1. Language Fundamentals

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.

1) Identifiers :—

→ A name in Java program is called Identifier.

→ It can be a class name or method name or variable name or label name.

Ex:
```
class Test                                   ①
{
    public static void main(String[] args)
                       ②      ③       ④
    {
        int a=10;
            ⑤
    }
}
```

Rules for Identifiers :—

1. The only allowed characters in Java identifiers are a-z, A to Z, 0 to 9, _(underscore) and $ (dollar).

   If we are toying to use any other character then we will get compile time error.

Ex: total_number ✓

total# ✗

2. Identifiers should not start with digits.

Ex: total123 ✓

123total ✗

3. Java identifiers are case sensitive. Ofcourse Java language itself is considered as Case sensitive programming language.

Ex:
```
class Test
{
    int number =10;
    int NUMBER =20;
    int Number =30;
}
```
} we can differentiate w.r.t. case

4. There is no length limit for Java identifiers, but it is never recommended to take lengthy identifiers becoz it reduces readability of the code.

DEMO

5. We can't use reserved words as identifiers.

Ex: int a=10; ✓

✗ int if=10; ⟶ CE

⟶ Reserved word.

6. All predefined Java class names & interface names we can use as identifiers.

Ex:
```
class Test
{
  P S V m(-)
  {
    int String=10;          | int Runnable=20;
    S.o.p(String);          | S.o.p(Runnable);
  }
}
```
Olp=10 ✓ Olp: 20 ✓

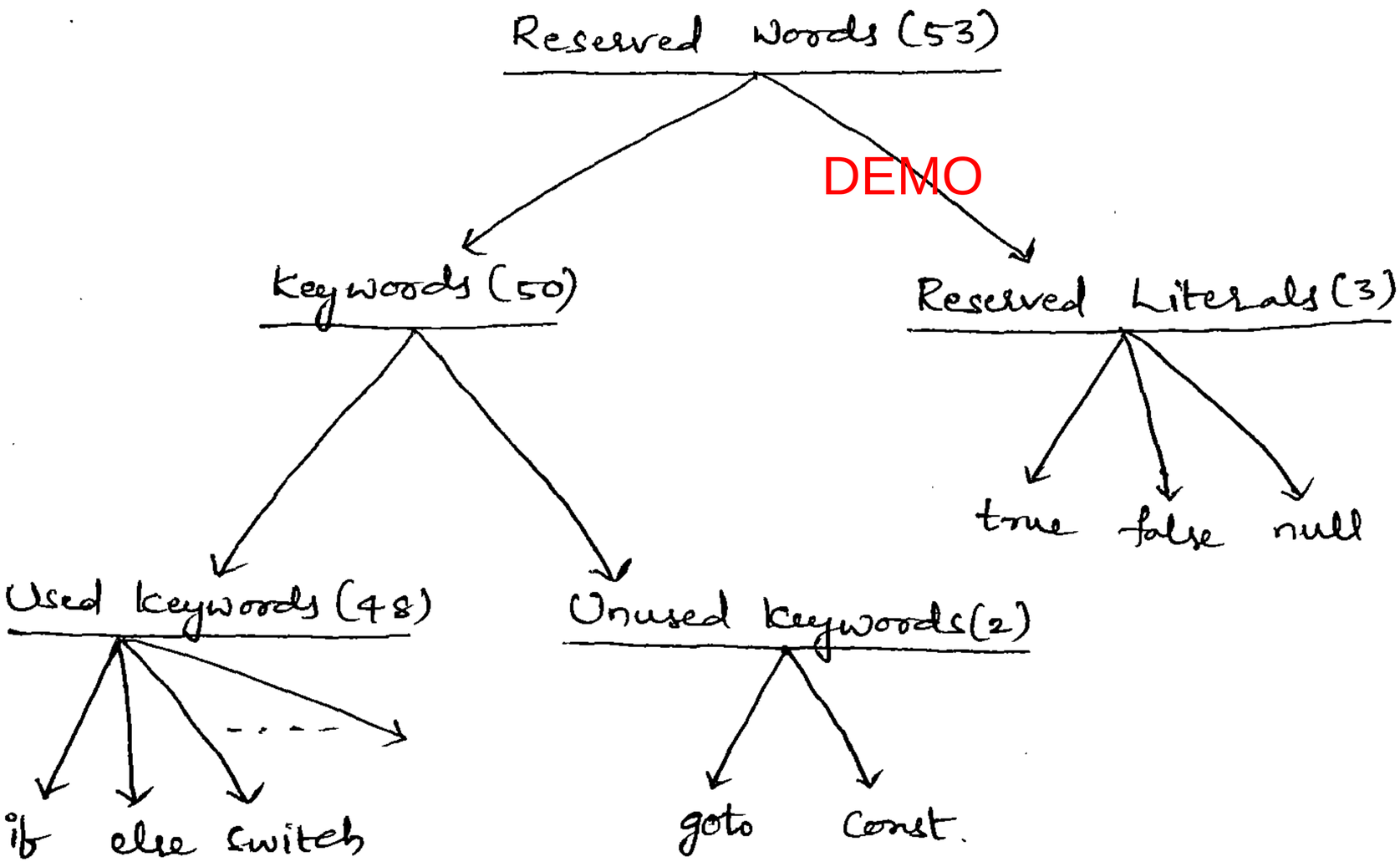
→ Eventhough it is legal to use predefined Java class names & interface names as identifiers, it is not a good programming practice.

Q: Which of the following are valid Java identifiers?

① ca$h ✓
② total_number ✓
③ all@hands ✗
④ Java2share ✓
⑤ _$_$_$ ✓
⑥ int ✗
⑦ Int ✓
⑧ Integer ✓

2) Reserved words :—

→ In Java some words are reserved to represent some meaning or functionality such type of words are called Reserved words



Used keywords (48)

Keywords for datatypes	Keywords for flow control	Keywords for modifiers	Keywords for Exception Handling	class related keywords	object related keywords	void return type keyword
byte	if	public	try	class	new	void
short	else	private	catch	interface	super	(1)
int	switch	protected	finally	package	this	
long	case		throw	import	instanceof	
float	default	final	throws	extends	(4)	
double	while	static	assert (1.4v)	implements		
char	do	abstract	(3)	(6)		
boolean	for	native				
(8)	break	synchronized				
	continue	volatile				
	return	transient				
	(11)	strictfp (1.2v)				
		(11)				

DEMO

→ If a method won't return anything then we should declare that method with void return type.

→ In Java return type is mandatory, but in c language return type is optional and default return type is int.

Unused keywords :—

1. goto :—

→ Usage of goto created several problems in old languages.

→ Hence SUN people banned this keyword in Java.

2. const :—

→ Use final instead of const.

Note:— By mistake if we are trying to use goto and const then we will get compile time error.

Reserved Literals :—

true } values for boolean data type.
false }

null ⇒ default value for object reference.

enum (1.5V) :—

→ If we want to represent a group of named constants then we should go for enum.

Ex:

```
enum Month
{
   JAN, FEB, ...., DEC;
}
```

```
enum Beer
{
  KF, KO, RC, FO;
}
```

Conclusions :—

1. All reserved words in Java contains only lower case alphabet symbols.

DEMO

2. The new keywords in Java are

   strictfp ⟶ 1.2v
   assert ⟶ 1.4v
   enum ⟶ (1.5v)

3. In Java, we have only new keywood, but not delete keywood becoz destruction of useless objects is the responsibility of Garbage Collector.

4. strictfp but not strictFp
   instanceof but not instanceOf
   synchronized but not synchronize
   extends but not extend
   implements but not implement
   import but not imports

   const but not constant.

Q: Which of the following list contains 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
✗ ⑦ implements, extends, imports
✗ ⑧ sizeof, instanceof
✗ ⑨ instanceOf, strictFp
✗ ⑩ byte, short, Int
✓ ⑪ None of the above

Q: Which of the following are valid Java reserved words?

✓ ① public
✓ ② static
✓ ③ void
✗ ④ main
✗ ⑤ String
✗ ⑥ args

3) Data Types :—

→ In Java, every variable has a type, & every expression has a type & every type is strongly checked.

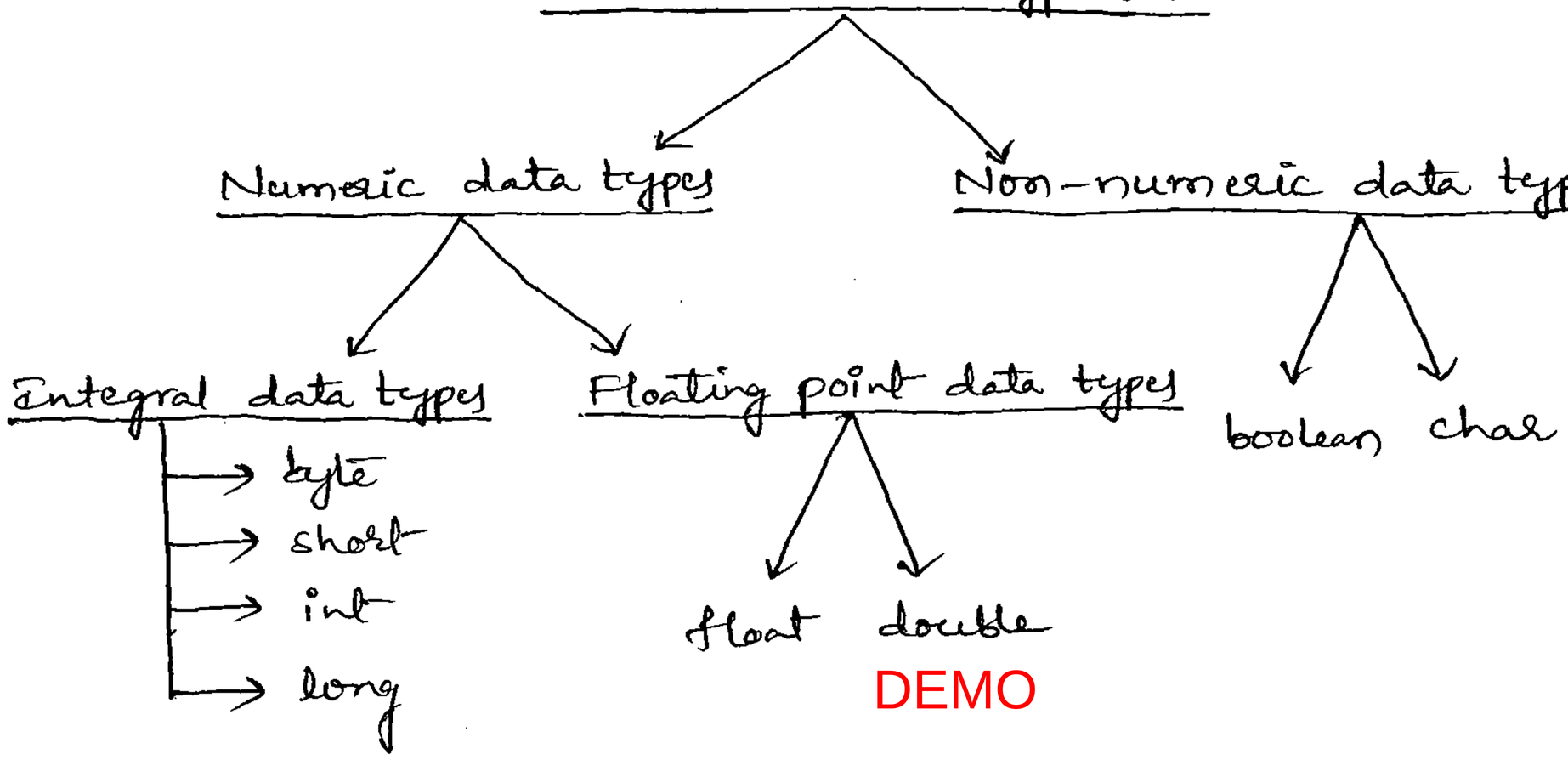
→ Each & every assignment should be checked by the compiler for type compatibility.

