Language fundamentals

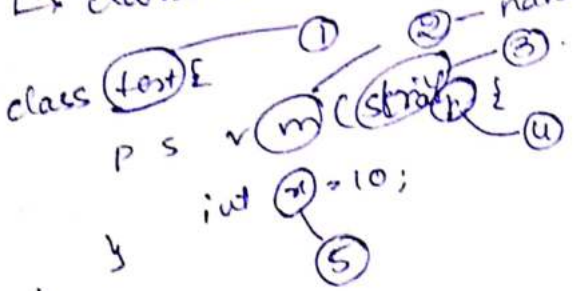
9/1/2020

1. Identifiers
2. Reserved words
3. Data-types
4. Literals
5. Arrays
6. Types of variables
7. Var-arg method
8. main method
9. Command line arguments
10. Java coding standards

Identifiers.

name in java program - identifiers

↳ classname, method, variable, label name



no. of identifiers = 5

test
main
String
args
x

rules:

1. a to z, A to Z, 0 to 9, $, _ (underscore) } allowed characters in identifiers

If any other character used we will get compile time error.

eg: total_num ✓ total # x.

2. total123 ✓ identifier should not start with digit
   123total X

3. We can differentiate w.r.t case

Java identifiers are case sensitive.

eg: int number = 10;
int Number = 20;
int NUMBER = 30; } all are different.

4. In java max characters allowed are — no length constraint → no (in limit for java identifier but not recommend to take too lengthy.

5. int x = 10; ✓
   int if = 20; ← reserved word → we can't use as identifier

6. int String = 888
   sop (String);
   predefined
   ↓
   O/P = 888
   valid. we will get O/P

   int Runnable = 999;
   sop (Runnable);
   interface name
   ↓
   O/P = 999.
   valid

* All java class & interface name we can use as identifiers | but recommended.

7. which of the following are valid Identifiers.

total-number ✓
total # X
123total X
total123 ✓
ca$h ✓
-$-$-$-$- ✓
au@hands X
Java2share ✓
Integer ✓
Int ✓
int X

In Java return type is mandatory, if a method wont return anything then we have declare that method with void return type.
But in C return type is optional, default return type is int,

Reserved words

In Java 53 reserved words are there to represent some meaning or functionality.

(50) key words

If the reserved word associated with functionality

- 48 used keyword
  - if
  - else
  - switch ---
- 2 unused key
  - goto
  - const

Reserved literals (3)

If the reserved word only to represent a value.

- true
- false
- null

keywords for data type (8)

- byte
- short
- int
- long
- float
- double
- boolean
- char

Keywords for flow control (11)

- if
- else
- switch
- case
- default
- do
- while
- for
- break
- continue
- return

Keywords for modifiers (11)

- public
- private
- protected
- static
- final
- abstract
- synchronized
- native
- strictfp (1.2 v)
- transient
- volatile

keywords for exception handling (6)

- try
- catch
- finaly
- throw
- throws
- assert (1.4 v)

class related keyword (6)

- class
- interface
- extends
- implements
- package
- import

object related (4)

- new
- instanceof
- super
- this

void return type keyword (1)

- void

unused Keywords. (2)

goto : usage of goto created several problems in old lang. & hence some people banned this Keyword in Java.

const : use final instead of const.

Note: goto & const are unused keyword & if we are trying to use we will get compile time error.

Reserved literals : true, false, null.

true } values for
false } boolean data type.

null → default value for object reference.

enum keyword (1) (1.5 V).

→ we can use enum to define a group of named constant.

eg: enum month { JAN, FEB, ... DEC }

→ all 53 reserved words are small characters. (lower case alphabet symbols)
→ " " " " do not have numbers.
→ In Java we have only new keyword & no delete keyword because destruction of useless objects is the responsibility of garbage collector.

→ strictfp → 1.2 V
→ assert → 1.4 V
→ enum → 1.5 V

} new keywords in Java

strictfp but not strictfp
instanceof " instanceOf
synchronized " synchronize
extendes " extend
implements " implement
import " Imports
const " constant

Q. which of the following contain only java reserved words.

1. new, delete ×
2. goto, constant ×
3. break, continue, return, exit ×
4. final, finally, finalize ×
5. throw, throws, thrown ×
6. notify, notifyAll method ×
7. implements, extends, imports ×
8. sizeof, instanceof ×
9. instanceOf, strictfp ×
10. byte, short, Int ×

reserved words.

x. None of the above is the correct Ans.

Q. which of follow are java reserved words

public — } reserved
static —
void —
main — method
String — predefined class
args — name of variable } not reserved word.

Data types

int x = 10.5 } Comes in C but not in java
boolean b = 0 }

each & every Java is

- strongly typed language (because of below reasons)
- i.e typechecking is very important.
- In Java every variable & expression has some type
- each & every data type is clearly defined
- every assignment should be checked by compiler for type compatability

→ Java has more object oriented programing langu compared to other language.

→ Java is not pure OOPS when it is itself consid because some features not supported by java

→ NO is recommed lang.

- Java is not considered as pure OOPS sence OOP features are not satisfied by Java like (operator overloading, multiple inheritance etc)

more over we are depending on primitive data types which are non objects.

Primitive data types (8)

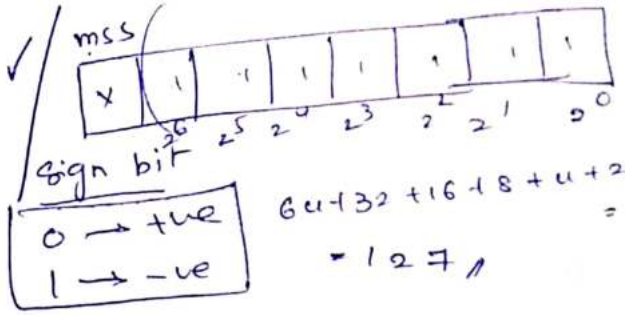
- Number data types
  - Integral data types
    - byte
    - short
    - int
    - long
  - Floating-point data types
    - float
    - double
- Non-numeric data types
  - char
  - boolean

expect boolean & char remaining data types are considered as signed data types because we can represent both positive & negative numbers.

1 byte: size : 1 byte = 8 bits

MAX-VALUE : +127 ✓

MIN-VALUE : −128 ✓



represented in 2's complement.

range −128 to 127.

The most significant bit acts as sign bit

→ +ve numbers will be represented directly in memory

→ where as −ve no. in 2's complement form.

```
byte b = 10;   ✓
byte b = 127;  ✓
byte b = 128;  ✗ error: possible loss of precision
                       found: int
                       required: byte
```

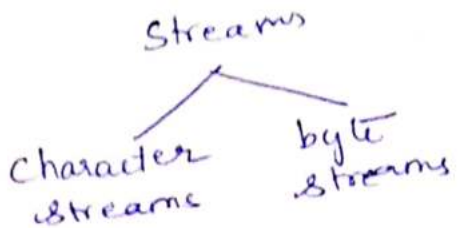
byte b = 10.5; → same error previous found: double req: byte

byte b = true; → error: incompatible types found: boolean req: byte

byte b = "durga" → " " " " found: java.lang.String req: byte

---

file supported from or n/w supported from is byte



→ Byte is the best choice if we want to handle data in form of streams either from the file or from the n/w (file supported / n/w supported for is byte).

---

(2) Short

most rarely used data type

size: 2 bytes (16 bits)

Range: -2^{15} to $2^{15}-1$ [-32768 to 32767].

short s = 32767; ✓

short s = 32768; ✗

short s = 10.5; ✗

short s = true; ✗

short s = "durga"; ✗

→ short data type is best suitable & used for 16 bit processor like 8085. but it is (processors) now out-dated. & hence corresponding short data type is also outdated.

10/1/2020

③ int

common data type used.

size = 4 bytes (32 bits)

Range : -2^{31} to $2^{31}-1$

[-2147483648 to 2147483647]

int x = 2147483647 ✓

int x = 2147483648; ✗ → CE: integer number too large:

int x = 2147483648l; → CE: possible loss of precision
found: long
required: int

int x = true; CE: incompatible types.
found: boolean
req: int.

④

long

eg: amt of distance travelled by light in 1000 days, to hold this value int may not be enough we should go for long data type.

long l = 1,26,000 × 60 × 60 × 24 × 1000. miles

→ length the no. of characters present in a string file may exceed int range hence, the return type of length method is long "

long l = f.length();

size: 8 bytes (64 bits

Range: -2^{63} to $2^{63}-1$

All the data types byte, short, int, long meant for representing integral values.
if we want to represent floating point values then we should go for floating point data type.

float

① 5 to 6 decimal accuracy

② Single precision
less accuracy

③ size: 4 bytes

④ Range:
$-3.4e38$ to $3.4e38$
$\downarrow$
3.4×10^{38}

double

① 10 to 15 decimal accuracy

② double precision

③ size: 8 byte.

④ Range
$-1.7e^{308}$ to $1.7e^{308}$
1.7×10^{308}

Boolean data type.

In Java size: Not applicable (virtual machine dependent)
Range: " " (But allowed values are true / false)

boolean b = true; ✓
boolean b = 0; ✗ compile error: incompatible type:
found: int
required: boolean.

boolean b = True; ✗
CE: cannot find symbol
symbol: variable True
location: class test.

boolean b = "True"; ✗ CE: incompatible type.
found: java.lang.string.
req: boolean

```
int x = 0;
if(x){
  sop("Hello");
}else{
  sop("Hi");
}
```

In Java it is not applicable two will get compile time error.

CE: incompatible type:
found: int
req: boolean

```
while(1){
  sop("Hello");
}
```

→ In C many times Hello is printed but in Java it is compile time error.

char :-

in C - char size : 1 byte

size : 2 bytes

old languages like C / C++ are ASCII code based. & the no. of different allowed characters are ≤ 256.

→ To represent these 256 characters 8 bits are enough hence the size of char in old languages is 1 byte.

⇒ But Java is unicode based and the no. of different unicode characters are > 256 and ≤ 65536.

→ To represent these many characters 8 bits are not enought so we go for 2 bytes or 16 bits, hence the size of char in Java 2 bytes.

size : 2 bytes

Range : 0 to 65535 (1).

Primitive data types.

data type	size	Range	wrapper class	default
byte	1	-2^7 to 2^7-1 (-128 to 127)	Byte	0
short	2	-2^{15} to $2^{15}-1$ (-32768 to 32767)	Short	0
int	4	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	Integer	0
long	8	-2^{63} to $2^{63}-1$	Long	0
float	4	-3.4e38 to 3.4e38	Float	0.0
double	8	-1.7e308 to 1.76308	Double	0.0
boolean	NA	-NA { But allowed values are true \ false }	Boolean	false.
char	2 byte	0 to 65535	Character	0 { represents space character

* null for the primitive is not applicable

null is the default value for object reference & not applicable for primitives. and if we are trying to use for primitive then we will get compile time error.

char ch = null; CE : incompatible types
found : null type
req : char.

Literals

int x = 10;

int → data type / keyword
x → name of variable / identifier
10 → const value / literal.

Literal : a constant value which can be ~~applied~~ assigned to the variable is called literal

Integral Literals

(to integral data types (byte, short, int, long) we can specify literal value in following ways)

① decimal literal : (base - 10)
0 to 9
int x = 10; ← decimal form

② Octal form (base - 8)
0 to 7
int x = 010; ← 0 in the front of a number represents octal form in Java.

③ Hexa decimal form : (base - 16)
0 to 9
a to f
int x = 0x10;

Hexadecimal form. allowed digits 0 to 9, a to f.

for extra digits (a to f)

Java is case sensitive but for extra digits in hexadecimal it can use anything either small or caps.

→ In very few area Java is not case Sensitive.

The literal value should be prefixed with

0x or 0X.

```
int x = 10; ✓
int x = 0786 ; X CE : integer number too large
int x = 0777 ; ✓
int x = 0XFace; ✓
int x = 0XBeef;
int x = 0XBeer;
```

0 to 9.
a to f

---

eg class Test {
p s v m (Str [] args) {
int x = 10;
int y = 010;
int z = 0x10;
SOP (x + " " + y + " " + z);
}
}

o/p : 10 8 16

$(10)_8 = (?)_{10}$

$0 \times 8^0 + 1 \times 8^1 = 8$

$(10)_{16} = (?)_{10}$

$0 \times 16^0 + 1 \times 16^1 = 16$

By default every integral literal is of int type.

By we can explicitly as long type by suffixed by l or L.

eg: 10l, 0x10l, 010L,

```
int x = 10; ✓
long l = 10L; ✓
int x = 10L; X ——— CE: PLP
long l = 10; ✓
```

CE: Possible Lossy Conversion
found: long
req: int.

**By default every integral literal is int

There is no direct way to specify byte & short literals explicitly but indirectly we can specify, whenever we are assigning integral literal to byte variable & if the value within the range of byte then compiler treats it automatically as byte ily as short.

```
byte b = 10; ✓
byte b = 127; ✓
byte b = 128; X ——— CE: PLP
```

```
short s = 32767; ✓
short s = 32768; X
```

Floating point literals :

float f = 123.456; ✗ CE : P L P.

↓

it is double.

***
every floating point literal is by default double. 8 byte to a byte not suitable.

float f = 123.456F; ✓

↓

treated as float.

double d = 123.456 ; ✓

we can specify floating pt literals as float by suffixed with f or F.

double d = 123.456D ; ✓

float f = 123.456d ; ✗ CE. P L P
found : double
req : float.

we can specify explicit floating pt literal as double suffixed by D or d, but not required

double d = 123.456 ; ✓

double d = 0123.456; ✓ not octal but decimal value.

double d = 0X123.456; ✗ : CE malformed floating pt literal.

We can specify floating pt literals only in decimal form & not in octal & hexa decimal forms.

double d = 0786; ✗ treated as integral because no.

double d = 0xFace; ✓ integral literal

o/p : 64206.0.

double d = 10; ✓ o/p 10.0.

double d = 0786.0; ✓

double d = 0xface.0; ✗

We can assign integral literal directly to floating point variables. and that integral literal can be specified in either decimal, octal or hexadecimal form.

double d = 10 ✓

int x = 10.0 ✗ C.E : P LP found: double req: int.

we cant floating pt literals to integral type

double d = 1.2e3; 1.2×10^3

sop(d); o/p 1200.0 $= 1.2 \times 1000$

$= 1200.0$

float f = 1.2e3; ✗ C.E : P L P found: double req: float

float f = 1.2e3F; ✓

float f = 1.2e3F ✓

we can specify floating pt literal even in exponential form (scientific notation).

Boolean literals

true / false → only allowed values.

boolean b = true; ✓

boolean b = 0; ✗ — CE: incompatible types
found: int
req: boole

boolean b = True; ✗ — CE: cannot find symbol
variable
location: class test

boolean b = "true"; ✗ — CE: incompatible types
found: java.lang.string
req: boolean.

