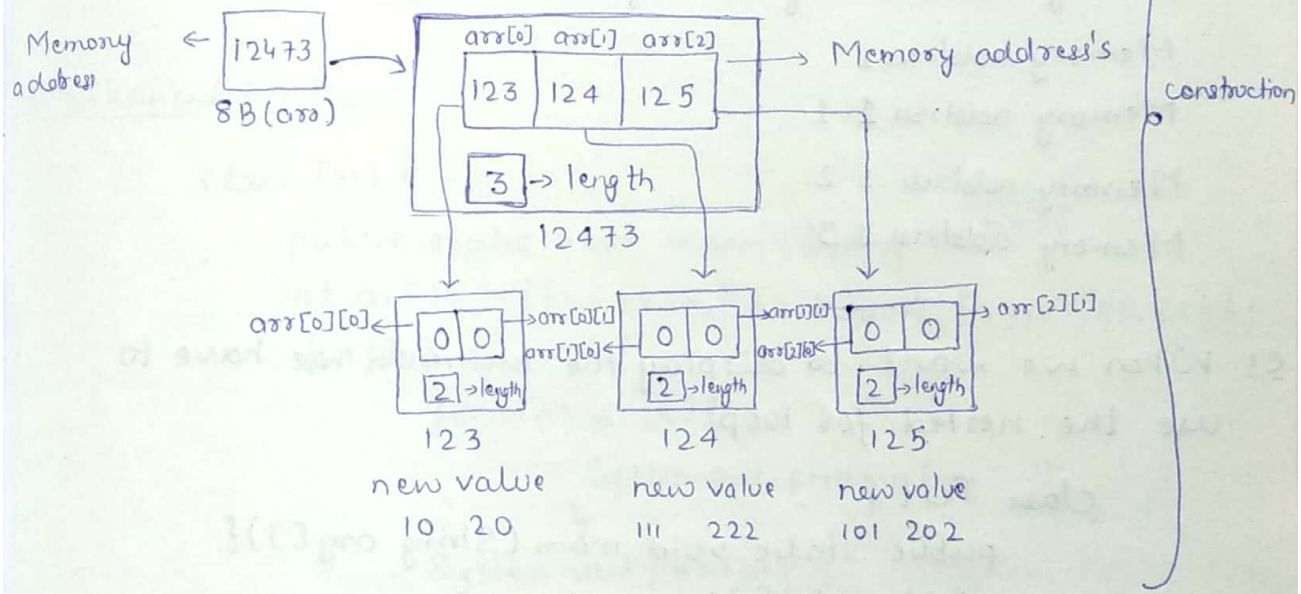

} declaration

```
arr = new int [S1] [S2]
```

S1 → No of 1D Array

S2 → elements of each Array (optional)

```
arr = new int [3] [2];
```




# NOTE.

If we don't define the secondary size then we can have different size in all three arrays (i.e 5, 4, 2).

```
arr [0] [0] = 10;
arr [0] [1] = 20;

arr [1] [0] = 111;
arr [1] [1] = 222;

arr [2] [0] = 101;
arr [2] [1] = 202;
```

} initialization.

ex.

```
Class Test{
    public static void main (String jlc[]){
        int ar[][];
        ar= new int[3][2];
            System.out.println (ar);
        for(int i=1; i<ar.length; i++){
            System.out.println (ar[i]);
            }
        }
    }
```

O/P

memory address's of all reference arrays.

Memory address 1

Memory address 1 1

Memory address 1 2

Memory address 1 3

ex- When we want to display the elements, we have to use the nested for loop.

```
class Test{
    public static void main (String arg[]){
     int ar[][];
     ar= new int[3][2];
          System.out.println (ar.length);
          System.out.println (ar[0].length);
          System.out.println (ar[1].length);
          System.out.println (ar[2].length);
          ar[0][0] = 10;
          ar[0][1] = 20;
          ar[1][0] = 111;
          ar[1][1] = 222;
          ar[2][0] = 101;
          ar[2][1] = 202;
     for(int i=0; i< ar.length; i++){
          for(int j=0;j<ar[i].length;j++){
               System.out.println(ar[i][j]+" ");
          }
          System.out.println();
}}}
```

=> for declaration, construction and initialization in one line.

```
class Test {
    public static void main (String jlc[]) {
        int ar[][]= {{1,2,3,4,5,6},{11,22},{101},{25,45,65,85}};
        for (int i=0; i<ar.length; i++) {
            for (int j=0; j<ar[i].length; j++) {
                System.out.println (ar[i][j]+" ");
            }
            System.out.println ();
        }
    }
}
```

=> enhanced for loop-

```
class Test {
    public static void main (String jlc[]) {
    int ar[][]={{1,2,3,4,5},{11,22},{101},{25,45,65,85}};
     for (int x[]; ar) {
            for (int y: x) {
                System.out.println (y+" ");
            }
            System.out.println ();
        }
    }
}
```

* To make Anonymous Array-

```
int ar[][]= new int[][] {{1,2,3,4,5,6},{11,22},{101}};
```