→ Hence Java language is considered as strongly typed programming language.

*** → Java is not considered as Pure object oriented programming language becoz several OOP features (like Multiple Inheritance, Operator overloading etc) are not supported by Java.

*** → Moreover we are depending on primitive data types which are non-objects.

Primitive Data Types (8)



DEMO

→ Except boolean and char the remaining data types are considered as Signed data types becoz we can represent both in +ve & -ve numbers.

Integral data types :-

1) byte :-

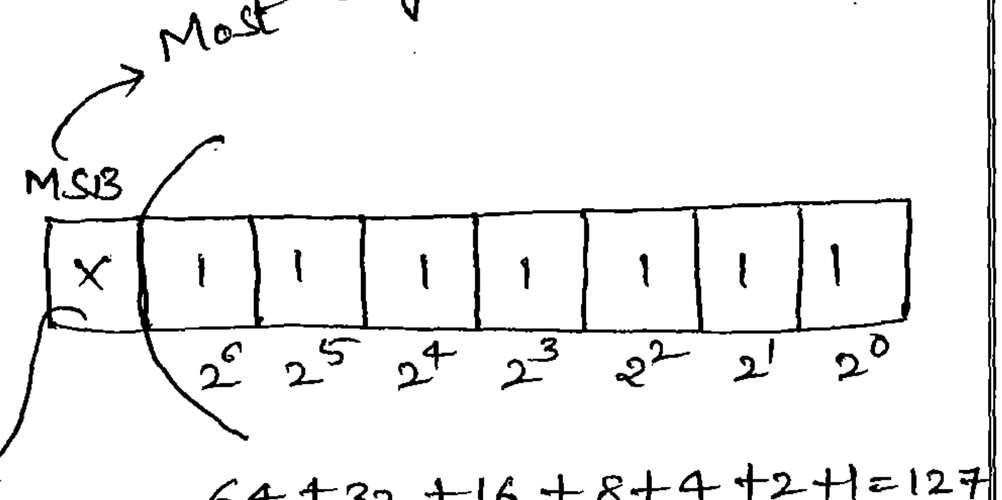
Size : 1 byte (8 bits)

MAX_VALUE : +127

MIN_VALUE : -128

Range : -128 to +127

[-2^7 to 2^7-1]



→ The MSB acts as Sign bit.

→ 0 means +ve number & 1 means -ve number.

→ +ve numbers will be represented directly in the memory where as -ve numbers will be represented in 2's complement form.

Ex: ✓ byte b = 10;

✓ byte b = 127;

✗ byte b = 128;

CE: possible loss of precision
found: int
required: byte

✗ byte b = 10.5;

CE: PLP
found: double
required: byte

✗ byte b = true;

CE: incompatible types
found: boolean
required: byte

✗ byte b = "durga";

CE: incompatible types
found: j.l.String
required: byte

→ byte data type is best suitable if we want to handle data in terms of Streams either from the file or from the network.

2) short :-

→ The most rarely used data type in Java is short data type.

Size : 2 bytes [16 bits]

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

Ex: ✓ short s = 32767;

✗ short s = 32768;

CE: PLP
found: int
required: short

✗ short s = 10.5; → CE : PLP found : double required : short

✗ short s = true; → CE : incompatible types found : boolean required : short

→ short data type is best suitable for 16-bit processors like 8086, but these processors are completely out-dated and hence the corresponding short data type is also out dated data type.

3) int :-

→ The most commonly used data type in Java is int data type.

Size : 4 bytes [32-bits]

Range : -2^{31} to $2^{31}-1$ [-2147483648 to 2147483647]

Ex: int x = 2147483647;

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

✗ int x = 2147483648l; → CE : PLP found : long required : int

✗ int x = 10.5; → CE : PLP found : double required : int

✗ int x = true; → CE : incompatible types found : boolean required : int

4) long :—

→ Sometimes int may not enough to hold big values then we should go for long data type.

Ex①: To hold amount of distance travelled by light in 1000 days int may not enough then we should go for long data type.

```
long l = 1,26,000 X 60 X 60 X 24 X 1000 miles;
```

Ex②: To hold no of characters present in a big file int may not enough then we should go for long data type.

```
long l = f.length();
```

Size : 8 bytes [64 bits]

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

Note:— All the above 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 types.

Floating point data types:—

float	double
1. If we want 5 to 6 decimal places of accuracy then we should go for float.	1. If we want 14 to 15 decimal places of accuracy then we should go for double.
2. float follows single precision.	2. double follows double precision.
3. Size : 4 bytes Range : -3.4e38 to 3.4e38	3. Size : 8 bytes Range : -1.7e308 to 1.7e308

boolean data type :—

Size : Not Applicable (VM dependent)

Range : Not Applicable (But only allowed values either true or false)

Ex: ✓ boolean b = true;

✗ boolean b = 0; → CE : incompatible types
found : int
required: boolean

✗ boolean b = True; → CE : cannot find symbol
Symbol : variable True
location: class Test

✗ boolean b = "true"; → CE : incompatible types
found: j.l.String
required : boolean

DEMO

```
int x=0;
if(x)
{
  S.o.p("Hello");
}
else
{
  S.o.p("Hi");
}
```

CE : incompatible types
found : int
required : boolean

```
while(1)
{
  S.o.p("Hello");
}
```

char data type :—

→ Old languages like C, C++ are ASCII code based and the no. of ASCII characters are less than or equal to 256, to represent these characters 8bits are enough.

→ Hence the size of char is 1 byte.

→ But Java is UNI code based and the no. of UNI code characters are > 256 & ≤ 65536.

→ To represent these many characters 8 bits may not enough then we should go for 16 bits.

→ Hence the size of char in Java is 2 bytes.

Size : 2 bytes (16 bits)

Range : 0 to 65535.

Summary of Java Primitive Data Types :—

data type	Size	Range	Corresponding Wrapper class	Default value
byte	1 byte	-128 to 127 $(-2^7$ to $2^7-1)$	Byte	0
short	2 bytes	-32768 to 32767 $(-2^{15}$ to $2^{15}-1)$	Short	0
int	4 bytes	-2147483648 to 2147483647 $[-2^{31}$ to $2^{31}-1]$	Integer	0
long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
float	4 bytes	$-3.4e38$ to $3.4e38$	Float	0.0
double	8 bytes	$-1.7e308$ to $1.7e308$	Double	0.0
boolean	NA	NA (allowed values true/false)	Boolean	false
char	2 bytes	0 to 65535	Character	space character

DEMO

Note :- The default value for object reference is null.

4) Literals :—

→ Any constant value which can be assigned to a variable is called Literal.

Ex: int x = 10;

Data type/ Keyword — Name of variable/ Identifier — constant value/ literal.

Integral Literals :—

→ For Integral data types (byte, short, int, long) we can specify literal value in the following ways.

1) Decimal Literals (base-10) : —

→ Allowed digits are 0 to 9.

Ex: int x = 10;

2) Octal Literals (base-8) : —

→ Literal value should be prefixed with '0' and allowed digits are 0 to 7.

Ex: int x = 010;

3) Hexadecimal Literals (base-16) : —

→ Allowed digits are 0 to 9, a to f.

→ For extra digits we can use both lower case and upper case (a to f or A to F).

DEMO

→ This is one of very few areas where Java is not case sensitive.

→ Literal value should be prefixed with 0x or 0X.

Ex: int x = 0X10;

→ These are the only possible ways to specify literal value.

Q: Which of the following are valid declarations?

① int x = 10;

② int x = 0786; → CE: integer number too large

③ int x = 0777;

④ int x = 0XFace;

⑤ int x = 0XBeef;

⑥ int x = 0XBeer; → CE

Ex:
```
class Test
{
  P s v m(-)
  {
    int x=10;
    int y=010;
    int z=0X10;
    S.o.p(x+" "+y+" "+z);
  }
}
```
o/p: 10 8 16

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

$= 0\times8^0 + 1\times8^1 = 0+8 = 8$

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

$= 0\times16^0 + 1\times16^1 = 0+16 = 16$

→ By default every integral literal is of int type, but we can specify explicitly as long type by suffixed with l or L.

Ex: ✓ int x=10;

✓ long l=10L;

✓ long l=10;

int x=10L; ⟶ CE: PLP
found: long
required: int

→ There is no direct way to specify byte and short literal explicitly.

→ Whenever we are assigning integral literal to the byte variable and if the value is within the range of byte then compiler automatically treats it as byte value.

→ My short literals also.

Ex: ✓ byte b=10;

✓ byte b=127;

byte b=128; ⟶ CE: PLP
found: int
required: byte

✓ short s = 32767;

short s = 32768; ⟶ CE: PLP
found: int
required: short

Floating point literals:—

→ By default every floating point literal is of double type, but we can specify explicitly as float type by suffixed with 'f' or 'F'.

Ex: float f = 123.456; ⟶ CE: PLP
found: double
required: float

✓ double d = 123.456;

✓ float f = 123.456f;

→ We can specify explicitly floating point literal as double type by suffixed with 'd' or 'D' ofcourse this convention is not required.

DEMO

Ex: ✓ double d = 123.456D;

float f = 123.456D; ⟶ CE: PLP
found: double
required: float

→ We can specify floating point literals only in decimal form and we can't specify in octal & heradecimal forms.

Ex: ✓ double d = 123.456;

✓ double d = 0123.456; ⟶ 8 is treated as decimal only but not octal

double d = 0X123.456; ⟶ CE:

→ we can assign integral literal directly to the floating point data types and that integral literal can be specified either in octal or hexadecimal or decimal form.

Ex: ✓ double d = 123.456;

double d = 0786; → CE: integer number too large

✓ double d = 0X face;

✓ double d = 0X Beef;

✓ double d = 0777;

✓ double d = 10;

→ But we can't assign floating point literals to the integral types.

Ex: int x = 10.5; → CE: PLP
found: double
required: int

→ we can specify floating point literals even in exponential form also (scientific notation).

Ex: ✓ double d = 1.2e3;

S.o.p(d); ⇒ O/P : 1200.0

$1.2e3 = 1.2 \times 10^3$
$= 1.2 \times 1000$
$= 1200.0$

float f = 1.2e3; → CE: PLP
found: double.
required: float

✓ float f = 1.2e3f;

boolean Literal :—

→ The only allowed values for boolean data type are true or false.

Ex: ✓ boolean b = true;

✗ boolean b = 0; → CE : incompatible types
found : int
required : boolean

✗ boolean b = True; → CE : cannot find symbol
symbol : variable True
location : class Test

✗ boolean b = "true"; → CE : incompatible types
found : j.l. String
required : boolean

DEMO

Ex:
```
int a=0;
if(a)
{
  S.o.p("Hello");
}
else
{
  S.o.p("Hi");
}
```
→ CE : incompatible types
found : int
required : boolean

```
while(a)
{
  S.o.p("Hello");
}
```

char Literal :—

→ We can represent a char literal as a single character within single quotes.

Ex: char ch = 'a'; ✓

✗ char ch = a; → CE : cannot find symbol
symbol : variable a
location : class Test

✗ char ch = "a"; → CE: incompatible types
found: j.l.String
required: char

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

→ We can represent a char literal as integral literal which represents UNI code value of that character.

→ The integral literal can be specified either in decimal or octal or hexadecimal.

→ The allowed range is 0 to 65535.

Ex: ✓ char ch = 97;
S.o.p(ch); ⟹ O/P: a

DEMO

char ch = 0777; ✓
char ch = 0XFace; ✓
char ch = 0XBeef; ✓
char ch = 65535; ✓
✗ Char ch = 65536; → CE: PLP
found: int
required: char

→ We can represent a char literal in UNI code representation which is nothing but '\uXXXX'
→ 4 digit hexa decimal number.