Char literals

char ch = 'a'; ✓ we can char literal as single character within single coats.

char ch = a; ✗

char ch = "a"; ✗ — CE: incompatible types
found: j.l.stig
req: char

char ch = 'ab'; ✗ — CE1: unclosed char literal
CE2: unclosed " "
CE3: not a statement

char ch = 97; ✓ — range - 0 to 65535
sop(ch); a

→ we can specify char literal as integral literal which represent unicode value of the character & the integral literal can be specified in decimal, octal or hexadecimal & in a range 0 to 65535.

char ch = 0xface; ✓

char ch = 0777; ✓

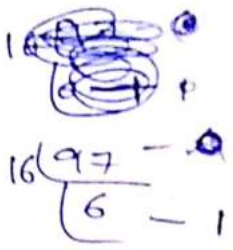
char ch = 65535; ✓

char ch = 65536; ✗ CE: PLP
found: int
req: char

fact 5: 17:00 min! 1970, 1971, 1972: ~~always~~ pruned ? because the corresponding char not available in system.

→ '\uXXXX' — unicode representation

4-digit hexa-decimal number



char ch = '\u0061';

sop(ch);

→ we can represent char literal in unicode representation.

char ch = '\n'; ✓

char ch = '\t'; ✓

char ch = '\m'; ✗ — illegal escape characters

~~escape~~ characters

Every escape character is a valid char literal.

8 escape characters are there in Java.

\n → New line

\t → Horizontal Tab

\r → carriage return

\b → Back space

\f → form feed

\' → single quote

\" → double quote

\\ → Back slash.

```
sop ("This is ' symbol");   X.
sop ("This is \' symbol");  ✓   .o/p
                                This is ' symb
sop ("This is " symbol");   X.
sop ("This is \" symbol");  ✓
sop ("This is \\ character"); o/p This is \chara
```

Carriage return : moving to first cell of next line.

---

which of the following are valid

o to 9
a to f.

```
char ch = 65536;      X
char ch = 0xBeer;     X
char ch = \uface;     X
char ch = '\ubeef';   ✓
char ch = '\m';       X
char ch = '\iface';   X
```

String literals

```
String s = "Sin";
```

any seq. of characters within double coats is treated as string literal.

1.7 Version enhancements w.r.t literals

① Binary literals:

```
int x = 0B1111;
sop(x); 15
```

~~for~~ 1.6 we can specify in binary, octal & hexadecimal

bcd 1.7 we can specify in binary also.

suffixed with 0B

double d = 123456.789;

// but from 1.7 V.

double d = 1_23_456.7_8_9;

usage of underscore symbol in numerical literals.

→ because for readability.

from 1.7 V we can _ symbol b/w digits of numeric literals.

* At the time of compilation these underscore symbols will be removed automatically, hence after computation.

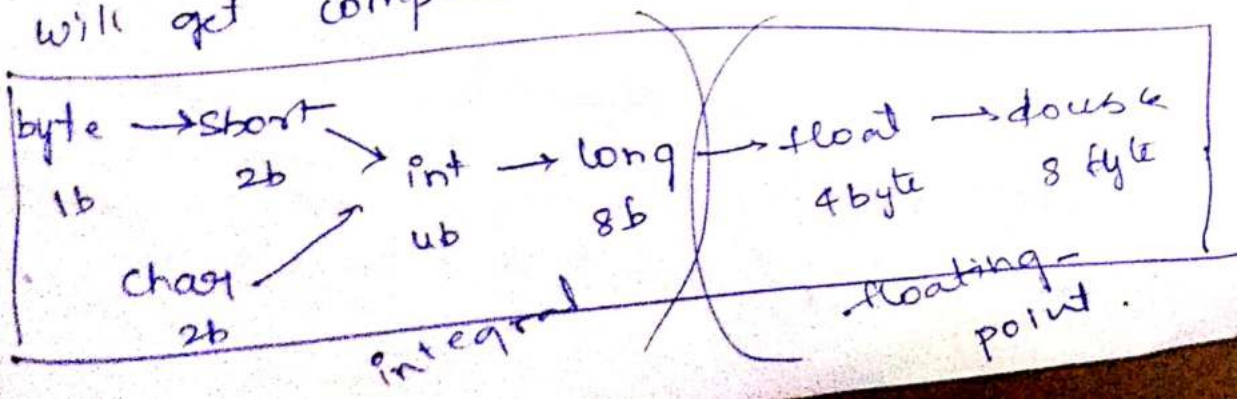
double d = 123456.789;

✓ double d = 1__23_4_5__6.7_8_9;
double d = 1___2_3_4_5_6.7_8_9;

→ we can use more than one underscore symbol b/w the digits.

double d = _1_23_456.7_8_9;
double d = 1_23_456_.7_8_9;
double d = 1_23_456.7_8_9_; } X

→ we can use underscore symbol only b/w the digits if we are using anywhere else we will get compile time error.



NOTE: 8 byte long value we assign to a byte float variable because both are following diff memory representations internally.

```
float f = 10.1;
SOP(f) ; 10.0 ✓ o/p:
```

short signed char unsigned.

even though both are 2 bytes. we cant assign.

both are if char 0 to 65535 but short can't accept.

14/1/2020

Arrays

→ Arrays are fixed in size — disadv. i.e we cannot create an array this is no ↑ or ↓ size based on our requirement.

→ Arrays contains only homogeneous data element.

→ An array is an indexed collection of fixed no. of homogeneous data elemet.

→ adv. • represent huge values by single variable
• readability

Array declaration

1D array declaration

```
int[] x; ✓ recommend. name is clearly seperated from type.
int x[];
int []x;
```

x is a type of 1D array.

At the time of declaration we can't specify the size.

```
int[6] x; ✗
int[] x; ✓
```

2D array declaration

```
int[][] x; ✓
int [][]x; ✓
int x[][]; ✓
int[] []x; ✓
int[] x[]; ✓
int []x[]; ✓
```

If we want to specify dimension before the variable the facility is applicable only for first variable in a declaration. If we are trying to apply for remaining var will get error

```
int[] a,b; ✓   a→1, b→1
int a[],b; ✓   a→2, b→1
int[] a[],b[]; ✓   a→2, b→2
int[] []a,b; ✓   a→2, b→2
int[] []a,[]b; ✓   a→2, b→3
int[] []a,[]b; ✗ CE
int[] []a,[]b,[]c; ✗
```

3D array decl

```
int[][][] a;
int [][][]a;
int a[][][];
int[] [][]a;
int[] a[][];
int[] []a[];
```

```
int[][] []a;
int[][] a[];
int [][]a[];
int [] a[][];
```

all correct valid

Array Creation

Object can be created only in classes.

* Every array in Java is an object.

hence we can create array by using new operator.

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

→ For every array we have class (but we can't use them) corresponding

```
Sop(a.getClass().getName());
```

o/p `[I` — 1D.

`[[I` — 2D.

Array type	corresponding class name
int[] x;	[I
int[][]	[[I
double[]	[D
short[]	[S
byte[]	[B
boolean[]	[Z

① At the time of array creation compulsory we should specify the size otherwise we will get compile time error.

```
int[] x = new int[];  X
int[] x = new int[6];  ✓
int[] x = new int[0];  ✓
```

eg: of array size 0. ——— no values.

```
psvm(String[] args){
    sop(args.length);    o/p 0
}
```

——— passing value via command line

javac testclass A B C D o/p 4.

int[] x = new int[-3]; no compile time error ✓
but error in Run time : -ve array size exception.

int[] x = new int[10]; ✓
int[] x = new int['a']; ✓

byte b = 20;
int[] x = new int[b]; ✓

short s = 30;
int[] x = new int[s]; ✓
int[] x = new int[10L]; ✗ C.E: PossibleLossOfPrecision found: long required: int.
int[] x = new int[2147483647]; ✓

int[] x = new int[2147483648]; ✗
same time we get runtime error because int is of 4 bytes
2147483647 x 4 is too large & that space may not be available.

RE: java.lang.OutOfMemoryError : Requested array size exceeded virtual limit.

→ In Java

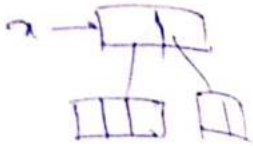
2D-Array Creation

→ In Java 2D-array not implemented by using matrix style.

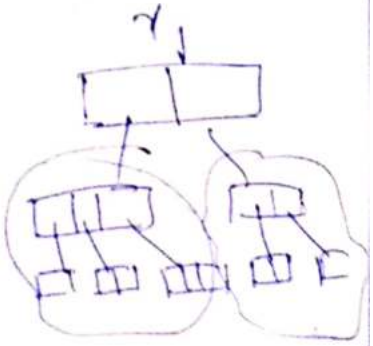
→ We follow array of array's approach for multidimensional array creation.

→ The main adv this approach memory utilization will be improved.

eg: int [][] x = new int[2][];
x[0] = new int[3];
x[1] = new int[2];



int[][][] x = new int[2][][];
x[0] = new int[3][];
x[0][0] = new int[1];
x[0][1] = new int[2];
x[0][2] = new int[3];
x[1] = new int[2][2];



---

int[] a = new int[7]; ✓
int[] a = new int[3]; ✓
int[][] a = new int[][]; ✗
int[][] a = new int[3][]; ✓
int[][] a = new int[][4]; ✗ — without box cannot specify other
int[][] a = new int[3][4]; ✓
int[][][] a = new int[3][4][5]; ✓

int[][] a = new int[3][4][]; ✓

int[][][] a = new int[3][][5]; ✗

int[][][] a = new int[][4][5]; ✗

Array initialization

int[] a = new int[3];

sop(a); ——— o/p [I@3e25a5 (hashcode in hexadecimal)

sop(a[0]); ——— o/p 0 ✓

→ Once we create an array every element by default value but it is 0.

→ Whenever we are trying to print any reference variable two string method will be called which is implemented by default to return the String in ~~hashcode~~ following form

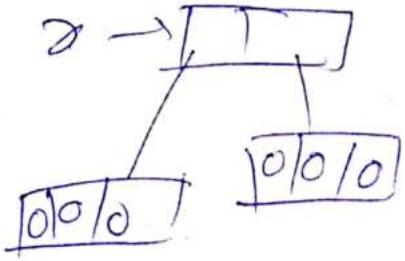
classname@hashcode-in-hexadecimal form

int[][] a = new int[2][3];

sop(a) = [[I@3e25a5 ;

sop(a[0]); [I@19219af ;

sop(a[0][0]); 0 ; ✓



```
int[][] a = new int[2][];
sop(a); [[I@3e25a5;
sop(a[0]); null;
sop(a[0][0]); null pointer exception
```

→ every array ele by default initialized with default values.

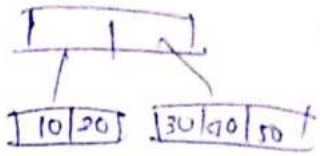
→ if we are not satisfied with default values we can override with customised values.

```
int[] a = new int[6];
a[0] = 10;
:
a[5] = 60;
a[6] = 70; R.E: Array IndexOutofBound
a[-6] = 80; R.E: "
a[2.5] = 90; CE: PLP found: double
                 req: int
```

Array declaration, creation & Initialisation in a single line;

```
int[] a = {10, 20, 30};
char[] ch = {'a', 'e', 'i', 'o', 'u'};
Str[] s = {"A", "B", "CD"};
```

int[][] x = {{10,20},{30,40,50}};



int[][][] x = {{{10,20,30},{40,50,60}},{{70,80},{90,100,110}}};

sop(x[0][1][2]); o/p 60

sop(x[1][0][1]); o/p 80

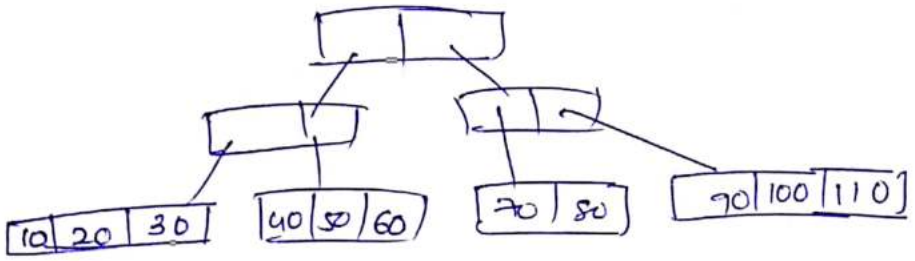
sop(x[2][0][0]); o/p RE: AIOOB

sop(x[1][2][0]); o/p RE: AIOOB

sop(x[1][1][1]); o/p 100

sop(x[2][1][0]); o/p R-ArrayIndexOutOfBound(AIOOB)

We can't directly access so count int diagram



int[] x = {10,20,30}; ✓

int[] x; ✓

x = {10,20,30}; ✗ CE: illegal start of exp.

We can't divide the line, it should be done in one line only

---

length vs length()

length:

int[] x = new int[6];

sop(x.length); 6 ✓

sop(x.length()); ✗

CE: cannot find symbol
symbol: method length()
loc: class int[]

length is a final variable applicable for arrays.
length variable represents the size of array.

String S = "Sion";
SOP(S.length); ——CE: cannot find symbol
Symbol: variable length
Loc: class test
SOP(S.length()); ✓

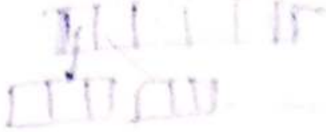
length method is a final method applicable for string objects.
→ length() returns no. of characters present in the String.

NOTE:
length applicable for arrays but not for String objects
length() " " String objects but not for arrays

String[] S = {"A", "AA", "AAA"};
① SOP(S.length); ✓ o/p 3.
② SOP(S.length()); ✗ CE: Cannot find Symbol
Symbol: method length()
Location: class String[].
③ SOP(S[0].length); ✗ CE: cannot find Symbol
Symbol: variable length
Locat: class java.lang.String
④ SOP(S[0].length()); ✓ o/p 1

→ In multidimensional array length represents only base size but not total size.

int[][] x = new int[6][3];
sop(x.length); 6 ✓
sop(x[0].length); 3 ✓



→ There is no direct way to specify the total length of multidimensional array but indirectly we can find as follows.
x[0].length + x[1].length + x[2].length + - -

Anonymous array

an array without name.
just for one time use.

anonymous arr

```
class Test {
    p s v m (String[] args) {
        sum(new int[] {10, 20, 30, 40});
    }
    p s v sum(int[] x) {
        int total = 0;
        for (int x1 : x) {
            total = total + x1;
        }
        sop("The sum" + total);
    }
}
```

new int[3] {10, 20, 30} ✗ CE:

while creating anonymous arrays we should not specify size.

we can create multidimensional anonymous array

```
new int[][] {{10,20},{30,40,50}};
```

Based on our req we can give the name for Anonymous array then it is no longer Anonymous

byte → short → int → long → float → double
char

Array element assignments

Case I:-

```
int[] x = new int[5];
x[0] = 10;   ✓
x[1] = 'a';  ✓
byte b = 20;
x[2] = b;    ✓
short s = 30;
x[3] = s;    ✓
x[4] = 10l;  X CE: PLP
                found: long
                req: int
```

In case of primitive type array as array element we can provide any type which can be implicitly promoted to declared type.

In case of float type arrays the allowed data types of byte, short, char, int long float.

Case II:-

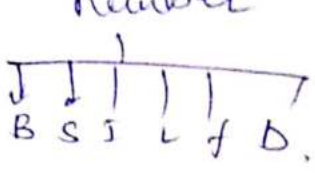
```
object[] a = new object[10];
a[0] = new Object();      ✓
a[1] = new String("sri");  ✓
a[2] = new Integer(10);    ✓
```

eg: Number[] n = new Number[10];

n[0] = new Integer(10);

n[1] = new Double(10.5);

n[2] = new String("sini");



↓ CE: incompatible types
found: java.lang.String
req: java.lang.Number

In case of object type array as array elements we can provide either declared objects or its child class objects

Runnable[] r = new Runnable[10]; ✓

r[0] = new Thread(); ✓

r[1] = new String("sini"); ✗ CE

For interface type array as array element its ~~interface~~ implementation class objects are allowed.

Array type	allowed Element types
Primitive Arrays	any type which can be implicitly promoted to declared type.
Object type Array	either declared type or its child class objects
Abstract class type array. eg: Number	its child class objects
interface type array.	its implementation class objects are allowed

Array Variable Assignments

int[] a = {10, 20, 30, 40};
char[] ch = {'a', 'b', 'c', 'd'};

int[] b = a; ✓
int[] c = ch; ✗

char → int ✓
char[] → int[] ✗

Element level promotions are not applicable at any level.

char → int ✓
char[] → int[] ✗
int → double ✓
int[] → double[] ✗
float → int ✗
float[] → int[] ✗
String → Object ✓
String[] → Object[] ✓

But in case of Object type array child class type array can be promoted to parent class type array.

String[] s = {"A", "B", "C"}; ✓
Object[] a = s;

int[] a = {10, 20, 30, 40, 50, 60};
int[] b = {70, 80};

| 10 | 20 | 30 | 40 | 50 | 60 |

| 70 | 80 |

a = b; ✓
b = a; ✓

Whenever we are assigning one array to another internal elements will not be copied (only reference variables will be reassigned).

⇒ int[][] a = new int[3][];

a[0] = new int[4][3]; ×

↓ CE: incompatible types
found : int[][]
req : int[]

a[0] = 10; ✗ CE

Whenever we are assigning one array to other array the dimension must be matched.
in place of 1D int array we should 1D array only
if we are trying to provide any other dimension we will get compile time error.

⇒ Whenever we are assigning one array to other both dimensions & type must be matched but no need of size.

class Test{
psvm(String[] args){
for(int i=0; i<=args.length; i++){
sop(args[i]);
}
}

i<=args.length

Java Test A B C ↵ ... o/p
A
B
C
AIOOBE

Java Test A B ↵ o/p
A
B
RE: AIOOBE

Java Test ↵ o/p RE : AIOOBE

```
class Test{
  p s v m(String[] args){
    String[] argh = {"x","y","z"};
    args = argh;
    for(String s:args){
      Sop(s);
    }
  }
}
```

args → A B C
argh → x y z

args = A B C
argh → x y z

Java Test ↵ — o/p x y z

Java Test A B ↵ — o/p x y z

Java Test A B C ↵ — o/p x y z

```
int[][] a = new int[4][3]; — 5
a[0] = new int[4]; — 1
a[1] = new int[2]; — 1
a = new int[3][2]; — 4
```

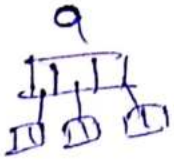
36 + 13 = 49

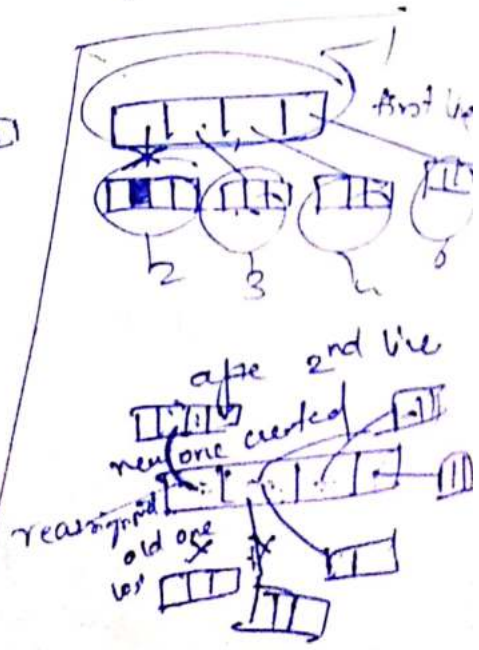
111111 / 23456 / 134567

(10) Total how many objects created?

(20) " " " eligible for GC?

Garbage collection





Types of variables

14/1/2020

division-1:

Based on type of value represented by a variable all variables are divided into 2 types

(i) Primitive variables: can be used to represent primitive values.

```
int x = 10;
```

(ii) Reference variables: can be used to refer objects.

```
Student s = new Student();
```

s is pointing to object.

division-2:

Based on position of declaration & behaviour all variables are divided into 3 types.

① Instance ② static ③ local.

① Instance variable:

→ If the value of a variable is varied from object to object such types of variables are called instance variable.

→ for every object a seperate copy of instance variables will be created.

→ Instance variables should be declared within the class directly but outside of any method / block / constructor.

- I.V will be created at the time of object creation & destroyed at the object destruction.

→ Hence, the scope of I.V is same as the scope of object.

→ ~~objects~~ Instance variables will be stored in heap memory as a part of object.

```
class Test {
    int x = 10;
    p s v m(String[] args){
        sop(x);   — CE: non-static variable x cannot be referenced from a static context
        Test t = new Test();
        sop(t.x);   o/p 10 ✓;
    }
    Public void m1() {
        sop(x);   o/p 10 ✓;
    }
}
```

Instance variable

Instance method

→ we can't access instance variable directly from static area, but we can access by using object reference.

→ But we can access instance variable directly from instance area.

```
class Test {
    int x;
    double d;
    boolean b;
    String s;
    p s v m(String[] args){
        Test t1 = new Test();
        sop(t1.x);   // 0      sop(t1.b);   false
        sop(t1.d);   // 0.0    sop(t1.s);   null
    }
}
```

→ for instance variable JVM will provide default & we don't req. to perform initialisation explicitly.

→ Instance variables also known as object level variables or attributes

Static variables

```
class student {
    String name;
    int rollno;
static String cname;
}
```

→ If the value of a variable is not varied from object to object then it is not recommended to declare variable as instance var. we have to declare such type of variables at class level by using static modifier.

→ In the case of instance variables for every object a seperate copy will be created, but in case of static variable a single copy will be created at class level and shared by every object of the class.

→ static variables should be declared within the class directly but outside of any method / block / constructor.

→ At the time of class loading static variable will be created.

→ " " " " unloading " " destroyed

hence scope of static variable is exactly same as .class file.

Java test ↵

activities done by JVM.

① start JVM
② create & start main thread
③ Locate Test.class file
④ Load Test.class — static var. creation
⑤ execute main() method
⑥ unload Test.class — Static var destruction
⑦ Terminate main Thread
⑧ shut down JVM

→ static variables will be stored in method area.

```
class Test{
    static int x = 10;
    p s v m (String[] args){
        Test t = new Test();
        sop(t.x);
        sop(Test.x);
        sop(x);
    }
```

→ we can access static variables either by object reference or class name but recommend to use class name.

→ within the same class not req to use class name we can access directly.

```
class Test{
    static int x = 10;
    p s v m (String[] args){
        sop(x);
    }
    public void m1(){
        sop(x);
    }
}
```

→ we can access static variables directly from both instance & static areas

Static variables

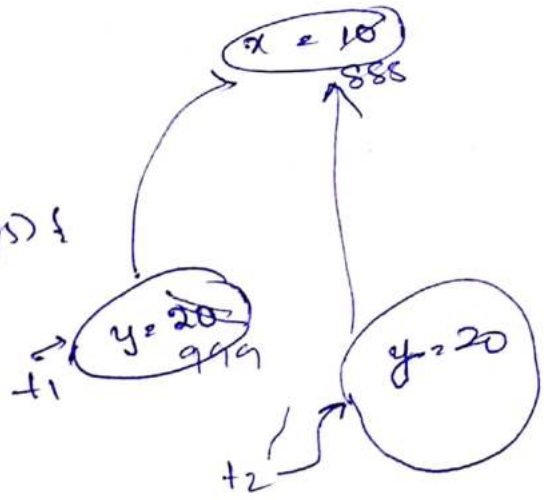
```
class Test{
    static int x;
    static double d;
    static String s;

    P S V m (String[] args){
        sop(x); // 0
        sop(d); // 0.0
        sop(s); // null
    }
}
```

→ for all also JVM will provide default value & not req. to initialize explicitly

→ static variables also known as class level variables or fields.

```
class Test{
    static int x = 10;
    int y = 20;
    P S V m (String[] args){
        Test t1 = new Test();
        t1.x = 888;
        t1.y = 999;
        Test t2 = new Test();
        sop(t2.x + " " + t2.y);
    }
}
```



o/p 888 20

→ static variable one time it is created & shared by all objects.

→ Instance variable diff for diff objects.

Local Variables:

20/1/20

→ Sometimes to meet temporary requirements of the programmer we can declare variable inside a method / block / constructor. are called as local variables / temporary / stack / automatic variables.

→ local variables will be stored inside stack memory.

→ local variables will be created while executing the block in which we declared it.

→ Once block execution is completed, local var will be destroyed.

→ Scope of local variable is the block in which we declared it.

```
class Test{
  p s v main(String[] args){
    int i=0;
    for(int j=0; j<3; j++){
      i=i+j;
    }
    sop(i+" "+j);
  }
}
```

```
class Test{
  p s v main(String[] args){
    try{
      int j=Integer.parseInt("ten");
    }catch(NumberFormatException e){
      j=10;
    }
    sop(j);
  }
}
```

CE: cannot find symbol
symbol: variable j
location: class Test.

- For local var. JVM will not provide default values. Compulsory we should perform initialization explicitly. before using that variable.

```
class Test{
  p s v main(String[] args){
    int x;
    sop("Hello");
  }
}
```

✓

o/p: Hello.

```
class Test{
  p s v m(String[] args){
    int x;
    sop(x);
  }
}
```

CE: variable x might not have been initialized.

If we are not using it is not required to perform initialisation.

```
class Test{
  p s v m(String[] args){
    int x;
    if(args.length>0){
      x = 10;
    }
    sop(x);
  }
}
```

CE: variable x might not have been initialized.

We get error because it is not guaranteed that all time command line arguments are passed if passed ok, if not x is not initialized & can't print

```
class Test{
  p s v m(String[] args){
    int x;
    if(args.length>0){
      x = 10;
    }else{
      x = 20;
    }
    sop(x);
  }
}
```

Java Test ✓ A B ↵
o/p 10

Jar Test ↵
o/p 20

→ It is never recommended to initialize local variables in if, else block logical

→ It is always recommended to initialize local variable at the time of declaration, atleast with default values.

↳ because there is no guarantee for execution of these blocks always at run time.

```
class Test {
    int x = 10;        — instance variable
    static int y = 20; — static variable
    p s v main(String[] args){
        int z = 30;
    }
}
```

static / instance	acce
public	anywh
private	within clas
default	Current packa
protected	acces packa but ou packag wont h in child clas

→ The only applicable modifier for local variables is final.

→ By mistake if we are trying to apply any other modifer we will get compile time error.

```
class Test {
    p s v m(String[] arg) {
        public / private / protected / static / transient / volatile int x = 10; ✗
        final int x = 10; ✓
    }
}
```

CE: illegal start of expression.

```
class Test {
    int x = 10;
    static int y = 20;
    p s v m (String[] args) {
        int z = 30;
    }
}
```

If we are not declaring with any modifier then by default it is default but this rule is applicable only for instance & static variables but not for local variables.

Conclusions:

1) For instance & static variables JVM will provide default values & not req. to perform initialisation explicitly.

2) But for local variables JVM will not provide default values we should perform initialisation explicitly before using variable.

3) Instance & static variables can be accessed by multiple threads & hence they are not thread safe.

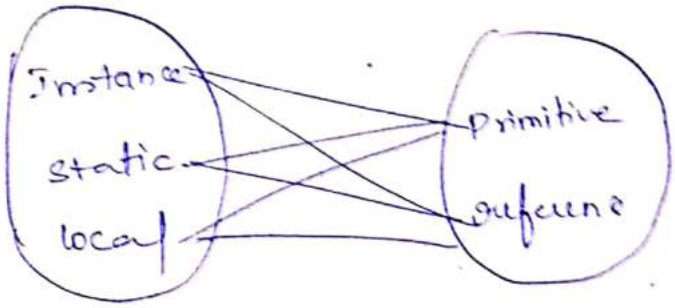
4) In case of local variables for every thread a seperate copy is created hence local var. are thread safe.

Type of variable	is Thread-safe?
Instance "	×
Static "	×
local "	✓

→ Every var in java should be instance, static or local.

→ Every var in should be either primitive or reference.

Hence various possible combinations of variables in Java




```
class Test {
    int x=10;          ——→ instance – primitive
    static String s="sini";  → static – reference
    P S V m(S[] args){
        int[] y = new int[3];  → local – reference
    }
}
```

~~Uninitialised~~ uninitialised array

```
class Test {
    int[] x;
    P S V m(SS[] args){
        Test t = new Test();
        sop(t.x);      o/p – null
        sop(t.x[0]);   x
    }
}
```

R.t: ~~Null~~ null point exception

I. Instance level

① `int[] x;`
`sop(obj.x);` null
`sop(obj.x[0]);` R.t: NPE

② `int[] x = new int[3];`
`sop(obj.x);` [I@3e25a5
`sop(obj.x[0]);` 0

II. static level

① static int[] x;

sop(x); null

sop(x[0]); RE: NPE

② static int[] x = new int[3];

sop(x); [I@3e25a5

sop(x[0]); 0.

III local level

① int[] x;

sop(x); } CE: variable x might not have been initialized

sop(x[0]);

int[] x = new int[3];

sop(x); [I@3e25a5

sop(x[0]); 0

⇒ Once we create an array every array element by default initialized with 0. irrespective of whether it is instance/static or local:

Var-arg methods [variable number of argument methods].

sum(10,20);
sum(10,20,30);
sum(10,20,30,40);
sum(10,20,30,40,50);

sum(int... a)

```
p s v sum(int a, int b){
    sop(a+b);
}
p s v sum(int a, int b, int c){
    sop(a+b+c);
}
p s v sum(int a, int b, int c,
          int d){
    sop(a+b+c+d);
}
```

We need not write diff. function so we replace by only one function!

Until 1.4 ver we can't declare a method with variable no. of arg.

If there is a change in no of arg. compulsory we should go for new method it ↑ length of the code & reduces readability. To overcome this problem we have var-arg methods in 1.5V.

According to this we can declare a method which can take ~~variable~~ any no. of arg such type of methods are called as var-arg methods

```
m1(int... x)
```

we can declare varargs method as this

↳ we can call this method by passing any no. of value

```
m1();  ✓
m1(10);  ✓
m1(10,20);  ✓
m1(10,20,30);  ✓
m1(10,20,30,40)  ✓
```

```
class Test{
    p s v m1(int... x){
        sop("vara arg method");
    }
    p s v m(String[] args){
        m1();
        m1(10);
        m1(10,20);
        m1(10,20,30,40);
    }
}
```

o/p

```
vara-arg method
"  "  "
"  "  "
"  "  "
```

m1(int... x) → int[] x . It will be treated to 1D array.

m1(10, 20, 30, 40);

int
sop(x.length); o/p 4.