Ex: ✓ char ch = '\u0061';
S.o.p(ch); ⟹ o/p: a
✗ char ch = \u0062;
✗ char ch = '\iface';
✓ char ch = '\ubeef';

→ Every escape character in Java is valid char literal.

Ex: ✓ char ch = '\n';

✓ char ch = '\t';

✗ char ch = '\m'; → CE: Illegal escape character.

Escape character	Description
\n	New line
\t	Horizontal tab
\r	Carriage return
\b	Back space
\f	Form feed
\'	single quote
\"	double quotes
\\	back slash

String Literal :—

→ A sequence of characters within double quotes is called String Literal.

Ex: String s = "Java";

***

1.7 version enhancements with respect to Literals :—

1) Binary Literals :—

→ Until 1.6 version we can specify literal value for the integral data types in the following 3 ways

1. decimal Literals
2. Octal Literals
3. hexadecimal Literals

→ But from 1.7 version onwards we can specify literal value even in binary form also.

→ The literal value should be prefixed with 0b or 0B.

→ Allowed digits are 0 and 1.

Ex: int x = 0B1111; ✓

S.o.p(x); ⟹ o/p : 15

2) Usage of (_) underscore symbol in Numeric Literals :—

→ From 1.7 version onwards we can use (_) symbol in numeric literals.

Ex: double d = 123456.789;

⇓

double d = 1_23_456.7_8_9;

double d = 123_456.7_8_9;

→ The main advantage of this approach is readability of the code will be improved.

→ At the time of compilation these underscore symbols will be removed automatically.

→ Hence after compilation the above lines will become

double d = 123456.789;

→ We can write any number of underscore symbols b/w the digits.

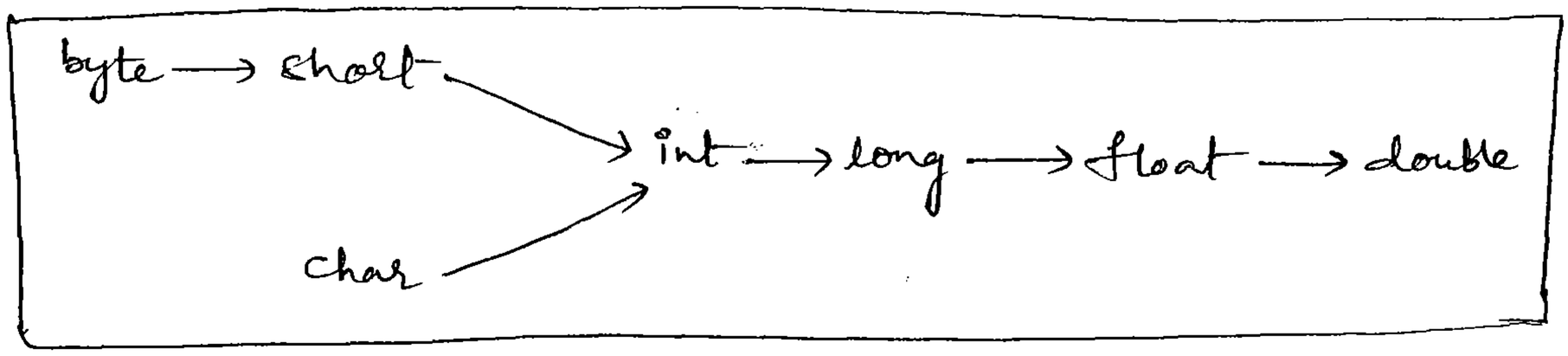
Ex: double d = 1_2__3___456.7_8__9; ✓

→ We can use underscore symbol only b/w the digits.

Ex: double d = _1_23_456.7_8_6; ✗

double d = 1_23_456_.7_8_6; ✗

double d = 1_23_456.7_8_9_; ✗



→ Eventhough long is 8 bytes we can assign its value to 4 bytes float variable becoz internally they follow different memory representations.

Ex: float f = 10l; ✓

S.o.p(f) ⇒ o/p : 10.0

5) Arrays :—

1. Introduction
2. Array Declaration
3. Array Creation
4. Array Initialization
5. Array Declaration, Creation & Initialization in a single line
6. length Vs length()
7. Anonymous Arrays
8. Array Element Assignments
9. Array variable Assignments.

DEMO

1) Introduction :—

→ An Array is an indexed collection of fixed no. of homogeneous data elements.

→ The main advantage of Arrays is we can represent multiple values by using a single variable. So that readability of the code will be improved.

→ But the main disadvantage of Arrays is fixed in size i.e., once we created an Array with some size there is no chance of increasing or decreasing the size based on our requirement.

→ Hence to use Arrays concept compulsory we should know the size in advance which may not possible always.

→ We can overcome this problem by using Collections.

2) Array Declaration :—

1-Dimensional Array Declaration :—

```
int[]  x;  → recommended to use becoz name is clearly separated from type.
int  []x;
int  x[];
```

*** → At the time of Array declaration we can't specify the size otherwise we will get compile time error.

Ex: `int[6] x;` → CE DEMO

`int[] x;` ✓

2-Dimensional Array Declaration :—

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

3-Dimensional Array Declaration :—

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

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

Q: Which of the following Array declarations are valid?

✓ 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

→ If we want to specify the dimension before the variable this facility is applicable only for the first variable in declaration. If we are toying to apply for the next variables we will get CE.

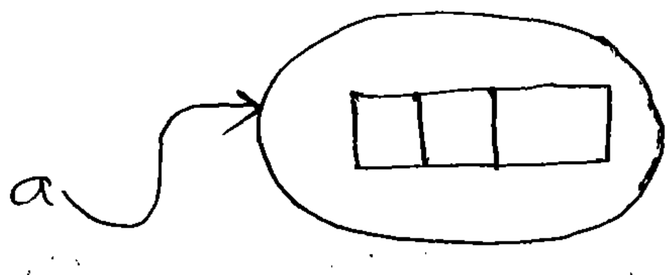
Ex: int[] []a, []b, []c;
✓ ✗ ✗

DEMO

3) Array Creation :—

→ Every Array in Java is an object. Hence we can create an Array by using new operator.

Ex: int[] a = new int[3];



→ For every Array type the corresponding classes are available & these classes are part of Java language & not applicable to the programmar.

Array type	Corresponding class name
int[]	[I
int[][]	[[I
byte[]	[B
short[]	[S
long[]	[J
float[]	[F
double[]	[D
boolean[]	[Z
char[]	[C
char[][]	[[C

DEMO

***

1. At the time of Array creation compulsory we should specify the size, o.w. we will get CE.

Ex: int[] x=new int[]; X

int[] x=new int[3]; ✓

2. It is legal to have an Array with zero size in Java.

Ex: int[] x=new int[0]; ✓

***

3. If we are toying to specify Array size with some -ve int value we will get Runtime Exception saying, NegativeArraySizeException.

Ex: int[] x=new int[-6]; → RE: NegativeArraySizeException

4. To specify Array size the allowed data types are

byte
short
char
int

By mistake if we are trying to provide any other type then we will get CE.

Ex: ✓ int[] a=new int[10]; → int

✓ int[] a=new int['a'];

byte b=10;

✓ int[] a=new int[b];

Short s=20;

✓ int[] a=new int[s];

✗ int[] a=new int[10l]; → CE: PLP found: long required: int

DEMO

Note:- The max. allowed Array size in Java is 2147483647, which is max. value of int data type.

Ex: ✓ int[] a=new int[2147483647];

✗ int[] a=new int[2147483648]; → CE: integer number too large

→ In the first case, we may get OutOfMemoryError, if sufficient heap memory is not available. This is the problem with machine but not with Java.

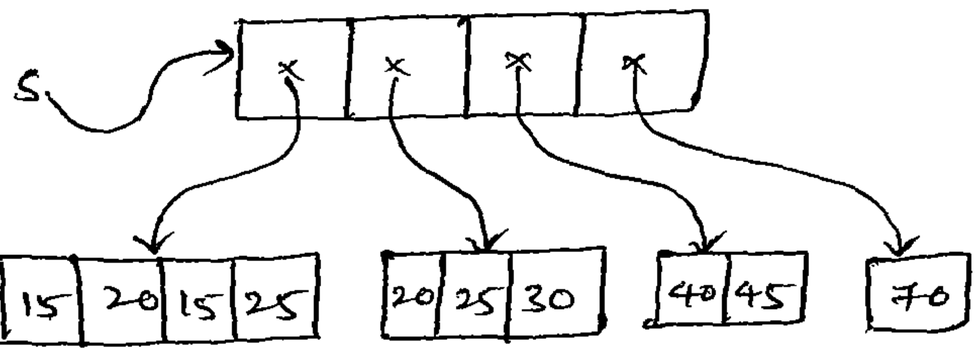
2- Dimensional Array Creation :-

→ In Java, multi-dimensional arrays are not implemented in matrix form and these are implemented by using Array of Arrays approach.

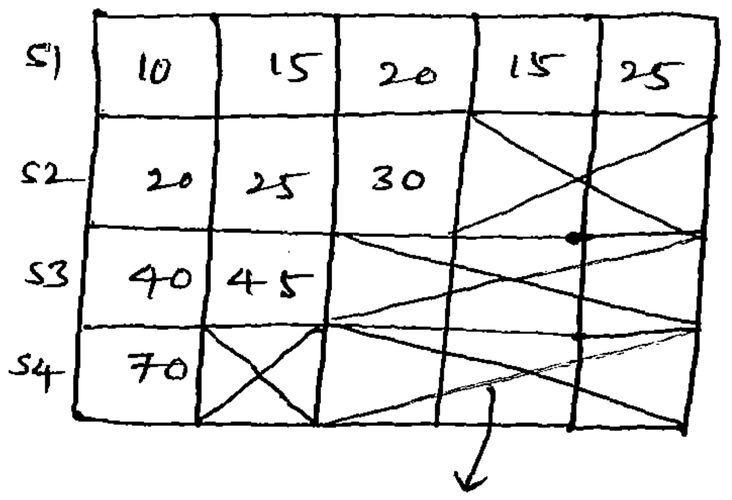
→ The main advantage of this approach is memory utilization will be improved.

Ex:

Array of Arrays



Matrix style

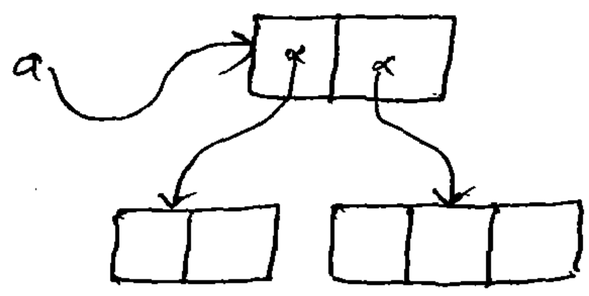


Memory wastage.