Internally var-arg parameter will be converted into 1D array hence within var-arg method we can differentiate values by index.

```
class Test{
  p s v m(String[] args){
    sum();
    sum(10,20);
    sum(10,20,30);
    sum(10,20,30,40);
  }
  p s v sum(int... x){
    int total=0;
    for(int x1 : x){
      total = total + x;
    }
    sop(total);
  }
}
```

o/p 0
30
60
100

m1(int... x) ✓
m1(int ...x) ✓ Case I:
m1(int...x) ✓
m1(int x...) ×
m1(int. ..x) ×
m1(int .x..) ×

Case II: we can mix var-arg parameter with normal parameter

m1(int x, int... y) ✓
m1(String s, double... y) ✓

m1(double... d, String s) ×

m1(char ch, String... s) ✓

Case II: If we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter.

m1(double... d, String s) ✗

m1(char ch, String... s) ✓

m1(int... x, double... d) ✗

Case IV :- Inside var-arg method we can take only one var-arg parameter & we can't take more than one var-arg parameter.

```
class Test{
    p s v m(int... x){
        sop("int...");
    }
    p s v m1(int[] x){
        sop("int[]");
    }
}
```

ID ⇒ m1(int[])

⇒ m1(int[])

cannot declare both m1(int[]) & m1(int...) in Test

Case V: Inside a class we can't declare var-arg method and corresponding 1D array method simultaneously otherwise we will get compile time error.

```
class Test{
    p s v m1(int... x){
        sop("var-arg method");
    }
    p s v m1(int x){
        sop("General method");
    }
    p s v m(String[] args){
        m1(); // var-arg method
        m1(10,20); // var-arg method
```

m1(10); General method.

Case iv:

→ In general var-arg method will get least priority i.e if no other method matched then only var-arg method will match. It is exactly as default case inside a switch

Equivalence b/w var-arg parameter & one-dimensional array:

m1(int[] x)	m1(int... x)
m1(new int[]{10});	✓
m1(new int[]{10,20});	✓
m1(new int[]{10,20,30});	✓
	m1(); ✓
	m1(10); ✓
	m1(10,20,30); ✓

m1(int[] x) ⟹ m1(int... x) ✓

main(String[] args) ⟹ main(String... args)

→ Wherever 1D array is present we can replace with var-arg parameter.

→ Wherever var-arg parameter present we can't replace with 1D array.

m1(int... x) ⟹ m1(int[] x) ✗

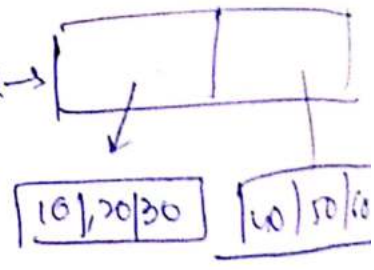
m1(int... a)	m1(int[] x)
m1();	x
m1(10);	x
m1(10,20,30);	x
m1(new int[]{10,20});	✓

1. m1(int... x) => int[] x
2. m1(String... x) => String[] x
3. m1(int[]... x) => int[][] x
4. m1(int[][]... x) => int[][][] x .

We can call this method by passing a group of int values & x will become 1D array.

We can call this method by passing a group of 1D int arrays & x will become 2D int array.

```
class Test{
    p s v m(String[] args){
        int[] a = {10,20,30};
        int[] b = {10,50,60};
        m1(a,b);
    }
    p s v m1(int[]... x){
        for(int[] x1 : x){
            sop(x1[0]);
        }
    }
```



o/p
10
10 ✓

Main.

class Test {

}

Javac Test.java

Java Test

RE : NoSuchMethodError : main.

whether the class contain main method or not & whether main method is declared acc. to req. or not these things wont be checked by compiler. at runtime JVM is responsible to check these things.

If JVM unable to find main method we get runtive error : NO such method error : main

public static void main (String[] args)

- public → To call by JVM from anywhere
- static → without existing object also JVM have to call main method
- void → main() method wont return anything to JVM
- main → This is the name which is configured inside JVM
- String[] args → Command line arguments.

→ At runtime JVM always searches for main method with the above prototype

→ The above syntax is very strict & if we perform any change then we will get runtime exception saying

NoSuchErrorMethod : main.

① static public

② main(String[] args) ✓
main(String .[]args) ✓
main(String args[]) ✓

③ main(String[] $in) ✓

④ main(String... args) ✓

Even though the syntax is very strict the above changes are acceptable. Instead of public static we can use static public i.e order of modifiers is not important

→ declares string arg in any acceptable format.

→ ~~Main~~ We can declare main method with the following modifiers.
- final
- synchronized
- strictfp.

```
class Test {
    static final synchronized strictfp public void
                                    main(String... $in){
        SOP("valid");
    }
}
```

o/p valid. ✓

Which of the following main method declarations are valid.

p s v m (String args) ×

p s v Main (String[] args) ×

public void main (String[] args) ×

p s int m (String[] args) ×

final synchronized strictfp p v m(String[] args) ×

final " " " static v m(String[] args)

public static void main (String... args) ✓

→ In which of the above we get compile time error

we won't get CE anywhere, but except last two cases in remaining we will get runtime exception saying : no such main

Case I :-

overloading methods

```
class Test {
    p s v m (String[] args) {
        SOP ("String[]");
    }
    p s v main (int[] args) {
        SOP ("int[]");
    }
}
```

o/p : String[].

→ overloading of the main method is possible but JVM will always call String[] args method only the other overloaded method we have call explicitly like normal method call.

P.Java

```
class p{
  p s v m (string[] args){
    sop("parent main");
  }
}
class c extends p{

}
```

Javac P.Java

Java P ↵

o/p : parent main

Jar c ↵

o/p : parent main

→ Inheritance concept applicable for main method hence while executing child class if a child doesn't contain main method then parent class main method will be executed.

```
class p{
  p s v m(string[] args){
    sop("parent main");
  }
}
class c extends p{
  p s v m(string[] args){
    sop("child main");
  }
}
```

It is method hiding but not overriding

Java P ↵

o/p : parent main

Java c ↵

o/p : child main

1.7 enhancements w.r.t main method. 21/1/20.

class Test {

}

1.6 V

javac Test.java ✓

java Test ↵

R.E : NoSuchMethodError: main

1.7 V

javac Test.java ✓

java Test ↵

R.E: Main method not found in class Test, please define main method as

p s v m (String[] args)

---

class Test {

static {

sop("Static Block");

}

}

1.6 V

javac Test.java ✓

java Test ↵

o/p: Static Block

R.E : NoSuchMethodError: main

1.7 V

javac Test.java ✓

java Test ↵

R.E: Main method not found in class Test, please define the main method as

p s v m (String[] args)

---

class Test {

static {

sop("Static Block");

System.exit(0);

}

}

1.6 V

javac Test.java ✓

java Test ↵

o/p: Static Block

1.7 V

javac Test.java ✓

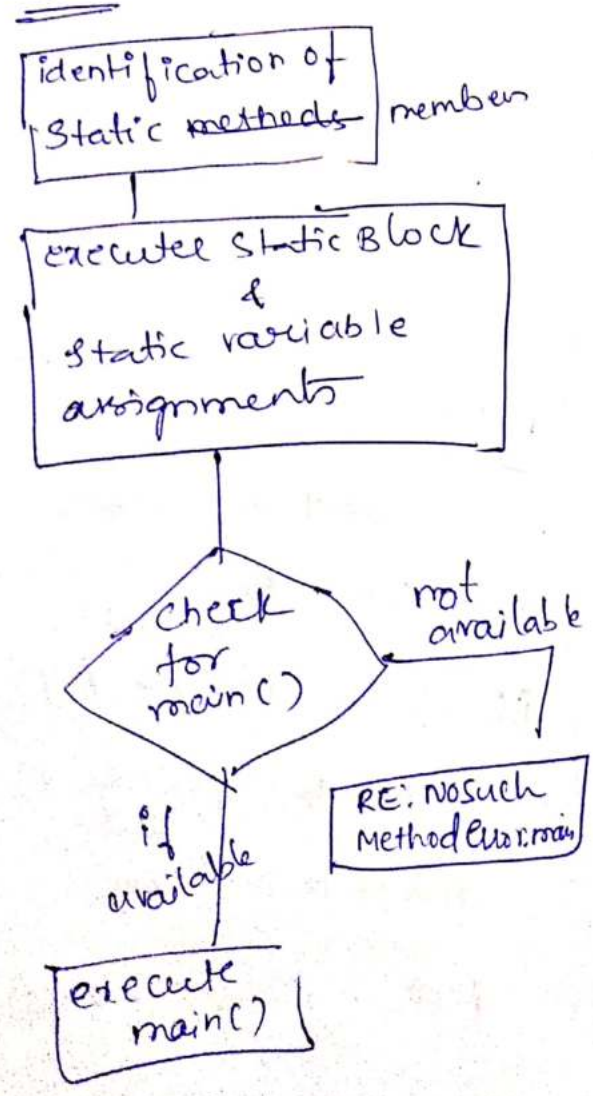
java Test ↵

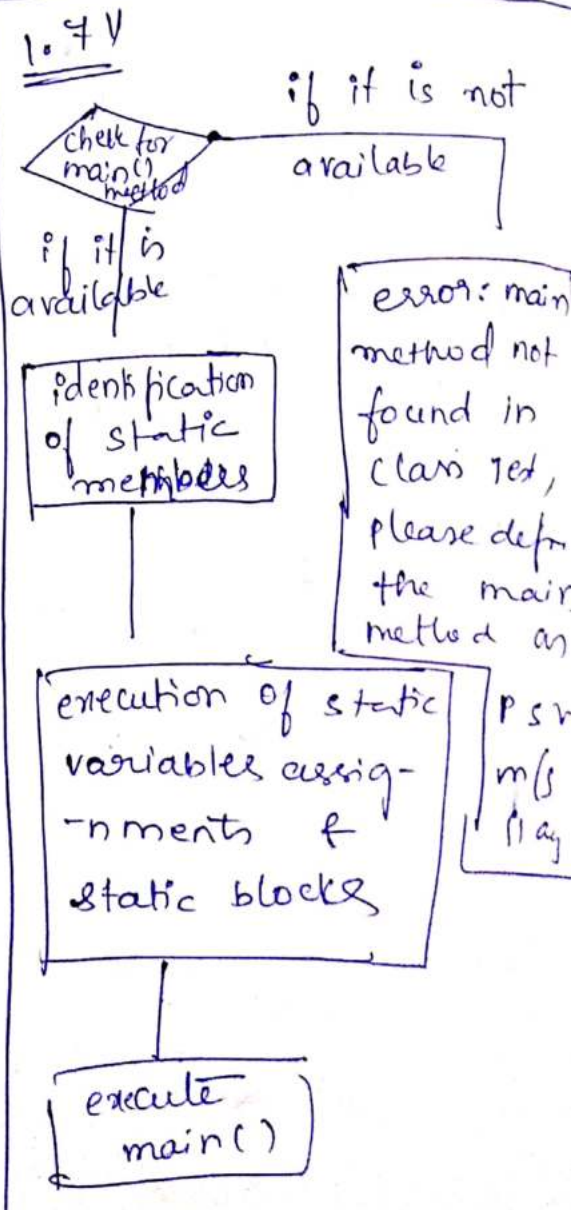
RE: main Meth not found in Class Test, Plase define the main method as

p s v m (String[] args).

```
class Test {
    static {
        sop("static Block");
    }
    p s v m(S[] args){
        sop("main method");
    }
}
```

1.6 V	1.7 V
Javac Test.Java ✓ Java Test ↵ o/p: static Block main method	Javac Test.Java Java Test ↵ o/p: static Block main method.





- without writing main method is it possible to print some statements to the console?

Ans:- yes, by using static block but this rule is applicable to 1.6 V only, from 1.7 V it is impossible to print statements to console.

Command line arguments

The arguments which are passed from command prompt are called CLA. With this arg JVM will create an array B by passing that array as arg JVM will call main method.

Java Test ↑ B C ↵

args[0] args[1] args[2]

args.length = 3

→ The main objective of command line arg is we can customise behaviour of main method.

Case I:

```
class Test{
  P S V m(S[]a){
    for(int i=0; i<=args.length; i++){
      SOP(args[i]);
    }
  }
}
```

if we replace <= with < then we won't get any exception

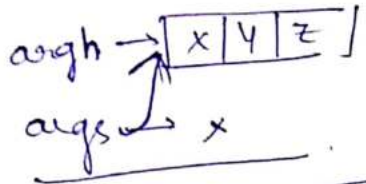
Java Test A B C

A
B
C
RE: AIOOBE

Java Test ↵

RE. AIOOBE

```
class Test {
  P S V m() {
    String[] argh = {"x", "y", "z"};
    args = argh;
    for(String s : args) {
      SOP(s);
    }
  }
}
```




```
Java Test A B C
x
y
z
Java Test A B
x
y
z
Java Test
x
y
z
```

Case I

Java Test 10 20

Case III:

```
class Test {
  P S V m(String[] args) {
    SOP(args[0] + args[1]);
  }
}
```

o/p 1020

within main method command line arguments are available in string form. So in the above code it is concatenation

Case IV: ~~Space is~~

usually space itself is the seperator b/w command line arg.

→ If our command line arg itself space so we have enclose that " " within double coats.

```
class Test {
  P S V m(String a) {
    SOP(args[0]);
  }
}
```

Java Test "Note Book"

o/p Note Book

Java Coding Standards

package com.sini.scjp;

public class Calculator {

public static int add(int number1, int number2){

return number1 + number2;

}

p s v m( ) {

}

}

highly recommended for company.

Hi-tech city

giving package meaningful name.

Whenever we are writing java code it is highly recommended to follow coding standard, whenever we are writing any component its name should reflect the purpose of the component (functionality). The main advantage of this approach readability & maintainability of code is improved.

class A {

public int m1(int x, int y) {

return x+y;

}

}.

Ameerpet standard,

Coding standards for classes :-

→ class name should start with uppercase character & every inner word should start with uppercase character.

eg String
StringBuffer
Account

Coding standards for Interface

- usually interface names are adjectives should start with uppercase character & if it contains multiple words every inner word also caps.

Runnable, Serializable, Comparable

Coding Standards for methods:

usually method names are either verbs or verb noun combination should start with lowercase if it contains multiple every inner word should be caps it is called camel case convention.

getName()
setSalary()
— verb — noun

print()
sleep() => verb
run()
eat()
start()

Coding standards for variable:

name
age
salary
mobileNumber

→ start with lowercase & upper case. (they are nouns & start with it is camel case convention)

Coding standards for constants:

MAX_VALUE
MAX_PRIORITY
NORM_PRIORITY
MIN_PRIORITY
PI

=> Noun

usually constant names are nouns should contain only upper case character & if it contains multiple words these words are seperated with _ symbol

we can declare constants with public static & final modifiers.

Java Beans Coding Standards:

→ A Java Bean is a simple Java class with Private properties & public get and set methods.

```
public class StudentBean {
    private string name;
    public void setName (string name) {
        this.name = name;
    }
    public string getName() {
        return name;
    }
}
```

class name ends with 'Bean' is not official convention from SUN.

Syntax for set Method

→ It should be public method, the return type should be void, method name should be prefixed with set, it should take parameter / argument

Syntax for get Method

→ should be public,
return type should not be void,
method name prefixed with get,
it should not take any argument.

```
private boolean Empty;
public boolean getEmpty(){
    return Empty;
}
public boolean isEmpty(){
    return empty;
}
```

→ for boolean get method name can be prefixed with either get or is but recommended to use is

Coding standards for listeners

Case I: To register a listener.

```
public void addMyActionListener(MyActionListener l)
Public void registerMyActionListener(MyActionListener l) X
Public void addMyActionListener(ActionListener l) X
```

→ method should be prefixed with add

Case II: To unregister a listener

```
Public void removeMyActionListener(MyActionListner l)
Public void unregisterMyActionListener(MyActionListner l) X
public void removeMyActionListener(ActionListener l) X
public void delete " " " ( " " l) X
```

→ Method name should be prefixed with remove.

Operators & Assignments

Increment

pre-increment
y = ++x

post-increment
y = x++

pre-decrement
y = --x;

post dec
y = x--;

expression	Intial val of x	value of y	final val of x
y = ++x	10	11	11
y = x++;	10	10	11
y = --x;	10	9	9
y = x--;	10	10	9

```
int x = 10;
int y = ++x;
sop(y); //
```

```
int x = 10;
int y = ++10;
sop(y);
```

→ CE: unexpected type found: value req: variable

1. We can apply increment & decrement operators only for variables but not for constant. if we are trying to apply for constant values we will get compile time error.

```
int x = 10;
int y = ++(++x);
sop(y);
```

CE: unexpected type found: value req: variable.

2. Nesting of increment & decrement operators not allowed.

```
int x = 10;
x = 11;
sop(x);
```
o/p 11 ✓

```
final int x = 10;
x = 11;
sop(x);
```
CE: cannot assign a value to final variable.

```
int x = 10;
x++;
sop(x);
```
o/p: 11

```
final int x = 10;
x++;
sop(x);
```
CE: Cannot assign a value to final variable.

(3) for final variables we can't apply increment & decrement operators.

```
int x = 10;
x++;
sop(x);
```
o/p: 11 ✓

```
char ch = 'a';
ch++;
sop(ch);
```
o/p: b ✓

```
double d = 10.5;
d++;
sop(d);
```
o/p: 11.5 ✓

```
boolean b = true;
b++;
sop(b);
```
✗ CE: operator ++ cannot be applied to boolean

(4) we can apply increment & decrement operators for every primitive type except boolean

```
byte b = 10;
b = b+1;
sop(b);
```
CE:

```
byte b = 10;
b++;
sop(b);
```
o/p: 11

difference b/w b++ & b = b+1; ⟶

If we apply any arithmetic operator b/w two variables a and b the result type is always max(int, type of a, type of b)

eg: 1:
```
byte a = 10;
byte b = 20;
byte c = (a+b);   // int
sop(c);
```
CE: Possible loss of precision
found: int
required: byte.

max(int, byte, byte)

```
byte c = (byte)(a+b);
sop(c);
```
o/p = 30 ✓

eg: 2:
```
byte b = 10;
b = b+1;
sop(b);
```
CE: Possible loss of precision
found: int
required: byte.

max(int, byte, int)
⇓
int

```
b = (byte)(b+1);
sop(b);
```
o/p: 11

```
b++ =
b = (type of b)(b+1);
```

In case of increment & decrement operators internal typecasting is done internally.

Arithmetic Operators (+, -, *, /, %)

If we apply any arithmetic operator b/w two variables a and b the result type is always max(int, type of a, type of b);

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

byte → short
int → long → float → double
char

eg:- sop('a' + 'b'); (97 + 98)
o/p 195

sop('a' + 0.89);
o/p ; 97.89

sop(10/0); — RE : / by zero
sop(10/0.0); o/p Infinity

(int, int) → int

Infinity: In integral arithmetic (byte, short, int, long) there is no way to represent infinity. Hence if infinity is the result we will get arithmetic exception in integral arithmetic.

But in floating point arithmetic (float, double) there is a way to represent infinity for this float & double classes contains the following two constants.

POSITIVE_INFINITY;
NEGATIVE_INFINITY;

even though result is infinity, we wont get any arithmetic exception in floating pt arithmetic

```
sop(-10.0/0) ; — Infinity
sop(10/0.0) ; Infinity
```

o/p undefined

```
sop(0/0) ; (int, int, int) => int
RE  AE : / by zero.
sop(0.0/0) ; — double. NaN
```

NaN (Not a Number):

In integral arithmetic (byte, short, int, long) there is no way to represent undefined results hence if the result is undefined we will get runtime exception saying arithmetic Except : / by zero.

~~sop(0/0)~~ .

But in floating pt arithmetic (float, double) there is a way to represent undefined result. for this Float & Double classes contain NaN constant. Hence if the result is undefined we wont get any arithmetic exception in floating pt arithmetic.

```
sop(-0/0.0) ; NaN
sop(0.0/0) ; NaN
```

```
sop(10/0) ; RE : AE / by zero.
sop(10/0.0) ; Infinity
sop(0/0) ; RE : AE : / by zero
sop(0.0/0) ; NaN
```

```
Sop (10 < Float.NaN);   false
Sop (10 <= Float.NaN);  false
Sop (10 > Float.NaN);   false
Sop (10 >= Float.NaN);  false
Sop (10 == Float.NaN);  false
Sop (Float.NaN == Flat.NaN); false
Sop (10 != Float.NaN);  true
Sop (Float.NaN != Float.NaN); true
```

NOTE: for any x value including NaN the following expression returns false.

```
x < NaN
x <= NaN
x > NaN
x >= NaN
x == NaN
```

⟹ false

for any x value including NaN, the following expression returns true

```
x != NaN ⟹ true
```

Arithmetic Exception

- It is runtime exception but not compile time exception
- It is possible only in integral arithmetic but not in floating point arithmetic
- The only operator which cause AE are / and %.

String Concatenation Operator (+):

The only overloaded operator in Java is '+' operator

Sometimes it acts as arithmetic addition operator

& " " " String concatenation ".

```
String a = "Siri";
int b=10, c=20, d=30;
sop(a+b+c+d);
sop(b+c+d+a); 60Siri
sop(b+c+a+d); 30Siri30
sop(b+a+c+d); 10Siri2030.
```

a + b + c + d

"Siri10" + c + d

Siri1020 + d

Siri102030.

If atleast one argument is type is String then + ope acts as concatenation operator & if both arguments are number type. Then + ope acts as addition operator.

Operator overloading is possible only in this case in Java.

```
String a = "Siri";
int b=10, c=20, d=30;
```

① a = b+c+d; ✗ CE: incompatible types found: int req: java.lang.String

② a = a+b+c; ✓ o/p Siri1020

③ b = a+c+d; ✗ CE: incompatible types found: java.lang.String req: int

④ b = b+c+d; ✓

b = 60.

Relational operators (<, <=, >, >=)

97.0 < 97 (

97 > 65

```
sop(10 < 20); true
sop('a' < 10); false
sop('a' < 97.6); true
sop('a' > 'A'); true
sop(true > false); — CE: operator > cannot be applied to boolean, boolean
```

→ We can relational operators for every primitive type except boolean.

```
sop("durga123" > "durga"); ✗
```

CE: operator > cannot be applied to java.lang.String, java.lang.String

→ We can't apply relational operators for object types.

```
sop(10 < 20); true;
sop((10 < 20) < 30); CE: operator cannot be applied to boolean, int
```

true < 30

→ Listing of relational operators is not allowed otherwise we will get compile time error

Equality operators (==, !=)

```
sop(10 == 20); false
sop('a' == 'b'); false
sop('a' == 97.0); true
```

sop (false == false); true

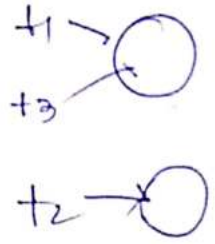
- we can equality operator for every primitive type including boolean also

- we can apply equality operator for object types also for object reference r_1 and r_2

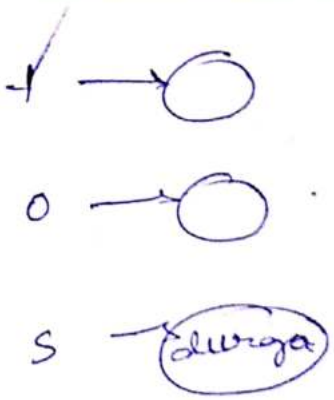
$r_1 == r_2$ return true if and only if both references pointing to same object.

reference comparision / address comparision

eg:-1 : Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
sop (t1 == t2); false
sop (t1 == t3); true



Thread t = new Thread();
Object o = new Object();
String s = new String("durga");
sop (t == o); false
sop (o == s); false
sop (s == t); → CE: incomparable types:
j.l.String & j.l.thread



thread & object parent & child
object & String parent & child
str & thread not comparable

→ If we apply equality operator for object types then compulsory there should be some relation b/w argument types either child to parent or parent to child, or same type otherwise we can get compile time error saying incomparable types.

** Difference b/w == operator & equals() method

== operator meant for reference comparison equals() meant for content comparison.

```
String s1 = new String("durga");
String s2 = new String("durga");
Sop(s1 == s2); false ✓
sop(s1.equals(s2));
        ↓
      true ✓
```

s1 → durga
s2 → durga

r == null ⟹ false in

NOTE: for any object reference r,

String s = new String("durga");

but null == null is always true;

sop(s == null); o/p: false

String s = null;
sop(s == null);
true

sop(null == null);
true.

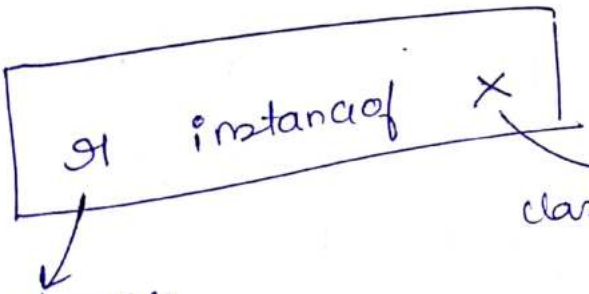
instanceof operator

to check

We can use 'instanceof' oper. whether the given object is of particular type or not.

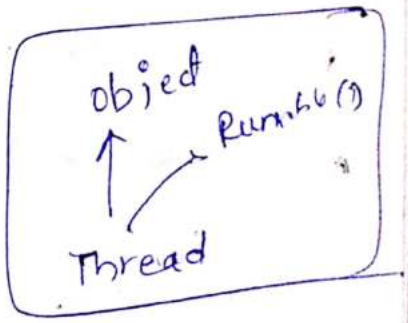
```
Object o = d.get(0);
if (o instanceof Student) {
    Student s = (Student) o;
    // perform student specific functionality
} else if (o instanceof Customer) {
    Customer c = (Customer) o;
    // perform customer specific functionality
}
```

Syntax:



eg

```
Thread t = new Thread();
sop(t instanceof Thread); true
sop(t instanceof Object); true
sop(t instanceof Runnable); true.
```



Eg 2: Thread t = new Thread();
sop(t instanceof String); CE: inconvertible types; found: j.l.Thread; required: j.l.String

→ To use instanceof operator there should some relation b/w argument type.

X,
null instanceof X ⇏ false

→ For any class/interface null instanceof X is always false.

sop(null instanceof Thread); false
sop(null instanceof Runnable); false
sop(null instanceof Object); false

Bitwise Operators (&, |, ^) (and, or, XOR)

& - AND => if both arg true returns true
| - OR => if atleast one arg true returns true
^ - XOR => both arg are diff returns true.

sop(true & false); false
sop(true | false); true
sop(true ^ false); true

sop(4 & 5); 4 → 100, 101 → 100
sop(4 | 5); 5 → 100, 101 → 101
sop(4 ^ 5); 1 → 100, 101 → 001

We can these operators for integral types also.

Bitwise Complement operator (~):

```
sop(~true); → CE: operator ~ cannot be
                  applied to boolean.

sop(~4);
```

We can apply ~ ope. only for integral types but not for boolean type, if we are trying to apply for boolean-type we get CE.

```
sop(~4); o/p -5
```

~ 100

011

so bits.

sign bit.

4 = 0 0000 . . . 0100 — value

~4 = 1 1111 . . . 1011 — 2's com

-ve. 0000 0100

+1

0000 0101 = 5

~4 = -5

→ The most significant bit acts as sign bit

0 means +ve
1 means -ve → +ve numbers will be represent directly in memory.

-ve number will be represented in the memory indirectly in 2's complement form.

Boolean Complement operator (!)

sop (!4); → CE: operator ! cannot be appl. to int.

sop (!-false); true;

we can apply this operator only for boolean not for integral types

=> Applicable for both boolean & integral types

~ => applicable for only integral but not for boolean type

! => applicable only for boolean but not for integral type.

22|1|20

* Short-Circuit Operators (&&, ||) exactly same as bitwise operators except the following table.

&, |	&&, ||
① Both arguments should be evaluated always	① Second arg. evaluation is optional
② Relative performance is low	② Relative performance is high
③ Applicable for both boolean & integral types	③ Applicable only for boolean not for integral

* x && y => y will be evaluated iff x is true. i.e., if x is false then y wont be evaluated.

* x || y => y will be evaluated if x is false i.e y will not be evaluated if x is true.

```
int x =10, y =15;
if (++x <10 & ++y >15){
        x++;
}
else{
    y++;
}
sop(x + ". " +y);
```

	x	y
&	11	17
&&	11	16
|	12	16
||	12	16

```
int x =10;
if (++x <10 && (x/0 >10)){
    sop("Hello");
}
else
{ sop("Hi");
}
```

① CE
② RE : AE: / by 0
③ Hello
④ Hi ✓

in first case 11 < 10 is false so x/0 is not evaluated so it will come to else part so the else is evaluated.

→ If we replace && with & then will get RE : AE / by 0.

Type-cast Operator :

There are 2 types of type-casting

1. Implicit Type-casting
2. explicit Type-casting

1. Implicit typecasting:

```
int x = 'a';
sop(x);   97
```
[compiler converts char to int automatically by Implicit typecasting]

```
double d = 10;
sop(d);  10.0
```
[compiler converts int to double automatically by Implicit typecasting]

① Compiler is responsible to perform implicit typecasting.

② Whenever we are assigning small data type value to bigger data type variable implicit typecasting will be performed.

③ It is also known as widening or upcasting.

④ No loss of info in this typecasting.

The following are various possible conversion where implicit type casting will be performed.

byte → short → int → long → float → double.

char → int

2. Explicit typecasting:

```
int x = 130;
byte b = x;   - CE: PLP
                found: int
                req: byte
```

```
int x = 130;
byte b = (byte)x;
sop(b);  o/p: -126
```

① Programmer is responsible to perform explicit typecasting.

② Whenever assigning big data type to small data type variable explicit typecast req.

③ also known as narrowing (or) downcasting

④ Chance of loss of info.

byte ← short ← int ← long ← float ← double
char ↙

L → R ⇒ Implicit type conversion
R → L ⇒ explicit " "

Whenever we are using bigger dT value to smaller by explicit typecasting the MSB will be lost. we have to consider only LSB bits.

int x = 130 ⇒ 0000 . . . 0 10000010

byte b = (byte)x ⇒ 10000010 — 8 bits

MsB
-ve

2's compl

2 | 130
2 | 65 — 0
2 | 32 — 1
2 | 16 — 0
2 | 8 — 0
2 | 4 — 0
2 | 2 — 0
2 | 1 — 0

1111101
+1
11111110.