Ex: ① `int[][] a=new int[2][];`

`a[0]=new int[2];`

`a[1]=new int[3];`



Ex ②: `int[][][] a=new int[2][][];`

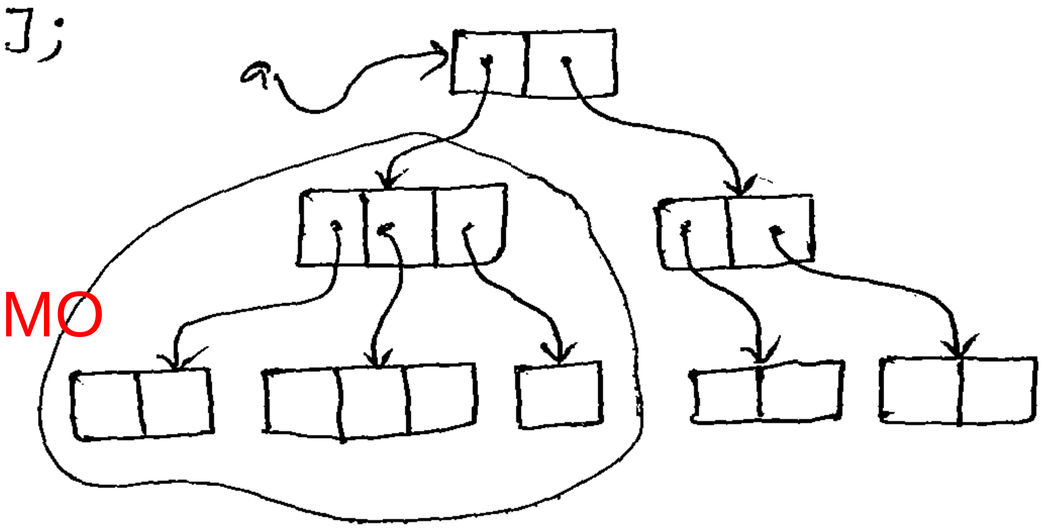
`a[0]=new int[3][];`

`a[0][0]=new int[2];`

`a[0][1]=new int[3];`

`a[0][2]=new int[1];`

`a[1]=new int[2][2]`



DEMO

Q: which of the following are valid?

✗ ① `int[] a=new int[];`

✓ ② `int[] a=new int[3];`

✗ ③ `int[][] a=new int[][];`

✓ ④ `int[][] a=new int[3][];`

✗ ⑤ `int[][] a=new int[][4];`

✓ ⑥ `int[][] a=new int[3][4];`

✓ ⑦ `int[][][] a=new int[3][4][5];`

✓ ⑧ `int[][][] a=new int[3][4][];`

✗ ⑨ `int[][][] a=new int[3][][5];`

✗ ⑩ `int[][][] a=new int[][4][5];`

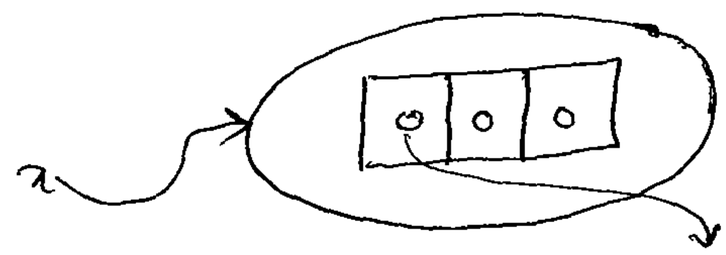
4) Array Initialization :—

→ Once we created an array its elements are by default initialized with default values.

Ex: ① `int[] a=new int[3];`

`S.o.p(a);` ⇒ o/p : [I3e25a5

`S.o.p(a[0]);` ⇒ o/p : 0



`S.o.p(a.toString());` default value

className@hexadecimal_hashcode

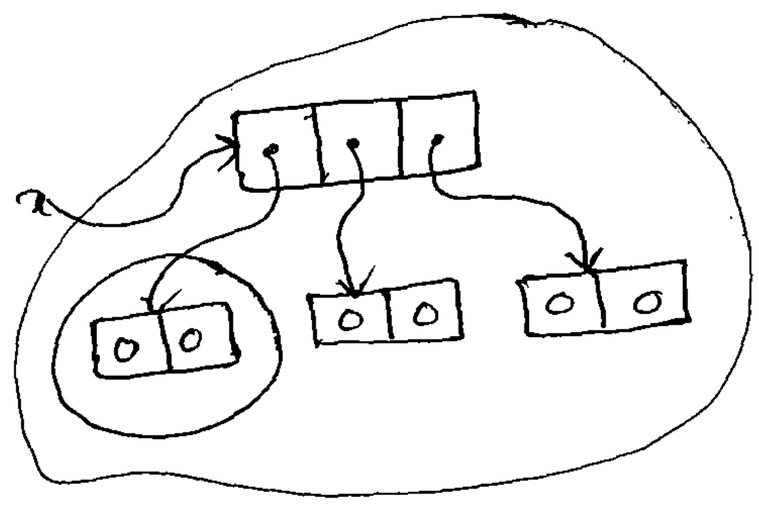
Note:- Whenever we are trying to print any object reference internally toString() method will be called

Ex②: `int[][] a=new int[3][2];`

`S.o.p(a);` ⇒ o/p : [[I3e25a5

`S.o.p(a[0]);` ⇒ o/p : [I19821f

`S.o.p(a[0][0]);` ⇒ o/p : 0



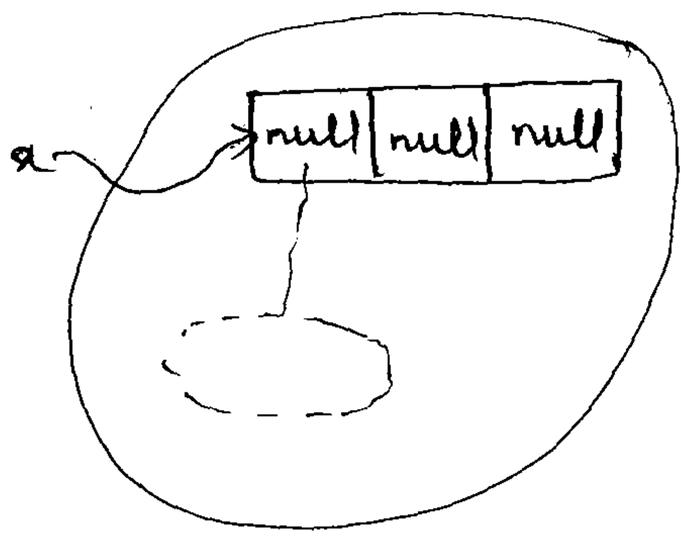
DEMO

Ex③: `int[][] a=new int[3][];`

`S.o.p(a);` ⇒ o/p : [[I3e25a5

`S.o.p(a[0]);` ⇒ o/p : null

`S.o.p(a[0][0]);` → RE : NPE



Note:- If we are trying to perform any operation on null then we will get NullPointerException.

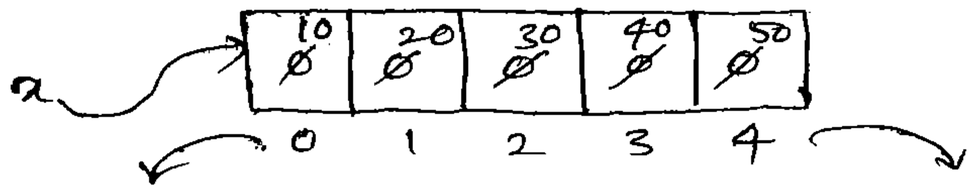
→ Once we created an array every array element is by default initialized with default values.

→ If we are not satisfied with default values then we can override these default values with our customized values.

Ex: `int[] a=new int[5];`

`a[0]=10;`

```
x[1] = 20;
x[2] = 30;
x[3] = 40;
x[4] = 50;
```




```
x[5] = 60;   ⟶ RE : ArrayIndexOutOfBoundsException
x[-5] = 70;  ⟶ RE : AIOOBE
x[1.5] = 80; ⟶ CE : PLP
                found : double
                required : int
```

Note :- If we are trying to access array element with out of range index then we will get RE saying, ArrayIndexOutOfBoundsException.

5) Array Declaration, Creation & Initialization in a single line :-

→ We can declare, create & initialize an array in a single line.

ex:

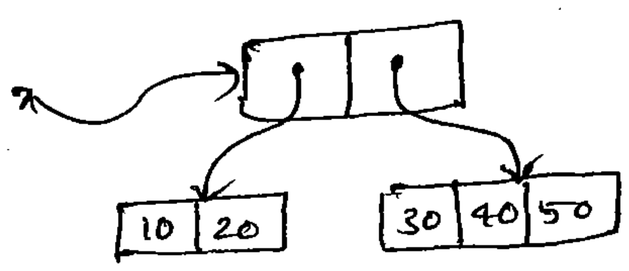
```
int[] x;
x=new int[3];
x[0]=10;
x[1]=20;
x[2]=30;
```

⟹

```
int[] x={10, 20, 30};
String[] s={"nag", "chiru", "venki" "balu"};
char[] ch={'a', 'e', 'i', 'o', 'u'};
```

DEMO

→ Even we can extend this short cut for multi-dimensional arrays also.



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

ex: `int[][][] a={{{10,20,30},{40,50}},{{60,70,80},{90,100,110}}};`

S.o.p(a[2][1][0]); ⟹ RE : AIOOBE

S.o.p(a[1][1][0]); ⟹ o/p : 90

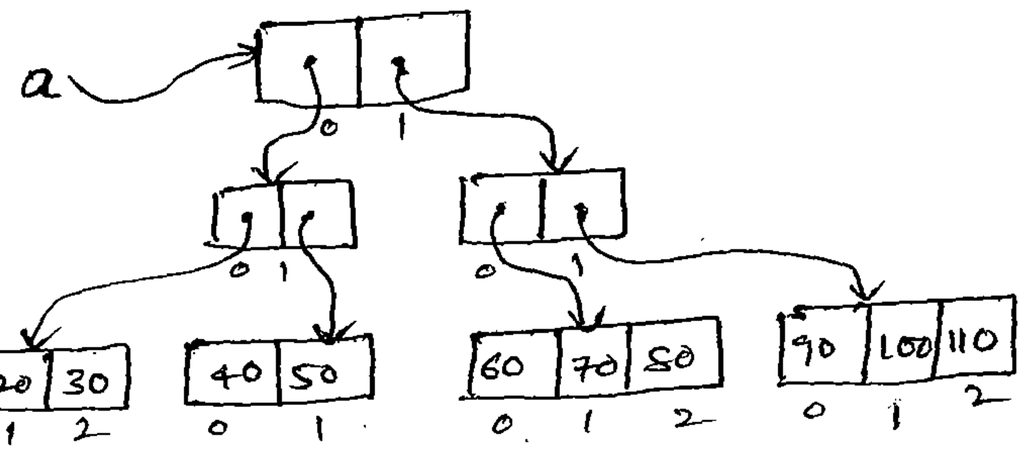
S.o.p(a[0][1][2]); ⟹ RE : AIOOBE

S.o.p(a[1][0][2]); ⇒ o/p : 80

S.o.p(a[1][1][2]); ⇒ o/p : 110

S.o.p(a[0][0][2]); ⇒ o/p : 30

S.o.p(a[2][0][1]); → RE : AIOOBE



→ If we want to use this short cut compulsory we have to perform in a single line only.

→ If we are toying to divide into multiple lines we will get CE.

Ex: int[] x = {10, 20, 30};

int[] x;
x = {10, 20, 30}; → CE : Illegal start of expression

***

6) length Vs length() :—

DEMO

length :—

→ length is a final variable applicable only for arrays.

→ length variable represents size of the array.

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

S.o.p(x.length()); → CE : cannot find symbol
symbol : method length()
location : class int[]

S.o.p(x.length); ⇒ o/p : 4

length() :—

→ length() is a final method applicable for String objects.

→ It returns no. of characters present in String.

Ex: String s = "durga";

S.o.p(s.length); → CE : cannot find symbol
symbol : variable length
location : class j.l.String

S.o.p(s.length()); ⇒ o/p : 5

Note:- length variable applicable for arrays, but not for String objects whereas length() method applicable for String objects, but not for arrays.