$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \times 0$

= 126.

so ans −126

short — 16 bits

int x = 150;
short s = (short)x;
sop(s); 150
byte b = (byte)x;
sop(b);

150 ⇒ 000 . . 01001011 0

0000 01001011 0
+ve 150.

byte b = (byte)x ⇒ 1 0010110

2's comp

-ve 1101001
+1
11010110.

$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

= −106.

eg:-

```
double d = 130.456;
int x = (int)d;
Sop(x); o/p: 130 ✓
byte b = (byte)d;
Sop(b); o/p : -126 ✓
```

If we apply floating type ~~data type~~ value to integral types by explicit typecasting the digits after . will be lost.

22/1/20

Assignment Operators:

① Simple: int x = 10;

② chained: a = b = c = d = 20;

③ compound: a += 20;

```
int a,b,c,d;
a = b = c = d;
sop(a + " " + b + " " + c + " " + d); //p 20 20 20 20
```

```
int a = b = c = d = 20;
```

CE: cannot find symbol.
symbol: variable b
location: class Test.

- we cant perform chained assignment directly at the time of declaration.

```
int b,c,d;
int a = b = c = d = 20;
```

Compound:

```
int a = 10;
a += 20;
Sop(a) = 30.
```

+=	&=	>>=
-=	|=	>>>=
*=	^=	<<=
/=		
%=		

total 11.

only possible compound operators.

x >> 2

1 1 0 1 0 1 1 1

Sign bit ⇒ >>

only with zero ⇒ >>>.

```
byte b = 10;
b = b+1;
sop(b);
```

invalid

max(int, byte, int)

int.

CE: P C P

found: int

req: byte

```
byte b = 10;
b++;
sop(b);
```

o/p: 11

b = (byte)(b+1);

```
byte b = 10;
b += 1;
sop(b);
```

sop(b);

11

in compound operator internally typecasting is performed.

```
byte b = 127;
b += 3;
sop(b);
```

-126

```
int a,b,c,d;
a = b = c = d = 20;
a += b -= c *= d /= 2;
sop(a+" "+b+" "+c+" "+d);
```

o/p: -160 -180 200 10

d = d/2

d = 20/2

d = 10

c = c*d

= 20*10

= 200.

b = b - c

= 20 - 200

= -180

a = a + b

= 20 - 180

= -160.

Conditional operator (?:)

conditional operator is ternary operator.

```
int a = (10 < 20) ? 30 : 40;
```

o/p: 30.

```
int a = (10 > 20) ? 30 : ((40 > 50) ? 60 : 70)
sop(a);
```

70.

a++; } unary operator

++a;

a+b; } binary operator

The only possible ternary operator in java is conditional operator.

new operator:

We can use new operator to create object.

```
Test t = new Test();
```

Note:

- After creating an object constructor will be executed to perform initialisation of object.
- Hence constructor is not for creation of object & it is for initialisation of object.
- In Java we have only new keyword but not delete keyword because destruction of useless objects is the responsibility of garbage collector.

[] operator:

- We can use this operator to declare & create array.

```
int[] x = new int[10];
```

Java Operator Precedence:

1. Unary Operators:
   [], x++, x--
   ++x, --x, ~, !
   new, <type>
2. Arithmetic
   *, /, %
   +, -
3. Shift
   >>, >>>, <<
4. comparison operators:
   <, <=, >, >=, instanceof
5. equality
   ==, !=

6. Bitwise operators:

&
^
|

7. Short Circuit operators:

&&
||

8. Conditional operator

?:

9. Assign

~~Any~~

==, +=, -=, *=, ...

evaluation order of Java operands.

```
class Test
{
  p s v main(String[] args){
    sop(m1(1) + m1(2) * m1(3) / m1(4) + m1(5) * m1(6));
  }
  p s int m1(int i){
    sop(i);
    return i;
  }
}
```

32.

1 2 3 4 5 6

1 + 2 * 3 / 4 + 5 * 6

1 + 6/4 + 5 * 6

1 + 1 + 5 * 6

1 + 1 + 30

2 + 30

32.

→ In Java we have only operator precedence but not operand precedence. Before applying any operator all operands will be ~~precedence~~ evaluated from left to right.

① new vs newInstance()
② instanceof vs isInstance()
③ classNotFoundException vs NoClassDefFoundError

① new vs newInstance()

new:

```
Test t = new Test();
Student s = new Student();
Customer c = new Customer();
```

```
class Test {
p s v m(String[] arg) throws Exception {
    Object o = Class.forName(args[0]).newInstance();
    sop("Object created for "+" "+o.getClass().getName());
}
}
```

```
java Test Student ↵
o/p  Object created for : Student.

java Test java.lang.String ↵
o/p  Objected created for : java.lang.String.
```

→ We can use new operator to create an object if we know class name at the beginning.

→ newInstance() is a method present in class Class. video 24:00 min. chr 23 vide

We can use newInstance() to create Object if we don't know class name at the beginning and it is available dynamically at run time.

In the case of new operator based on our req. we can invoke any constructor.

```
Test t1 = new Test();
"  t1 = "  "  (10);
"  t2 = "  "  ("durga");
```

But newInstance() method internally calls no-argument constructor hence to use newInstance() compulsory corresponding class should contain no-arg. constructor otherwise we will get runtime error: Instatiation exception.

```
Test t1 = new Test();
```

at runtime if Test.class is not there will get R.E : NoClassDefFoundError : Test.

available then

```
Object o = class.forName(args[0]).newInstance();
```

Java Test Test123 ↵

RE: ClassNotFoundException : Test123.

while using newInstance() at runtime if corresponding .class not there we get RE: classNotFoundException : Test123.

new	newInstance()
① operator in Java	① method present in java.lang.Class
② we can use new operator to create object if we know classname at the beginning	② we can use this method to create object if we don't know class name at the beginning and it is available dynamically at runtime.

⑨ To use new operator class not req to contain no-arg constructor

⑩ To use newInstance() compulsory class should contain no-arg constructor otherwise we will get RE: InstantiationException

⑪ At runtime if class file not available then we get RE: NoClassDefFoundError

⑫ RE: classNotFound Exception: ;

*** ClassNotFoundException Vs NoClassDefFoundError.

→ For loadcoded class names, at run-time if the corresponding .class not available then we will get runtime exception saying NoClassDefFoundError, which is unchecked.

Test t = new Test();

At runtime if Test.class not available then we will get runtime excep: say

NoClassDefFoundError: Test.

→ For dynamically provided class names, at runtime if the corresponding .class not available then we will get runtime exception saying ClassNotFoundException, which is checked exception.

Object o = class.forName(args[0]).newInstance();

Java Test Student ↵

At runtime if Student.class not available we get RE: classNotFoundException: student ...

instanceof vs isInstance().

```
Thread t = new Thread();
sop(t instanceof Runnable);
```

sop(t instanceof Object)

instanceof is an operator in Java.
- we can use instanceof to check whether the given Object is of particular type or not & we know the type at the beginning.

```
class Test
{
  p s v m(String[] args) throws Exception {
      Thread t = new Thread();
      sop(Class.forName(args[0]).isInstance(t))
  }
}
```

Java Test Runnable ↵ o/p true
Java Test String ↵ o/p false.

isInstance() method present in java.lang.Class.
- we can use isInstance() to check whether the given Object is of particular type or not & we know the type at the beginning & it is available dynamically at runtime.

isInstance() is method equivalent of instanceof operator.

Flow-Control

1. Selection statements
   - if else
   - switch()
2. Iterative statement
   - while
   - do-while();
   - for()
   - foreach().
3. Transfer statements
   - break
   - continue
   - return
   - try-catch-finally
   - assert

flow control describes the order in which the statements will be executed at runtime.

Selection statements

- if else.

```
if (b) {
    Action if b is true
} else {
    Action if b is false
}
```

→ boolean if any other type we get CE.

```
int x = 0;
if (x) {
    sop("Hello");
} else {
    sop("Hi");
}
```

CE: incompatible types: found: int req: boolean

```
int x = 10;
if (x = 20) {
    sop("Hello");
} else {
    sop("Hi");
}
```

CE.

```
int x = 10;
if (x == 10) {
    sop("Hello");
} else {
    sop("Hi");
}
```

o/p : Hi.

```
boolean b = true;
if (b = false){
  sop("Hello");
}
else
{
sop("Hi");
}
```

o/p : Hi

```
boolean b = false;
if (b == false){
  sop("Hello");
}
else {
  sop("Hi");
}
```

o/p: Hello

Here assignment operator is given but the type is boolean so it is acceptable.

Here b == false is true so if is executed.

```
if (true)
  sop("Hello");
```
✓

```
if (true);
```
✓

```
if (true)
int x = 10;
```
✗ CE: not allowed to declare here.

```
if (true){
  int x=10;
}
```
✓

~~if~~ { int x = 10;;;;;;;;;;

else part & { } are optional.

- without { } only one statement is allowed under if which should not be declarative statement.

NOTE: There is no dangling else problem in java.

every else is mapped to the nearest if statement.

```
if (true)
  if (true)
    sop("Hello");
else
  sop("Hi");
```

here `if (true) sop("Hello");`

the else belongs to the above if even though while writing it is under if (true)

Switch

If several options are available it is not recommended to use if else it reduces readability to handle this only we should go for switch.

switch(x)
{
}

1.4V
byte
char
allowed short
arg. int
types
-2147483648

boolean
long
float
double
not allowed

1.4 V	1.5 V	1.7 V
byte	Byte	
Short	Short	String
Char	Character	
int	Integer	
	enum	

enum corresponding wrapper classes also allowed with

- { } braces are mandatory.
- except switch { } are optional.
- both case and default are optional i.e an Empty switch case is valid Java syntax.

```
int x=10;
switch(x){
}
```

```
int x = 10;
switch(x)
{
sop("Hello");
}
```

CE: case, default, or } expected

- Inside a switch every statement should be under some case or default independent cases are not allowed otherwise we will get compile time error.

```
int x = 10;
int y = 20;
switch(x){
  case 10:
    sop("Hello");
  case y:
    sop("20");
}
```
✗ CE: constant expression required

```
int x = 10;
final int y = 20;
switch(x){
  case 10:
    sop("10");
  case y:
    sop("20");
}
```
✓

every case label should be compile time const i.e constant expression

→ if declare y as final we won't get any compile time error.

→ Both switch arg & case label can be expression but case label should be constant expression.

```
int x = 10;
switch(x+1)
{
case 10:
  sop(10);
  break;
case 10+20+30:
  sop(60);
}
```
✓

```
byte b = 10;
switch(b)
{
case 10:
  sop(10);
  break;
case 100:
  sop(100);
  break;
case 1000:
  sop(1000);
}
```
b = -128 to 127

CE: P LP found: int req: byte

```
byte b = 10;
switch(b+1)
{
case 10:
  sop(10);
  break;
case 100:
  sop(100);
  break;
case 1000;
  sop(100);
}
```
✓

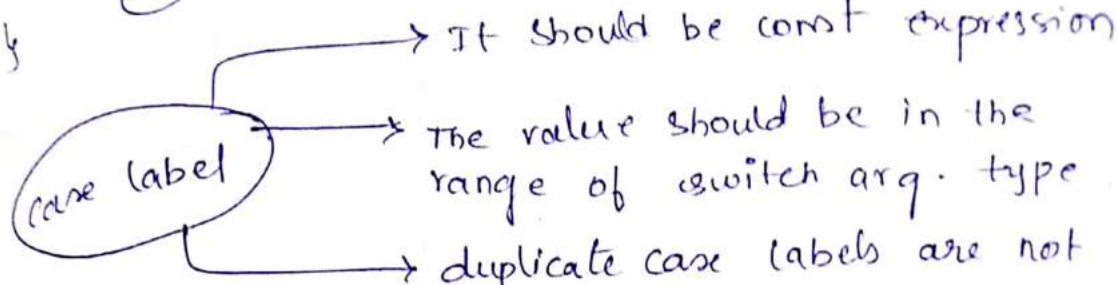
but no where relation or not assign so perfect alright

- Every case label should be in the range of switch argument type. or we get CE.

```
int x = 10;
switch(x){
  case 97:
  case 98:
  case 'a':
}
```

CE: duplicate case label

duplicate case labels are not allowed otherwise we will get compile time error



Fall-through inside switch

```
switch(x)
{
case 0:
  sop(0);
case 1:
  sop(1);
  break;
case 2:
  sop(2);
default:
  sop("def");
}
```

x = 0	x = 1
0	1
1	
x = 2	x = 3
2	def
def	

within in a switch if any case is matched from that case onwards all statements will be executed until break or end of switch. this called fall-through

The main adv of fall through is we can define common action for multiple cases. (Code reusability).

```
switch(7)
{
case 1:
case 2:
case 3:
  sop("0-4");
  break;
case 4:
case 5:
case 6:
  sop("4-1");
  break;
}
```

- within the switch we can take default case atmost once
- default case executed if no other case matched.
- within switch we can write default anywhere.
- but recommended to write at last.

```
switch(x){
  default:
    sop("def");
  case 0:
    sop(0);
    break;
  case 1:
    sop(1);
  case 2:
    sop(2);
}
```

x=0	x=1	x=2	x=3
0	1	2	def
	2		0

Iterative statements.

① while():

If we don't know no. of iterations in advance then we should go for while loop.

```
while(scs.next()) .  while(e.hasMoreElements()){
{
  ;
}
                     while(itr.hasNext()){
                     }
```

```
while(b)
{
  Action.
}
```

The arg should be boolean type if we are trying to provide any other type we get CE:

```
while(i){
  sop("Hello");
}
```

→ CE: incompatible types
found: int
required: boolean.

```
while(true)
sop("Hello");
```
✓

```
while(true);
```
✓

```
while(true)
int x=10;
```
✗

```
while(true)
{
int x=10;
}
```
✓

```
while(true)
{
sop("Hello");
}
sop("Hi");
```
✗ CE: unreachable statement

```
while(false)
{
sop("Hello");
}
sop("Hi");
```
✗ CE: unreachable statement

```
int a=10, b=20;
while(a<b){
sop("Hello");
}
sop("Hi");
```
✓ o/p Hello Hello ... infinite no. of time

```
int a=10, b=20;
while(a>b)
{
sop("Hello");
}
sop("Hi");
```
✓ o/p: Hi

```
final int a=10, b=20;
while(a<b)
{
sop("Hello");
}
sop("Hi");
```
✗ CE: unreachable statement

```
final int a=10, b=20;
while(a>b)
{
sop("Hello");
}
sop("Hi");
```
✗ CE: unreachable statement.

①
```
final int a=10;
int b=20;
sop(a);
sop(b);
```
after compilation →
```
sop(10);
sop(b);
```

②
```
final int a=10, b=20;
int c=20;
sop(a+b);
sop(a+c);
sop(a<b);
sop(a<c);
```
after compil →
```
sop(30);
sop(10+c);
sop(true);
sop(10<c);
```

→ every final variable will be replaced by the value at compile time only.

→ If every arg in a final variable (compile time const) then that operation should be performed at compile time only.

---

do while: → If we want to execute loop body atleast once we go for do while.

```
do {
   body
} while(b);
```

↓ should be boolean.

; is optional in C++ but mandatory in Java.

```
do
sop("Hello");
while(true);
```
✓

```
do;
while(true);
```
✓

```
do
int x=10;
while(true);
```
✗

```
do {
  int x = 10;
} while(true);
```
✓

```
do
while(true);
```
✗

```
do while(true)
sop("Hello");
while(false);
```
✓

```
do
  while(true)
    sop("Hello");
while(false);
```
✓

o/p: ~~Hello~~ Hello Hello ...

```
do
{sop("Hello");
}while(true);
sop("Hi");
```
✗ CE: unreachable statement

```
do {
  sop("Hello");
} while(false);
sop("Hi");
```
✓ o/p: Hello Hi

```
int a=10, b=20;
do{
  sop("Hello");
}while(a<b);
sop("Hi");
```
✓ Hello Hello ... infinite no. of times (a=10, b=20, true)

```
int a=10, b=20;
do{
sop("Hello");
}while(a>b);
sop("Hi");
```
✓

o/p Hello
Hi

```
final int a=10,b=20;
do{
sop("Hello");
}while(a<b);
sop("Hi");
```
✗

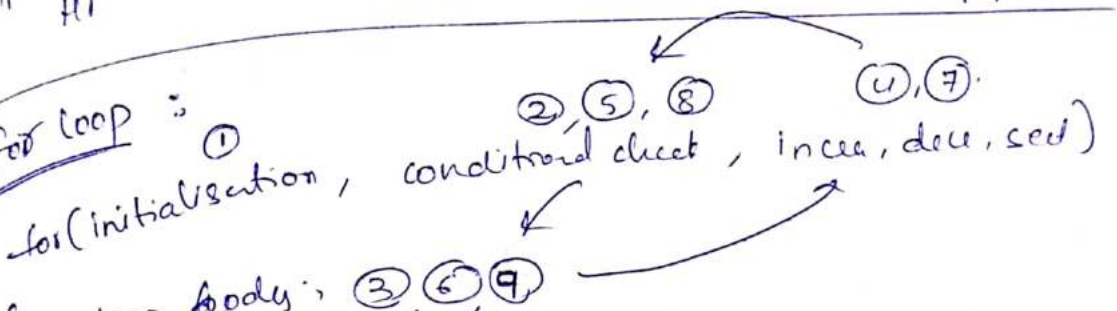
CE: unreachable statement

```
final int a=10,
      b=20;
do{
sop("Hello");
}while(a>b);
sop("Hi");
```
✓

o/p : Hello
Hi

---

for loop :



If we know no. of iteration in advance for loop is best choice.

```
for(i=0; true; i++)
  sop("Hello");
```
✓

```
for(i=0; i<10; i++);
```
✓

```
for(i=0; i<10;
    i++)
  int x=10;
```
✗

Initialization section

this part will be executed only once in loop life cycle

here we can declare and initialise local variables of for loop.

```
int i=0, j=0;
```
✓

```
int i=0, string s="durga";
```
✗

```
int i=0, int j=0;
```
✗

→ here we can declare any no. of variables but should of same type.

→ if diff data type variables declare then CE.

```
int i=0;
for(sop("Hello"); i<3; i++){
    sop("Hi");
}
```

o/p :
Hello
Hi
Hi
Hi

---

in the initialisation section we can type any valid java statement including sop.

```
for(break|return ; i<3; i++){
    sop("Hi");
}
```

CE: illegal start of expression

Conditional check:

```
for(int i=0; i<3; i++){
}
```

- it is optional.
- if not taking anything compiler will take true.
- any valid java expp. but of type boolean

---

increment-decrement section

```
int i=0;
for(sop("Hello"); i<3; sop("Hi")){
    i++;
}
```

o/p
Hello
Hi
Hi
Hi

any valid Java statement including sop.

```
for(;;);
```

```
for(;;)
{
sop("Hello");
}
```

infinite no. of times.

→ all 3 parts of for loop are independent of each other & optional.

```
for(i=0; true; i++)
{
sop("Hello");
}
sop("Hi");
```
CE: unreachable statement.

```
for(i=0; false; i++){
    sop("Hello");
}
sop("Hi");
```
CE: unreachable statement.

```
for(i=0; ; i++)
{
sop("Hello");
}
sop("Hi");
```
CE: unreachable statement.

```
final a=10, b=20;
for(i=0; a<b; i++){
    sop("Hello");
}
sop("Hi");
```
o/p: Hello
:

```
final a=10; b=20;
for(i=0; a>b; i++)
{
sop("Hello");
}
sop("Hi");
```
o/p: Hi

```
final int a=10, b=20;
for(i=0; a<b; i++){
    sop("Hello");
}
sop("Hi");
```
CE: unreachable stmt.

```
final int a=10, b=20;
for(i=0; a>b; i++){
    sop("Hello");
}
sop("Hi");
```
CE: unreachable stmt.

for each loop: (enhanced for loop)

- 1.5 version
- specially designed loop to retrive elements of arrays & collections

eg: print ele of 1D array

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

Normal for loop

```
for(i=0; i<x.length; i++)
{
   sop(x[i]);
}
```

enhanced for loop

```
for(int x1 : x)
{
  sop(x1);
}
```

eg 2: To print ele of 2D array.

```
int[][] x = {{10,20,30},{40,50}};
```

for loop

```
for(i=0; i<x.length; i++)
{
for(j=0; j<x[j].length; j++)
{
  sop(x[i][j]);
}
}
```

```
for(int[] x1 : x){
  for(int x2 : x1){
     sop(x2);
  }
}
```

eg3: 3D array

```
for(int[][] x1 : x){
  for(int[] x2 : x1){
    for(int x3 : x2)
    {
      sop(x3);
    }
  }
}
```

✓

for each loop is the best choice to retrieve elements of arrays & collection but it's limitation is it's applicable only for array & collection & it is not a general purpose loop.

```
for (int i=0; i<10; i++)
{ sop ("Hello");
}
```

→ we can write equivalent for each loop.

```
{
int[] x = {10, 20, 30, 40, 50};
for(i=x.length-1; i>=0; i--){
  sop (x[i]);
}
```

o/p: 50 40 30 20 10

→ we can't write an equivalent for-each loop directly.

---

by using normal for loop we can print elements in either original or reverse order.

by using for each loop we can print ele only in original order.

---

related to foreach loop.

Iterable (I).

```
for(eachitem x : target)
{
  ≡
}.
```

target → Array / collection.

x → Iterable object.

java.lang.Iterable (I)

→ I.S.V

→ one method

iterator().

The target ele in foreach loop should be iterable object

→ An object is said to iterable if & only if corresponding class implements java.lang.Iterable(I) interface.

→ java.lang.Iterable (I) contains only one method iterator().

return type

method

⊛ public Iterator iterator()

→ So all array related classes and collection implemented classes already implemented iterable interface.

→ Being a programmer we are not req to do anything we should aware the point.

Diff b/w Iterator and Iterable

Iterator (I)	Iterable (I)
1. It is related to Collections.	1. It is related to for each loop.
2. we can use to retrieve the elements of collection one by one.	2. The target ele in for-each loop should be Iterable.
3. java.util pkg.	3. java.lang pkg
4. 3 methods → hasNext() → next() → remove().	4. 1 method → iterator()

Transfer statements

1. Inside switch:

```
int x = 0;
switch(x){
case 0:
    sop(0);
case 1:
    sop(1); break;
case 2:
    sop(2);
default:
    sop("def");
}
```

o/p: 0 1

2. Inside loops

```
for (i=0; i<10; i++){
    if (i>=5){
        break;
    }
    sop(i);
}
```

o/p: 0 1 2 3 4

3: Inside labeled blocks

```
class Test{
  p s v m(String[] args){
    int x = 10;
    l1 : {
      sop ("begin");
      if (x >= 10)
        break l1;
      sop ("end");
    }
    sop ("Hello");
  }
}
```

o/p : begin
Hello.

→ These are only places where we can use break statement

if we are using anywhere else we get CE:

```
class Test{
  p s v m (S[] arg){
    int x = 10;
    if (x >= 10)
      break;
    sop ("Hello");
  }
}
```

CE : break outside switch or loop

Continue :

```
for(i=0; i<10; i++)
{
    if (i%2==0)
        continue;
    sop(i);
}
```

o/p
1
3
5
7
9

→ We can continue statement inside loops to skip current iteration & continues for the next iteration

```
class Test{
    p s v m (sty[] ar)
        int x=10;
        if (x>=10)
            continue;
            sop("Hello");
    }
}
```

→ we can use continue statement only inside loop. if we are using anywhere else we will get compile time error saying Continue outside of loop

CE: Continue outside of loop.

Labeled break & continue :-

We can use labeled break & continue to break or continue a particular loop in nested loops.

```
l1:
for(  ) {
    l2:
    for(  ) {
        for(  ) {
            break l1;
            break l2;
            break;
        }
    }
}
```

```
for(int i=0; i<3; i++){
    for(int j=0; j<3; j++){
        if(i==j)
            break;
        sop(i+" "+j);
    }
}
```

break:

```
1 0
2 0
2 1
```

break it:

no o/p

continue:

```
0 1
0 2
1 0
1 2
2 0
2 1
```

continue it:

```
1 0
2 0
2 1
```

do while vs continue

[dangerous combination].

```
int x = 0;
do{
    x++;
    sop(x);
    if(++x<5)
        continue;
    x++;
    sop(x);
}while(++x<10);
```

x = 0
x = 1
2
3
4
4
5
6
7
8
9
10

o/p : 1
4
6
8
10

Declarations & Access Modifiers:

1. Java Source file structure
2. class level modifiers
3. Member level modifiers
4. Interfaces

1. Java Source file structure:

```
class A
{
}
class B
{
}
class C
{
}
```

Can be saved as
A.Java ✓
B.Java ✓
C.Java ✓
Sir.Java ✗

- A Java program can contain any no. of classes but almost one class can be declared as public.
- If there is a public class then name of the program and name of the public class must be matched otherwise we'll get compile time error

Case I: if there is no public class. we can use any name there is no restriction.
A.Java, B.Java, C.Java, Sir.Java.

Case II: if suppose class B is public then name of the program should be B.Java. Otherwise we will get compile time error saying
Ct: class B is public, should be declared in a file name B.Java.

Case III: if suppose class B & C is public and name of the program is B.Java then we will get compile time error saying Ci.

CE: class C is public, should be declared in a file named as C.Java.

while declaring more than one class, if we give public to more than one class then it is not acceptable while saving program name.

```
class A {
    p s v m (String[] args){
        sop("A class main");
    }
}
class B{
    p s v m ( ){
        sop ("B class main");
    }
}
class C{
    p s v m (String args){
        sop("C class main");
    }
}
class D{
}
```

demo.Java

Javac demo.Java

A.class
B.class
C.class
D.class.

Java A ↵

o/p: A class main

Java B ↵

o/p B class main

Java C ↵

o/p C class main

Java D ↵

RE: NoSuchMethodError: main.

Java demo ↵

RE: NoClassDefFoundError: main

Conclusion:

→ Whenever we are compiling a Java program for every class present in the program a seperate .class file is generated.

→ We can compile a Java program (Java source file) but we can run Java.class.

→ Whenever we are executing a Java class the corresponding class main method will be executed.

→ If the class doesn't contain main method then we will get run time error.

RE: NoSuchMethodError: main.

→ If corresponding .class not available then get RE: NoClassDefFoundError: Demo.

→ It is recommended to declare multiple classes in a single source file, it is highly recommended to declare only one class file per source file & name of the program should be same as class name of this approach is readability & maintainability of code is improved.

```
class Test
{
  P S V m (String[] args)
  {
    ArrayList l = new ArrayList();
  }
}
```

CE: cannot find symbol Symbol: class ArrayList Loc: class Test

java.util.ArrayList

```
l = new java.util.ArrayList();
```

fully Qualified name

import statement

Types of import statements

explicit class import
eg: import java.util.ArrayList;
- recommended to use
- improves readability of code

implicit class import
eg import java.util.*;
- reduces readability of code

case II:
import java.util.ArrayList; ✓
import java.util.ArrayList.*; ✗
import java.util.*; ✓
import java.util; ✗

case III:
class myobject extends java.rmi.UnicastRemoteObject
{ ✓ }

The code compiles fine even though we are not writing import statement because we gave fully qualified name.

- whenever we use fully qualified name, it is not necessary to write import statement. & vice versa

case IV:
import java.util.*;
import java.sql.*;
class Test {
  ps vm(String[] args){
    Date d = new Date();
  }
}

CE: reference to Date is ambiguous.

List < util package
       awt package

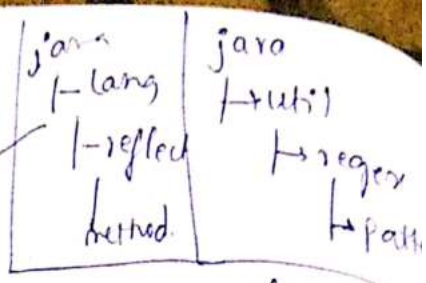
Only date & List are available in both packages & we get ambiguity problem.

case V: while resolving class name compiler will give precedence as
1. explicit class import
2. classes present in CWD (current working directory)
3. Implicit class import

```
import java.util.Date;
import java.sql.*;
class Test {
  ps vm ( ){
    Date d = new Date();
    sop(getClass().getName());
  }
}
```

o/p : util.Date

① import java.*;
② import java.util.*;
③ import java.util.regex.*;
④ No import req;

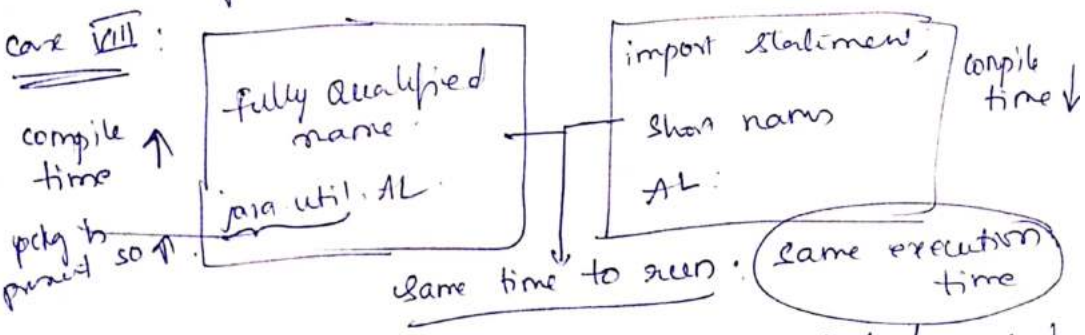


Case VI: When we are importing a java package all class and interfaces present in the package by default available but not subpackages If we want to use subpackage class - compulsory we should write import statement until subpackage level

Case VII: All classes & interface present in following packages are by default available to every java program hence we are not req to write import statement

① java.lang
② default pkg (CWD (Current working Directory))

Case VIII:



import statement is totally compile time related concept.
→ if more no. of i/ps more " " . but no effect on execution time (run time)

Case IX: Diff b/w C lang #include & Java lang import.