Ex: String[] s={"A", "AA", "AAA"};

✓ S.o.p(s.length); ⇒ o/p: 3

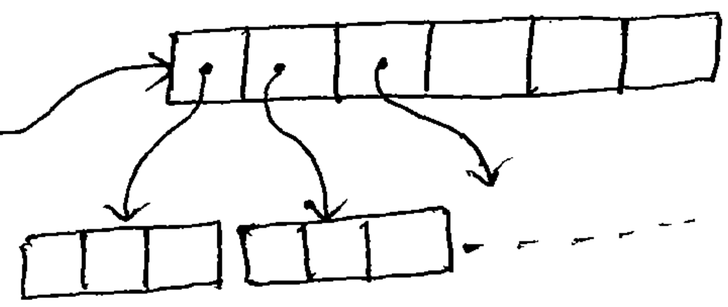
✗ S.o.p(s.length()); → cc: cannot find symbol
symbol: method length()
location: class String[]

✗ S.o.p(s[0].length); → cc: cannot find symbol
symbol: variable length
location: class j.l.String

✓ S.o.p(s[0].length());
⇓
o/p: 1

Note:- In multi-demensional arrays, length variable represents only base size, but not total size.

Ex: int[][] a=new int[6][3];



S.o.p(a.length); ⇒ o/p: 6

S.o.p(a[0].length); ⇒ o/p: 3

→ There is no direct way to find total length of multi-dimensional arrays, but we can find indirectly as follows.

```
a[0].length + a[1].length + a[2].length + ----
```

***

7) Anonymous Arrays:-

→ Sometimes we can declare an array without name such type of nameless arrays are called Anonymous Arrays.

→ The main purpose of anonymous arrays is just for instant use(1 time usage).

→ We can create anonymous arrays as follows.

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

→ We can create multi-dimensional anonymous arrays also.

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

→ While creating anonymous arrays we can't specify the size otherwise we will get CE.

Ex: new int[3]{10,20,30} ✗

new int[]{10,20,30} ✓

→ Based on our requirement we can give the name for anonymous array then it is no longer anonymous.

Ex: int[] a=new int[]{10,20,30}; ✓

Ex:
```
class Test
{
  P s v m(_)
  {
    sum(new int[]{10,20,30,40});
  }
  P s v sum(int[] a)
  {
    int total=0;
    for(int a1:a)
    {
      total=total+a1;
    }
    S.o.p("The sum:"+total);
  }
}
```
DEMO

→ In the above example, just to call sum(_) method we required an array, but after completing that sum(_) method call we are not using that array anymore.

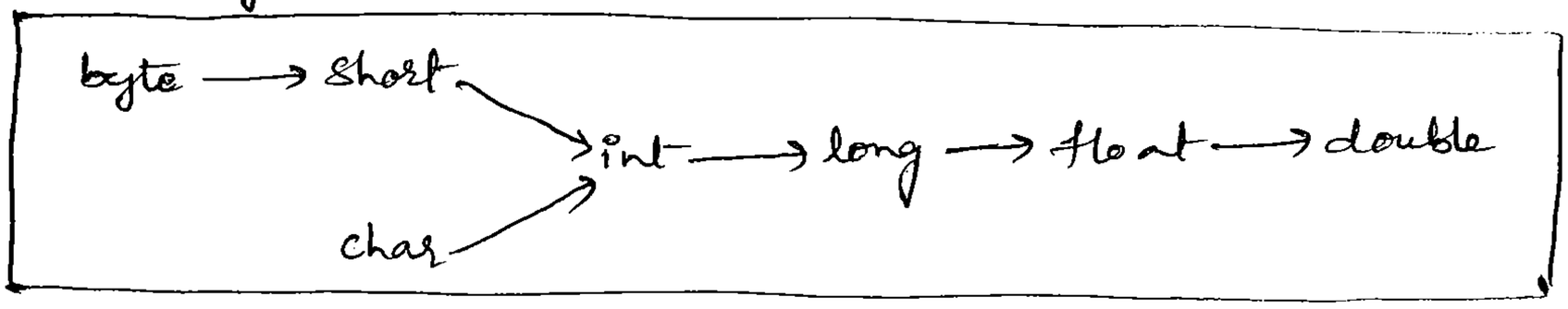
→ Hence anonymous array is the best choice for this requirement.

6) Array element Assignments :—

Case(i): For primitive type arrays as array elements we can provide any type which can be implicitly promoted to declared type.

Ex①: For int type arrays, the allowed element types are byte, short, char and int.

Ex②: For float type arrays, the allowed element types are byte, short, char, int, long and float.



Ex:
```
int[] a = new int[6];
✓ a[0] = 10;
✓ a[1] = 'a';
  byte b = 10;
✓ a[2] = b;
  short s = 20;
✓ a[3] = s;
✗ a[4] = 10.5;
```

DEMO

CE: PLP
found: double
required: int

Case(ii): For Object type arrays as array elements we can provide either declared type objects or its child class objects.

Ex①:
```
Object[] a = new Object[10];
✓ a[0] = new Object();
✓ a[1] = new String("durga");
✓ a[2] = new Integer(10);
```

Ex②:
```
Number[] n = new Number[10];
✓ n[0] = new Integer(10);
```

✓ n[1] = new Double(10.5);

✗ n[2] = new String("durga");

cc: incompatible types
found: j.l.String
required: j.l.Number

Case(iii): For interface type arrays as array elements we can provide its implementation class objects.

Ex: Runnable[] r = new Runnable[10];

✓ r[0] = new Thread();

✗ r[1] = new String("durga");

cc: incompatible types
found: j.l.String
required: j.l.Runnable

Array type	Allowed element type
1. Primitive type arrays	Any type which can be implicitly promoted to declared type.
2. Object type arrays	Either declared type or its child class objects
3. abstract class type arrays	Its child class objects are allowed
4. interface type arrays	Its implementation class objects are allowed.

DEMO

9) Arrays Variable Assignments :-

Case(i): Element level promotions are not applicable at array level.

For Example, char element can be promoted to int type, but char[] can't be promoted to int[].

Ex: int[] a = {10, 20, 30, 40};

char[] ch = {'a', 'b', 'c', 'd'};

✓ int[] b = a;

✗ int[] c = ch;

cc: incompatible types
found: char[]
required: int[]

Q: Which of the following promotions will be performed automatically?

✓ ① char → int

✗ ② char[] → int[]

✓ ③ int → double

✗ ④ int[] → double[]

✗ ⑤ float → long

✗ ⑥ float[] → long[]

✓ ⑦ String → Object

✓ ⑧ String[] → Object[]

→ But in case of Object type arrays child class type array can be assigned to parent class type array reference variable.

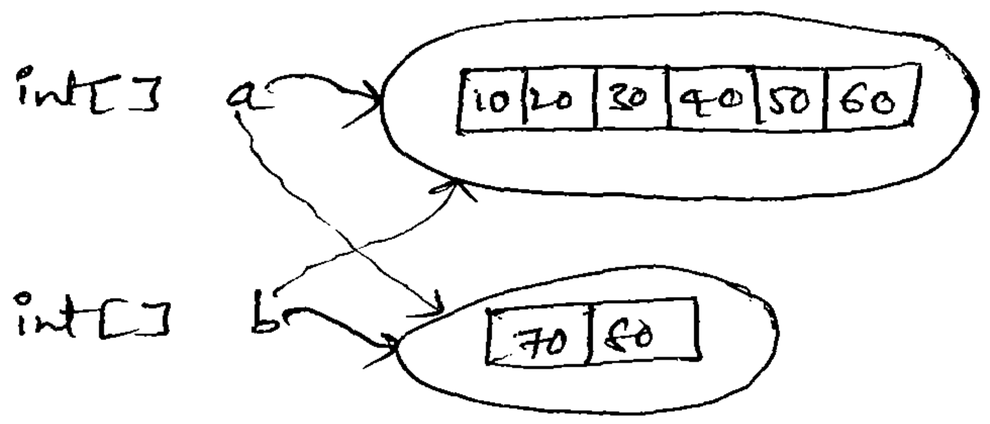
Ex: String[] s = {"A", "B", "C"};

Object[] o = s;

Case(ii): Whenever we are assigning one array to another array internal elements won't be copied, just reference variables will be reassigned. and hence sizes are not required to match, but types should be matched.

Ex: int[] a = {10, 20, 30, 40, 50, 60};

int[] b = {70, 80};



✓ ① a = b;

✓ ② b = a;

Case(iii): Whenever we are assigning one array to another array dimensions should be matched i.e., if we are expecting 1-D array we should provide 1-D array only. By mistake if we are providing any other dimension then immediately we will get CE.

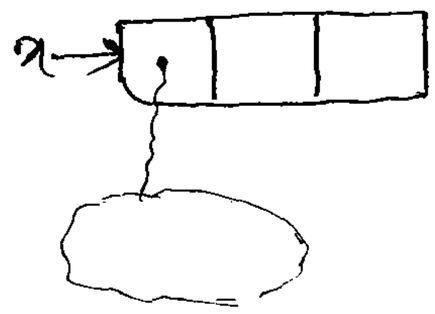
Ex: int[][] a = new int[3][];

✗ a[0] = new int[4][2];

CE: incompatible types
found: int[][]
required: int[]

```
X a[0] = 10;
```

ce: incompatible types
found: int
required: int[]



```
✓ a[0] = new int[4];
```

Note:- Whenever we are assigning one array to another array types and dimensions should be matched, but sizes are not required to match.

Ex:
```
class Test
{
  p s v main(String[] args)
  {
    String[] argh = {"A", "B", "C"};
    args = argh;
    for(String s : args)
    {
      S.o.p(s);
    }
  }
}
```

.java Test X Y Z ↵

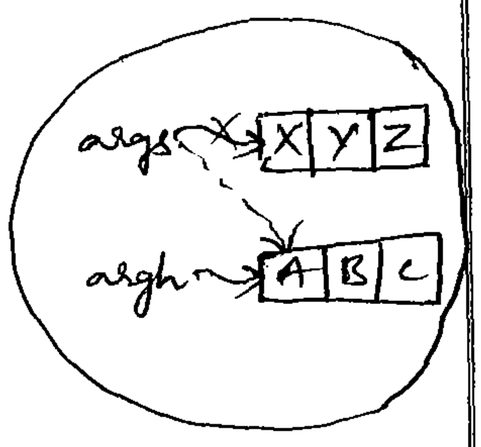
olp: A
B
C

java Test X Y ↵

olp: A
B
C

java Test ↵

olp: A
B
C



DEMO

Ex:
```
class Test
{
  p s v m(String[] args)
  {
    for (int i=0; i<=args.length; i++)
    {
      S.o.p (args[i]);
    }
  }
}
```

java Test A B C ↵

olp: A
B
C
RE: AIOOBE

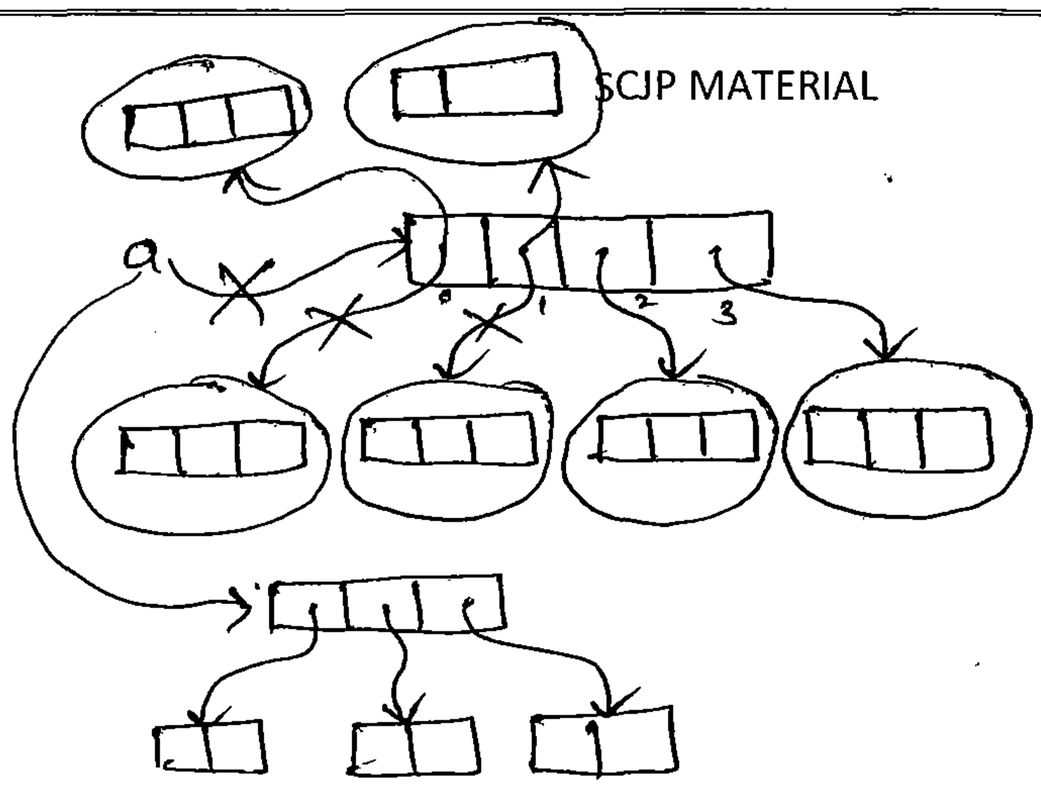
→ If we replace '<=' with '<' symbol then we won't get any Exception.

Ex: 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



Q: Total how many objects created?

Ans: 11

Q: How many objects eligible for GC?

Ans: 7

6) Types of variables :—

Division ①: Based on type of value represented by a variable all variables are divided into 2 types.

1. Primitive variables
2. Reference variables

1) Primitive variables :—

→ can be used to represent primitive values.

Ex: int x = 10;

DEMO

2) Reference Variables :—

→ can be used to refer objects.

Ex: Student s = new Student();

Division ②: Based on purpose & position of declaration all variables are divided into 3 types.

1. Instance variables
2. Static variables
3. Local variables.

1) Instance variables :—

→ If the value of a variable varied from object to object such type of variables are called Instance variables.

→ For every object a separate copy of instance variables will be created.

→ Instance variables will be created at the time of object creation and destroyed at the time of object destruction. Hence the scope of instance variables is exactly same as the scope of object.

→ Instance variables will be stored in the Heap memory as the part of object.

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

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

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

Ex:
```
class Test
{
    int a=10;
    p s v m(-)
    {
      S.o.p(a);
      Test t=new Test();
      S.o.p(t.a); ⇒ o/p : 10
    }
    public void m1()
    {
      S.o.p(a); ⇒ o/p : 10
    }
}
```

DEMO

CE : non-static variable cannot be referenced from a static context

→ For instance variables JVM will provide default values and we are not required to perform initialization explicitly.

Ex:
```
class Test
{
    int a;
    p s v m(-)
    {
      Test t=new Test();
      S.o.p(t.a); ⇒ o/p : 0
    }
}
```

→ Instance variables are also known as object level variables or attributes.

2) Static variables :—

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

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

→ static variables will be created at the time of class loading & destroyed at the time of class unloading. Hence the scope of static variables is exactly same as the scope of class file.

DEMO

java Test↵

1. Start JVM
2. Create & start main Thread
3. Locate Test.class
4. Load Test.class → static variables Creation
5. Execute main()
6. Unload Test.class → static variables destruction
7. Terminate main Thread
8. Shutdown JVM

→ static variables will be stored inside method area.

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

→ static variables can be accessed directly from both instance and static areas.

Ex:
```
class Test
{
  static int x=10;
  p s v main(-)
  {
    S.o.p(x); ⇒ o/p : 10
  }
  p v m1()
  {
    S.o.p(x); ⇒ o/p : 10
  }
}
```

→ static variables can be accessed either by object reference or by class name, but it is recommended to use class name.

→ Within the same class we can access static variables directly & it is not required to use class name.

Ex:
```
class Test
{
  static int x=10;
  p s v main(-)
  {
    Test t=new Test();
    S.o.p(t.x); ✓
    S.o.p(Test.x); ✓
    S.o.p(x); ✓
  }
}
```

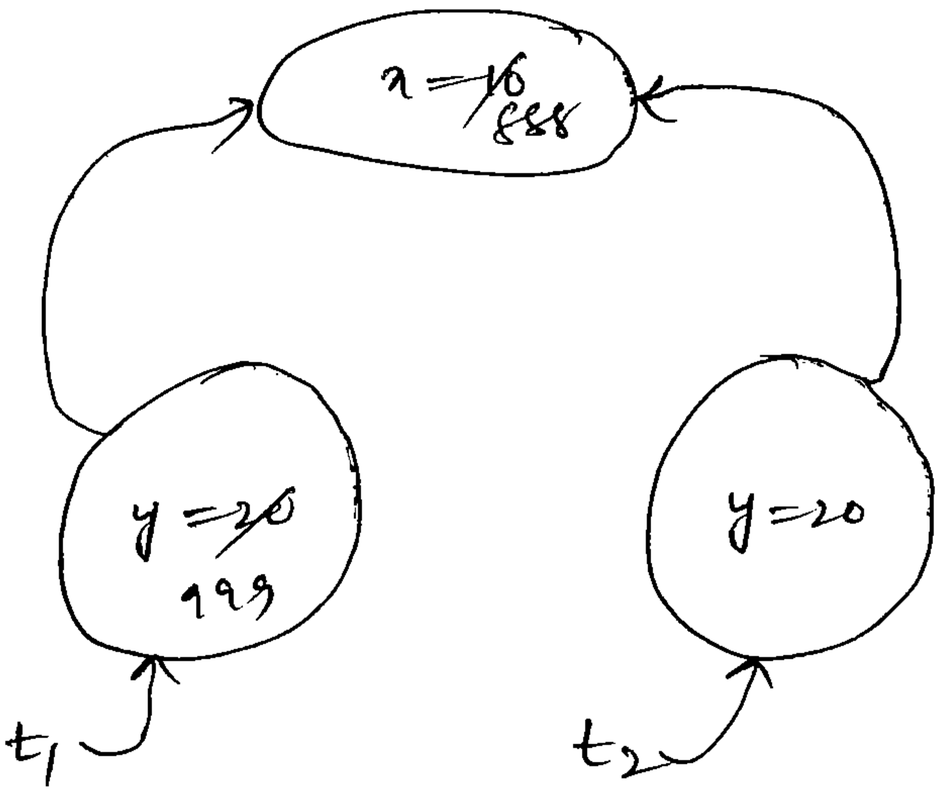
DEMO

→ For static variables we are not required to perform initialization explicitly, JVM will provide default values.

Ex:-
```
class Test
{
  static String s;
  static double d;
  p s v m(-)
  {
    S.o.p(s); ⇒ o/p : null
    S.o.p(d); ⇒ o/p : 0.0
  }
}
```

→ static variables are also known as <u>class level variables</u> or <u>fields</u>.

<u>Ex:</u>
```
class Test
{
    static int x=10;
    int y=20;
    P S V m(-)
    {
      Test t1=new Test();
      t1.x=888;
      t1.y=999;
      Test t2=new Test();
      S.O.p(t2.x+"---"+t2.y);
    }
}
```
<u>O/P</u>: 888 --- 20



<u>3) Local variables</u> :—

→ Sometimes to meet temporary requirements of the programmer we have to declare variables inside a method or block or constructor. Such type of variables are called <u>Local variables</u> or <u>Temporary variables</u> or <u>Automatic variables</u> or <u>stack variables</u>.

→ Local variables will be stored inside <u>Stack memory</u>.

→ Local variables will be created while executing the block in which we declared that variable.

→ Once that block execution completes automatically local variables will be destroyed. Hence the scope of local variable is exactly same as the scope of the block in which we declared that variable.

<u>Ex:</u>
```
class Test
{
  P S V m(-)
  {int i=0;
  try
  {
  int j=Integer.parseInt("ten");
  }
```

```
for(int j=0; j<3; j++)
 {
   i=i+j;
 }
 S.o.p(i+"---"+j);
}
}
```

```
catch(NFE e)
{
 j=10;
}
S.o.p(j);
```

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

→ For local variables, JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable

Ex:
```
class Test
{
 p s v m(-)
 {
  int x;
  S.o.p("Hello");
 }
}
```
o/p: Hello

```
class Test
{
 p s v m(-)
 {
  int x;
  S.o.p(x);
 }
}
```
DEMO

CE: variable x might not have been initialized

```
class Test
{
 p s v m(-)
 {
  int x;
  if(args.length>0)
  {
   x=10;
  }
  S.o.p(x);
 }
}
```

```
class Test
{
 p s v m(-)
 {
  int x;
  if(args.length>0)
  {
   x=10;
  }
  else
  {
   x=20;
  }
  S.o.p(x);
 }
}
```

java Test A B↵

o/p: 10

java Test↵

o/p: 20

Note:- ① It is never recommended to perform initialization for local variables inside logical blocks becoz there is no guarantee for the execution of these blocks always at runtime.

② It is highly recommended to perform initialization for local variables at the time of declaration atleast with default values.

***
Note:- The only applicable modifier for local variables is final.

By mistake if we are trying to use any other modifier we will get CE.

Ex:
```
class Test
{
    P s v m(-)
    {
        public int x=10;
        private int x=10;
        protected int x=10;
        transient int x=10;
        volatile int x=10;
        static int x=10;
        final int x=10;
    }
}
```

DEMO → CE: Illegal start of expression

Note:- If we are not declaring any modifier then it is <default> by default. But this rule is applicable only for instance and static variables but not for local variables.

Note:- For instance & static variables JVM will always provide default values and we are not required to perform initialization explicitly.

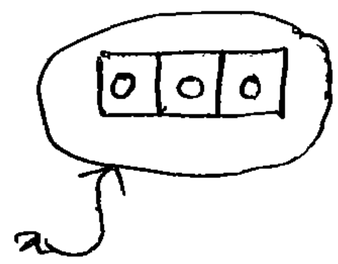
But for local variables JVM won't provide default values compulsory we have to perform initialization explicitly before using that variable.

Ex: class Test
{
int[] x;
P s v m(-)
{
Test t=new Test();
S.o.p(t.x); ⇒ o/p : null
S.o.p(t.x[0]); ⇒ RE : NPE
}
}

instance level:-

① int[] x;
S.o.p(obj.x); null
S.o.p(obj.x[0]); RE : NPE

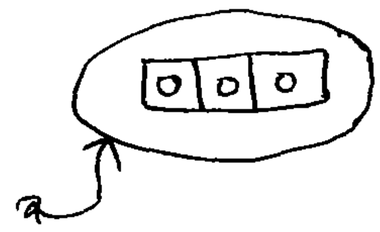
② int[] x=new int[3];
S.o.p(obj.x); ⇒ o/p : [I3e2595
S.o.p(obj.x[0]); ⇒ o/p : 0



static level:-

① static int[] x;
S.o.p(x); ⇒ o/p : null
S.o.p(x[0]); ⇒ RE : NPE

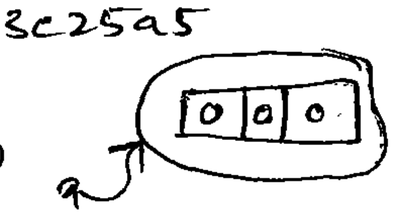
② static int[] x=new int[3];
S.o.p(x); ⇒ o/p : [I3e2595
S.o.p(x[0]); ⇒ o/p : 0



Local level:-

① int[] x;
S.o.p(x);
S.o.p(x[0]);
CE : variable x might not have been initialized

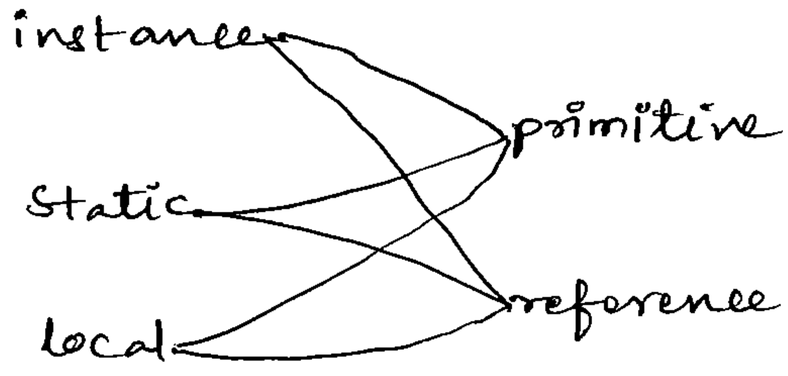
② int[] x=new int[3];
S.o.p(x); ⇒ o/p : [I3c25a5
S.o.p(x[0]); ⇒ o/p : 0



*** Note:- Once we created an array every element by default initialized with default values irrespective of whether it is static or instance & local array.