In case of C lang #include all i/p, o/p header files header files will be loaded at beginning only at translation time hence it is Static Include.

But in case of Java import statement no .class will be loaded at the beginning. whenever we are using a particular class then only corresponding .class will be loaded this is like dynamic Include or load on demand or load on fly

Static Import

1st new feature

for each

Var-arg

autoboxing & unboxing

Generics

Covariant return type

Queue

Annotation

enum

Static Import not use

reduce length of code & increase readability — SUN Microsystems

usage of static import leads confusion & reduces readability — useful wide expect

if no specific req, the not recomend to use.

without static import

```
class Test {
    p s v m(Str[] args)
    {
    sop(Math.sqrt(4));
    sop(Math.max(10,20));
    sop(Math.random());
    }
}
```

with static import

```
import static java.lang.Math.sqrt;
"      "      "    .   "  .  *;
class Test{
    p s v m( ){
    sop(sqrt(4));
    sop(max(10,20);
    sop(random());
    }
}
```

explain about System.out.println();

```
class Test{
    static String s = "java";
}
```

Test.s.length()

'Test' is class name

s is a static variable present in Test class of the type j.l.String

length() is a method present in String class.

```
class System{
    static PrintStream out;
```

System.out.println()

System is class present in j.l. pckg

out is a static variable present in System class of the type PrintStream

It is a method present in PrintStream class.

```
import static java.lang.System.out;
class Test
{
    p s v m(String[] args) {
        out.println("Hello");
        out.println("Hi");
    }
}
```

o/p = Hello.
o/p = Hi

out is static variable present in System class hence we can access by using class name System. whenever we write static import we can access out directly.

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
public class Test{
    p s v m(String[] args){
        sop(MAX_VALUE);
    }
}
```

CE: Reference to MAX_VALUE is ambiguous
available in Integer & Byte

```
import static java.lang.Integer.MAX_VALUE;   (2)
import static java.lang.Byte.*;              (3)
public class Test{
    static int MAX_VALUE = 999;              (1)
    p s v m(String[] args){
        sop(MAX_VALUE);
    }
}
```

o/p 999.

Priority: ① current class static members
② explicit static import
③ Implicit static import

from above program if ① is ||, then o/p is 2147483647.

from above program if ② & ① are ||
then o/p: 127.

Normal import

① explicit import

syntax:
import packagename.classname;

eg: import java.util.ArrayList;

② implicit import:

syntax:
import package.*;

eg:
import java.util.*;

Static import

① explicit static import

syntax:
import static packagename.classname.staticmember;

eg:
import static java.lang.Math.sqrt;
import static java.lang.System.out;

② Implicit static import:

syntax:
import static packagename.classname.*;

eg:
import static java.lang.math.*;
import static java.lang.System.*;

---

import java.lang.Math.*; ✗
import static java.lang.Math.*; ✓
import java.lang.Math.sqrt; ✗
import static java.lang.Math.sqrt(); ✗
import java.lang.Math.sqrt.*; ✗
import static java.lang.Math.sqrt; ✓
import java.lang; ✗
import static java.lang; ✗
import java.lang.*; ✓
import static java.lang.*; ✗

Two packages contain a class or interface with the same name is very rare and hence ambiguity problem is also very rare in normal import.

But two classes are interfaces certain a variable with same name is very common & hence ambiguity problem is also very " problem in static import.

usage of static import reduces readability & creates confusion & hence if there is no specific requirement it is not recommended to use static import.

Diff b/w normal import & static import.

we can use normal import to import class & interface of a particular package.

→ When N.I it is not req to use fully Qualified name & can use short names directly.

→ we can use static import to import static members of a particular class or interface, whenever we are writing sout it is not req to use class name to access static mems & can access directly

Packages.

→ It is an encapsulation mechanism to group related classes & interfaces into a unit which is nothing but package.

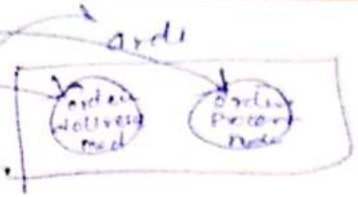
→ All classes & interfaces which are req for • db ops are grouped into a single package which is nothing but java.sql.pckg.

→ for file io operation are grouped into java.io.package.

① To resolve naming conflicts

② To improve modularity of app.

com.xyz.order.process
com.xyz.order.delivery
com.xyz.order



modules

⑥ maintability of app.
⑦ security

Universally accepted naming convention for packages.
→ use internet domain name in reverse

com.icicibank.loan.housing.account

clients internet domain name in reverse | module name | submodule name | classname

package com.durgsoft.scjp;
public class Test{
  p s v m (String[] args)
  {
    sop("pkg demo");
  }
}

① Javac Test.java
cwd:
Test.class

② Javac -d . Test.java
destination to place generated .class file
cwd

cwd:
com
durgasoft
scjp
Test.class

① compiles but does not place Test.class in the mentioned package
② compiles & puts Test.class in the mentioned package,

** If the corresponding package structure not already available then command will create itself corresponding package structure.

→ instead of . we can C:, D: etc as destination
any valid directory name.

Java -d F:\ Test.java

F:
com
durgasoft
scjp
Test.class

If Java -d Z:\ Test.java
If specified directory not available we get CE saying: directory not found: Z:\

→ At the time of execution we have provide fully Qualified name

java com.durgasoft.scjp.Test ← o/p for dir.

Conclusion..

1.
```
package pack1;
package pack2;
public class A{
}
```
CE: class, interface or enum expected

Package Pack2.

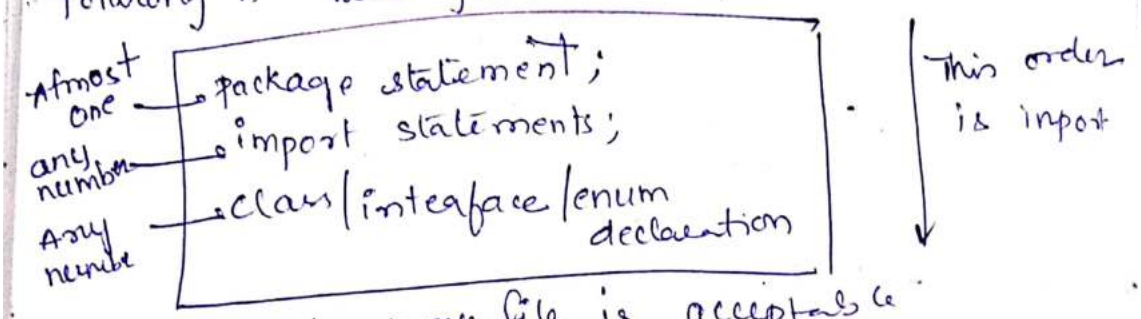
→ In any java source file there can be almost one pck stat, is only allowed

2.
```
import java.util.*;
package pack1;
public class A{
}
```
CE: class, interface or enum expected.

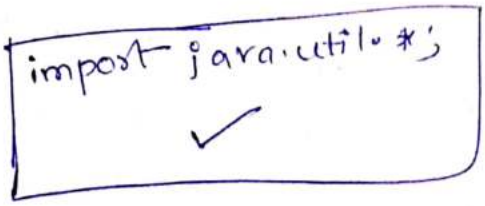
→ The first non comment statement should be package statement if it is available otherwise CE.

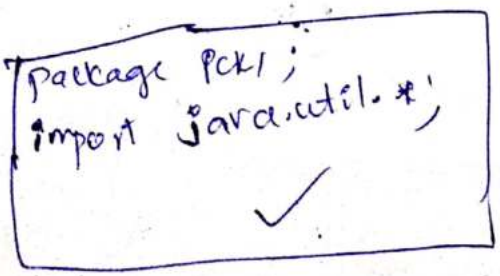
3.

following is valid java source file structure.



empty source file is acceptable.





class level modifiers :-

we have provide some info to JVM of our class.
like child class creation is possible or not
object creation " " " "
class can accessible from anywhere or not.

we can specify info apply declare, approp modifier.

public	final
private	abstract
<default>	static
protected	synchonized

the only applicable modifiers for top level class are

public,	strictfp,	abstract.
<default>,	final,	

but for inner class applicable modifiers.

public,
<default>
final
abstract
strictfp

+

private
protected
static

here class A & class B is inner class because see the braces. main is not in any of the class. so A B ← inner class

to recognize some times number in given Then we can identify!

```
private class Test
{
p s v m ( )
{
sop ("Hello");
}
}
```

✗

E: modifier private not allowed.

```
11 private class A
12 {
13 }
14 static class B
15 {
16 }
17 p s v main(S[] a)
   {
   sop ("Hello");
   }
```

✓

Access specifiers Vs Access Modifiers.

- public, private, protected, default are considered as specifiers.

except these remaining are considered as modifiers.
but this rule is applicable in C++, C but not in Java.

- In java all 12 are considered as modifiers there is no word as specifier.

private class Test {
}

C:\ modifier private not allowed.

Public classes.

-> If class declared as public we can access that class from anywhere.

```
package pack1;
public class A {
    public void m1()
    {
        sop("Hello");
    }
}
```

javac -d . A.java ✓

```
package pack2;
import pack1.A;
class B
{
P S V m(  )
    A a = new A();
    a.m1();
}
```

Javac -d . B.java ✓

java pack2.B ✓

O/p Hello

If class A is not public then while compiling B class we will get compile time error saying

pack1.A is not public in pack1: cannot be accessed from outside package

default class :-

- If class declared as default then we can access that class only within the current package i.e from outside package we can't access
- default access also known package level access.

final modifier :-

- final is a modifier applicable for classes methods and variables.

final method :-

whatever methods parent has by default available to the child through inheritance. If the child not satisfied with parent method implementation then the child is allowed to redefine the method this process is called overriding.

- If the parent class method declared as final we can't override in child class because its implementation is final.

```
class P {
    public void property()
    {
        sop("cash+land+gold");
    }
    public final void marry()
    {
        sop("Subbalakshmi");
    }
}
class C extends P {
    public void marry()
    {
        sop("Trisha");
    }
}
```

CE : marry() in C cannot override marry() in P; overriden method in final.

final class:

If a class declared as final we can't extend functio of that class i.e., we can't create child class fo that class i.e inheritance is not possible fo

final classes.

```
final class P
{

}
class C extends P
{

}
```

cE: cannot inherit from final P.

NOTE: Every method present in final class is always final by default but every variable present in final cla need not be final.

```
final class P {
    m1();
    m2();
    :
    mn();
}
```

→ we cannot create a child class and the methods inside P are also final bcoz as we can't create child that means the methods are also final.

```
final class P
    static int x = 10;
    p s v m() {
        x = 777;
        sq(x);
    }
}
```

o/p 777.

→ variables are not final.

Becoz of final dis adv.	adv
→ Inheritance ✗	→ security
→ polymorphism ✗	→ unique implementat

* With final key adv is security and we can provide unique implementation

* dis adv: we are missing key benefits of OOPS: inheritance (becaus of final classes) & polymorphism (becaus of final methods).

Hence no specific req, then it is not recommend to use final keyword

abstract modifier.

→ Abstract is a modifier applicable for classes & methods but not for variables.

abstract method. → The method that has only declaration & we dont know its implementation.

```
class vehicle
{ abstract
public int getNOofwheels();
}
class bus extends vehicle {
public int getNOofwheels()
{ return 6;
}
}
```

```
class auto extends vehicle
{
public int getNOofwheels()
{
return 3;
}
}
```

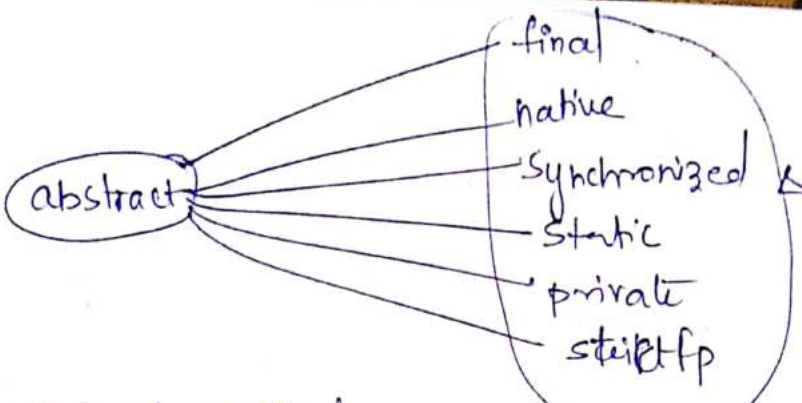
→ Even though we dont know the implementation still we can declare a method with abstract modifier i.e for abstract methods only declaration is available but not implementation hence abstract method dec. should end with ;.

public abstract void m1(); ✓
public abstract void m1(){ }; ✗

→ child class is responsible to provide implementation for parent class abstract method.

→ Order of modifiers not imp in java
public abstract or abstract public ✓

** → By declaring abstract method in parent class we can provide guidelines to child class such that the method must be compulsorily implemented by child.



- abstract method never talks about implementation if any modifier talks about implementation then it forms illegal combination.

abstract final void m1();

CE: illegal combination of modifiers: abstract and final.

abstract class:

→ The class which is having partial implementation

- for any java class we are not allowed to create object because of partial implementation such type of class we have to declare with abstract keyword.

→ i.e for " " instantiation is not possible.

```
abstract class Test {
    public static void main(String[] args) {
        Test t = new Test();   X
    }
}
```

CE: Test is abstract; Test can't be instantiated.

abstract class Vs abstract method.

1. If a class contains atleast one abstract method then compulsory we should declare class as abstract otherwise CE:

Reason: If a class contains atleast one

abstract method implementation is not completed hence it is not recommended to create object to restrict object instantiation we should declare class as abstract.

2. Even though class doesn't contain abstract method still we can declare class as abstract if we dont want instantiation i.e " class can contain zero no. of abstract methods also.

eg1: HTTPServlet is abstract but it doesn't contain any abstract methods.

3. Every adapter class is recommended to declare as abstract but it doesn't contain any abstract method.

class P{
  public void m1();
}

CE: missing method body for declare method as abstract.

X

class P{
  public abstract void m1(){ }
}

CE: abstract methods do not have body.

X

class P
{
public abstract void m1();
}

CE: P is not abstract and does not override abstract method m1() in P

X

abstract class P{
  public abstract void();
}

Abstract.

→ If we are extending abstract class then for each & every abstract method of parent class we should provide implementation otherwise declare child class as abstract.

```
abstract class P {
    public abstract void m1();
    public abstract void m2();
}
class C extends P {
    public void m1() { }
}
```

X

C is not abstract & does not override abstract method m2() in P.

In this case next level child class is responsible to provide child level implementation.

final Vs Abstract

* Abstract methods compulsory we should override in child classes to provide implementation whereas we can't override final methods hence final abstract combination is illegal comb. for methods.

* for final classes we can't create child classes where as for abstract class we should create child class to provide implementation. Hence final abstract comb. is illegal for classes.

**
```
abstract class Test
{
    public final void m1();
}
```

final class Test {

    public abstract void m1();

}

abstract
(final
↳ combo
illegal.

X

*** abstract class can contain final method whereas final class can't contain abstract method.

If method abstract then class also must be abstract (so)

- usage of abstract modifier is highly recommended because it promotes several OOP features like inheritance & polymorphism.