→ Every variable in Java should be either primitive or reference
→ Every variable in Java should be either instance or static or local.

→ Hence the following are various possible combinations of variables in Java.



Ex:
```
class Test
{
    static int i=10; ⟶ static – primitive
    int[] a=new int[3]; ⟶ instance – reference
    P S V m(-)
    {
      String s="durga"; ⟶ local – reference.
    }
}
```

Note:– Instance variables will be stored in the Heap area, static variables will be stored in Method area and local variables will be stored in Stack Memory.

***
7) var-arg method:–

→ Until 1.4 version we can't declare a method with variable no. of arguments.

→ If there is a change in no. of arguments compulsory we should declare a new method, which increases length of the code & reduces readability.

→ To overcome this problem SUN people introduced var-arg method concept in 1.5 version.

→ Hence from 1.5 version onwards we can declare a method with variable no. of arguments such type of methods are called var-arg methods.

→ We can declare a var-arg method as follows

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

→ We can call this method by passing any no. of int values including zero number also.

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

Ex:
```
class Test
{
  public static void m1(int... x)
  {
     S.o.p("var-arg method");
  }
  P S V m(-)
  {
     m1();
     m1(10);
     m1(10,20);
  }
}
```

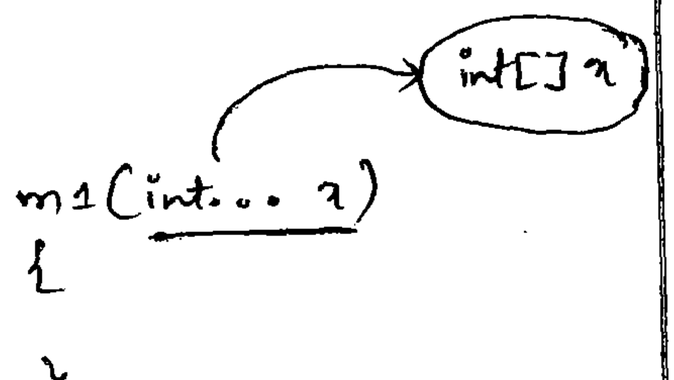
O/P : var-arg method
var-arg method
var-arg method

→ Internally var-arg parameter implemented by using 1-dimensional array.

DEMO

→ Hence within var-arg method we can differentiate arguments by using index.

Ex:
```
class Test
{
   P S V m(-)
   {
     sum();
     Sum(10,20);
     Sum(10,20,30);
   }
   P S V sum(int... x)
   {
     int total=0;
     for(int x1 : x)
     {
       total = total+x1;
     }
   } } S.o.p("The sum :"+total);
```



Case(i):

Q: which of the following var-arg method declarations are valid?

m1(int... x) ✓ → ellipse

m1(int ...x) ✓

m1(int x...) ✗

m1(int. ..x) ✗

m1 (int .x..) ✗

m1 (int...x) ✓

Case(ii): We can mix var-arg parameter with normal parameter also.

Ex: m1(int x, String... y) ✓

m1 (int x, int... y) ✓

Case(iii): If we mix var-arg parameter with normal parameter then var-arg parameter should be the last parameter.

Ex: m1 (String... s, double d) ✗ DEMO

m1(double d, String... s) ✓

Case(iv): In var-arg method, we can take only one var-arg parameter otherwise we will get CE.

Ex: m1(int... x, String... y) ✗

Case(v):

Ex:
```
class Test
{
  p s v m1(int x)
  {
   S.o.p("General method");
  }
  p s v m1(int... x)
  {
   S.o.p("var-arg method");
  }
  p s v main(--)
  {
   m1(); ⇒ O/P: var-arg method
   m1(10,20); ⇒ O/P: var-arg method
   m1(10); ⇒ O/P: General method
  }
}
```

→ In general var-arg method will get least priority i.e., if no other method matched then only var-arg method will get the chance. This is exactly same as default case inside switch.

Case(vi):

```
Ex: class Test
    {
      p s v m1(int[] x)
      {
      }
      p s v m1(int... x)
      {
      }
    }
```

int[] x

CE: Cannot declare both m1(int...) and m1(int[]) in Test

Equivalence b/w var-arg parameter and 1-D array:

Case(i): Whereever 1-D array present we can replace with var-arg parameter.

DEMO

Ex: m1(int[] x) ⇒ m1(int... x) ✓

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

Case(ii): Whereever var-arg parameter present we can't replace with 1-D array.

Ex: m1(int... x) ⇒ m1(int[] x) ✗

m1(int... x)	⇒	m1(int[] x) ✗
m1(new int{10,20,30})	⇒	✗
m1(new int{10,20})	⇒	✗
m1()	⇒	✗
m1(10,20)	⇒	✗
m1(10,20,30)	⇒	✗

Note①: m1(int... a)

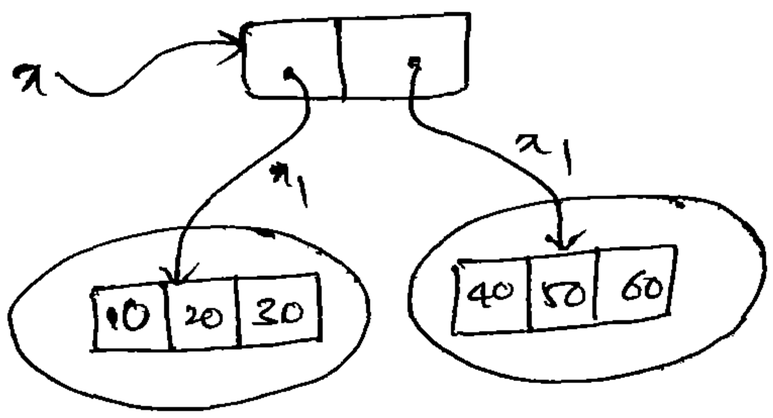
We can call this method by passing a group of int values and a will become 1-D int array. (int[] a).

Note②: m1(int[]... a)

We can call this method by passing a group of 1-D int arrays and a will become 2-D int array(int[][] a).

Ex:
```
class Test
{
  p s v m(-)
  {
    int[] a={10,20,30};
    int[] b={40,50,60};
    m1(a,b);
  }
  p s v m1(int[]... a)
  {
    for(int[] a1:a)
    {
      S.o.p(a1[0]); ⇒ o/p: 10
    }                          40
  }
}
```

DEMO



8) main(-) method:—

→ Whether the class contains main(-) method or not & whether main(-) method is properly declared or not these things won't be checked by compiler.

→ At runtime JVM is responsible to check these things.

→ At runtime JVM is unable to find required main(-) method then we will get RE saying, NoSuchMethodError: main.

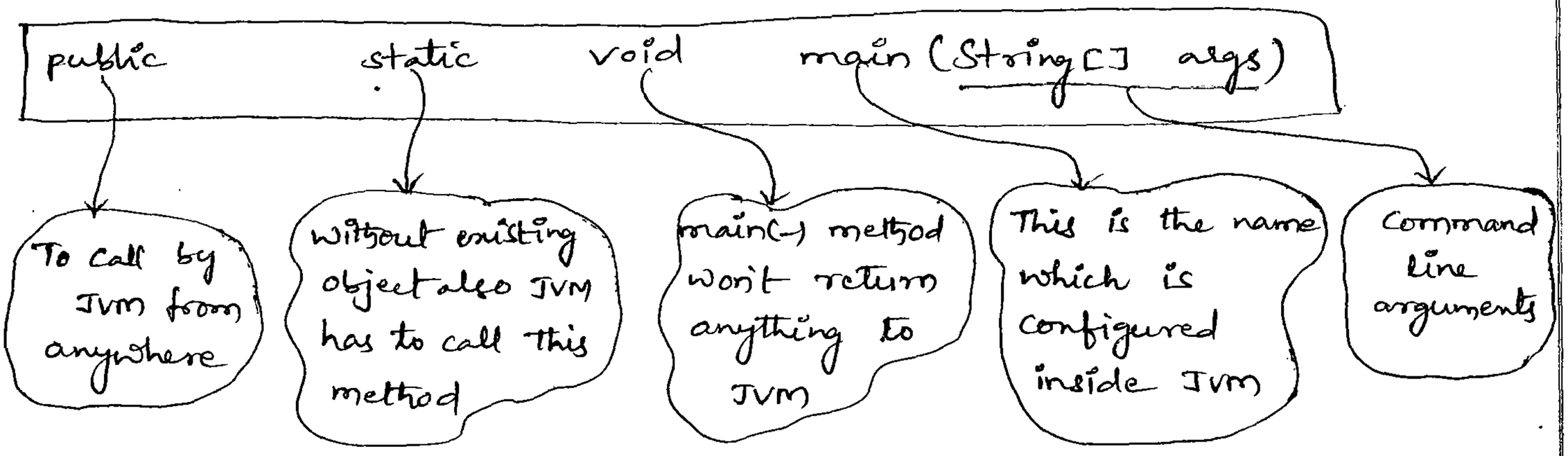
Ex:
```
class Test
{
}
```

javac Test.java↵

java Test↵

RE: NoSuchMethodError: main

→ At runtime JVM always searches for main(-) method with the following prototype.



→ If we perform any changes to the above syntax then we will get RE saying, NoSuchMethodError: main.

→ Eventhough the above syntax is very strict the following changes are acceptable.

① We can interchange modifiers order i.e., instead of public static we can take static public.

DEMO

② We can declare String[] in any acceptable form.

```
main(String[] args)
main(String []args)
main(String args[])
```

③ Instead of args we can take any valid Java identifier.

④ Instead of String[] we can take var-arg parameter.

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

⑤ We can declare main(-) method with the following modifiers also.

final
Synchronized
strictfp

Ex: class Test
{
static final synchronized strictfp public void main(String... durga)
{
S.o.p("Valid main method");
}
}
o/p : Valid main method.

Q: Which of the following are valid main(-) method declarations?

✗ ① public static void main(String args)
✗ ② public static void Main(String[] args)
✗ ③ public void main(String[] args)
✗ ④ public static int main(String[] args)
✗ ⑤ final synchronized strictfp public void main(String[] args)
✓ ⑥ final synchronized strictfp public static void main(String[] args)
✓ ⑦ public static void main(String... args)

DEMO

Q: In which of the above cases we will get CE?
Ans: We won't get compile time error anywhere.

Case(i): Overloading of the main(-) method is possible, but JVM will always call String[] argument main(-) method only. The other overloaded method we have to call explicitly then it will be executed just like a normal method call.

Ex: class Test
{
overloaded methods:
p s v main(String[] args)
{
S.o.p("String[]");
}
p s v main(int[] args)
{
S.o.p("int[]");
}
}

main(new int[]{10,20})

o/p : String[]

Case(ii): Inheritance concept is applicable for main(-) method. Hence while executing child class if child doesn't contain main(-) method then parent class main(-) method will be executed.

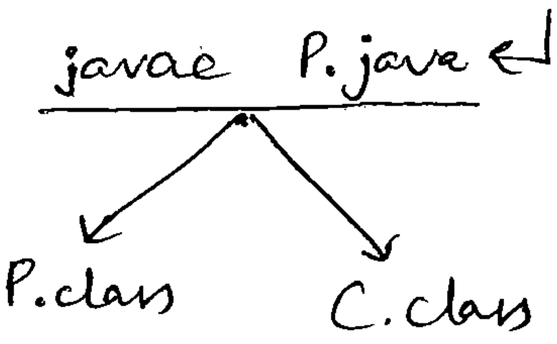
Ex:

P.java

```
class P
{
  p s v main(String[] args)
  {
    S.o.p("parent main");
  }
}
class C extends P
{

}
```

javac P.java ↵



java P ↵

o/p: parent main

java C ↵

o/p: parent main

Case(iii): It seems overriding concept applicable for main(-) method but it is not overriding, it is method hiding.

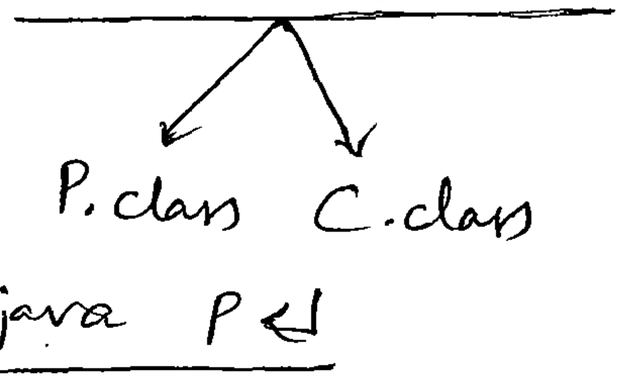
Ex:

P.java

```
class P
{
  P s v main(String[] args)
  {
    S.o.p("parent main");
  }
}
class C extends P
{
  P s v main(String[] args)
  {
    S.o.p("child main");
  }
}
```

It is method hiding but not overriding

javac P.java ↵



java P ↵

o/p: parent main

java C ↵

o/p: child main

1.7 version enhancements w.r.t main(-) method :—

→ If the class doesn't contain main(-) method until 1.6 version we will get RE saying, NoSuchMethodError: main.

→ But from 1.7 version onwards instead of NoSuchMethodError we will get more meaningful error information.

Ex: class Test
{
}

1.6 version	1.7 version
✓javac Test.java java Test RE: NoSuchMethodError: main	✓javac Test.java java Test Error: Main method not found in class Test, please define the main method as: public static void main(String[] args)

DEMO

→ Until 1.6 version to run a Java program main(-) method is not mandatory but from 1.7 version onwards main(-) method is mandatory.

→ Even though we are defining static block if the class doesn't contain main(-) method then it won't be executed.

Ex: class Test
{
static
{
S.o.p("Static block");
}
}

1.6 version	1.7 version
✓javac Test.java	✓javac Test.java

java Test ↵

o/p : static block

RE : NoSuchMethodError : main

java Test ↵

Error : Main method not found in class Test, please define the main method as :
public static void main (String [] args)

Ex:
```
class Test
{
   static
   {
     S.o.p ("static block");
     System.exit(0);
   }
}
```

1.6 version

javac Test.java ↵

java Test ↵

o/p : static block

1.7 version

javac Test.java ↵

java Test ↵

Error : Main method not ---

DEMO

Ex:
```
class Test
{
   static
   {
     S.o.p ("static block");
   }

   p s v main(String[] args)
   {
     S.o.p("main method");
   }
}
```

```
if(main method available)
 {
   normal flow
 }
else
 {
   Error information
 }
```

1.7 version

1.6 version

javac Test.java ↵

1.7 version

javac Test.java ↵

java Test↵

o/p : static block
main method

java Test↵

o/p : static block
main method

9) Command line arguments :—

→ The arguments which are passing from command prompt are called Command line arguments.

Ex: java Test A B C↵

args[0] ← A
args[1] ← B
args[2] ← C

args.length ⇒ 3

→ The main purpose of command line arguments is we can customize the behaviour of main(-) method.

DEMO

Ex①:
```
class Test
{
  p s v m(-)
  {
    for(int i=0; i<=args.length; i++)
    {
      S.o.p(args[i]);
    }
  }
}
```

java Test A B C↵

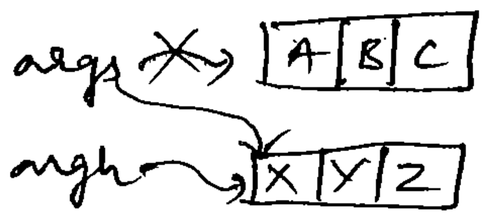
o/p: A
B
C
RE: AIOOBE

java Test A B↵

o/p: A
B
RE: AIOOBE

→ If we replace '<=' with '<' then we won't get any RE.

Ex②:
```
class Test
{
  p s v main(String[] args)
  {
    String[] argh = {"x","y","z"};
    args = argh;
```



```
        for (String s : args)
        {
            S.O.P(s);
        }
    }
}
```

java Test A B C ↵

O/P : X
Y
Z

java Test A B ↵

O/P : X
Y
Z

java Test ↵

O/P : X
X
Z

Eg②:

→ Within the main(-) method command line arguments are available in the form of String.

```
class Test
{
    p s v main(String[] args)
    {
        S.O.P(args[0]+args[1]);
    }
}
```

java Test 10 20 ↵

O/P : 1020

→ Usually space is the separator b/w command line arguments. If our command line arguments itself contains space then we should enclose that argument within double quotes.

Eg:③

```
class Test
{
    p s v main(String[] args)
    {
        S.O.P(args[0]);
    }
}
```

java Test "Note Book" ↵

O/P : Note Book.

10) Java Coding Standards :—

→ It is highly recommended to follow coding standards.

→ Whenever we are writing any class or method its name should reflect the purpose of that component.

→ This approach improves readability & maintainability of the code.

Ex:
```
class A
{
 public int m1(int i, int j)
 {
   return i+j;
 }
}
```
Ameerpet standard

```
package com.durgasoft.scjp;
public class Calculator
{
 public static int add(int num1,
                       int num2)
 {
   return num1+num2;
 }
}
```
Hitech city standard

Coding standards for classes :—

→ Usually class names should be nouns.

→ Should starts with uppercase character & if it contains multiple words every inner word should starts with uppercase character.

DEMO

Ex:
- String
- StringBuffer
- StringBuilder
- Student
- Customer
- Dog
- ⋮

⟹ nouns

Coding standards for interfaces :—

→ Usually interface names are adjectives.

→ Should starts with uppercase character & if it contains multiple words every inner word should starts with uppercase character.

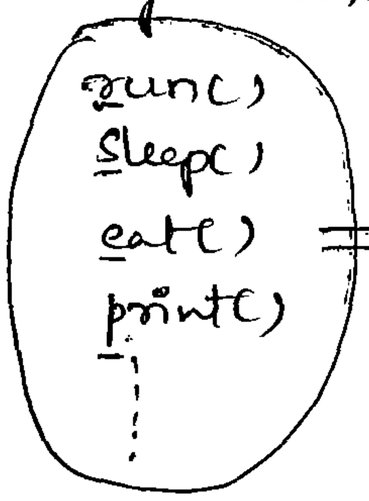
Ex:
- Serializable
- Runnable
- RandomAccess
- SingleThreadModel
- ⋮

⟹ Adjectives

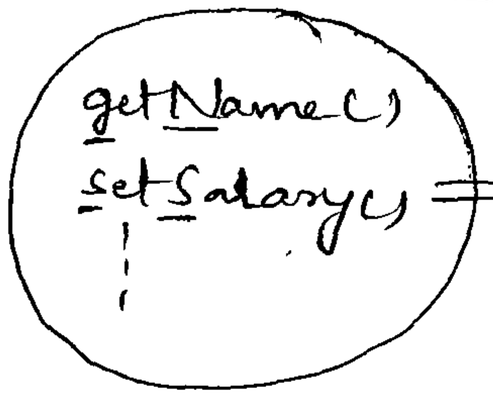
Coding standards for methods :—

→ Usually method names are either verbs or verb-noun combination.

Ex:



⟹ verbs



⟹ verb-noun

→ Method names should starts with lowercase character & if it contains multiple words every inner word should starts with uppercase character. (Camel-case convention).

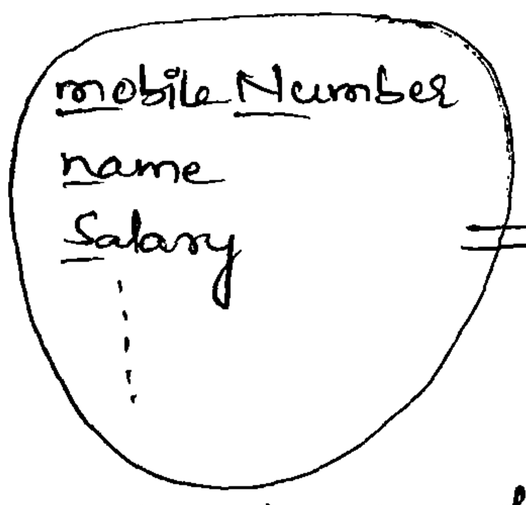
Coding standards for variables :—

→ Usually variable names are nouns.

→ Should starts with lowercase character & if it contains multiple words every inner word should starts with uppercase character. (Camel-case convention).

DEMO

Ex:



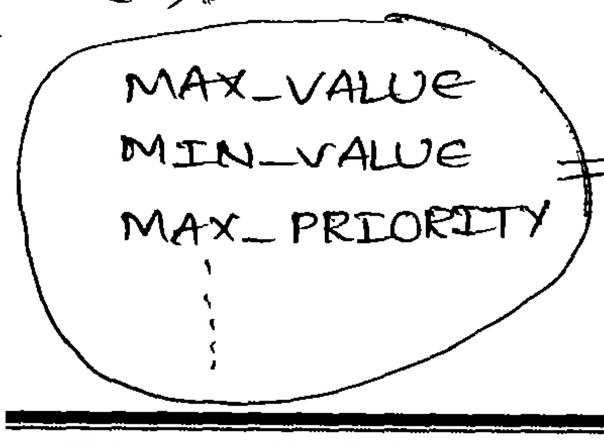
⟹ nouns.

Coding standards for constants :—

→ Usually constant names are nouns.

→ Should contain only uppercase characters and if it contains multiple words then each word is separated with underscore symbol (_).

Ex:



⟹ nouns.

Java Bean coding standards :—

→ A Java Bean is a simple Java class with private properties and public getter & setter methods.

Ex: 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 setter method:—

→ It should be public method.
→ return type should be void.
→ The method name should be prefixed with set.
→ Method should compulsory take some argument.

DEMO

Syntax for getter method:—

→ It should be public method.
→ The return type should not be void.
→ The method name should be prefixed with get
→ It should be no-argument method

*** Note:— For the boolean properties getter method name can be prefixed with either get or is and recommended to use "is"

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

Coding standards for Listeners :-

Case (i): To register a Listener :-

→ Method name should prefixed with add.

Ex: ✓ ① public void addMyActionListener(MyActionListener l)

✗ ② public void registerMyActionListener(MyActionListener l)

✗ ③ public void addMyActionListener(ActionListener l)

Case (ii): To unregister a Listener :-

→ The method name should prefixed with remove.

Ex: ✓ ① public void removeMyActionListener(MyActionListener l)

✗ ② public void unRegisterMyActionListener(MyActionListener l)

✗ ③ public void removeMyActionListener(ActionListener l)

✗ ④ public void deleteMyActionListener(MyActionListener l)

DEMO

DEMO