Language Fundamentals

(i) Identifiers

(ii) Reserved words

(iii) Data Types

(iv) Literals

(v) Arrays

(vi) Types of Variables

(vii) Var-arg method

(viii) main method

(ix) command-line arguments

(x) Java coding standard

Identifiers

Name in Java program is refferred as identifiers which can be used for ~~idef~~ identification purpose. It can be method name, variable name, class name or label name.

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

(5)

(i) Test = class name
(ii) main = method name
(iii) String = predefined class name
(iv) args
(v) x = name of variable

Rules :-

- Allowed: a to z, A to Z, 0 to 9, $, _ } If we use anything else we get compile time error
- Identifiers can't start with digit
  total123 ✓ 123 total ✗
- There is no length for Java identifiers.
- Reserved word can't be used as identifier :- Compile time error.

```
class Test {
    public static void main (String[] args) {
        int String = 8888; // int Runnable = 999;
        System.out.println(String);
    }
}
```

String → Predefined Class name
Runnable → Predefined Interface name

- All predefined Java Class Name or Interface name we can use as Identifier.

But not a good practice, reduces readability & increase confusion.

total_number ✓ ca$h ✓ Java2Share ✓ int ✗
total# ✗ _$_$_$_$ ✓ Integer ✓
123total ✗ call@hands ✗ Int ✓
→ Class name

Reserved Words

53 words, they represent some meaning or functionality.

(i) Keywords :- 50 → Used Keyword :- 48
→ UnUsed Keywords :- 2 (goto, const)

(ii) Reserved literals :- 3 → True, False, NULL (Reserved word representing only value)

→ Reserved word associated with functionality

Keywords for datatype (8)
byte short int long float double boolean char

Keywords for flow-control (11)
if, else, switch, case, default, while, do, for, break, continue, return

Keyword for modifiers (11)
public, private, protected, static, final, abstract, synchronized, native, strictfp, transient, volatile

Keyword for exception handling (6)
try, catch, finally, throw, throws, assert (1.4v)

Keyword for class related (6)
class, interface, extends, implements, package, import

Keyword for object related (4)
new, instanceof, super, this.

NOTE :- Delete keyword isn't there in Java, instead of delete there is a garbage collector ~~is~~ which is reponsible for deleting useless object.

In Java return type is mandatory, if a method wont return anything then we have to declare that method with void return type.

But in C language return type is optional and default return type is int.

Keyword for return type (1)
void.

Un-Used Keywords (2)
1. goto
2. const :- Final is used for this

SUN people band this keyword in Java

If we use this goto & const are used by mistake then we will get compile time error

- Reserved Literals (3)
  true } value for boolean type
  false

  null ⇒ default value for object reference

- Enum (1)
  We can use enum to defined a group named constant.

```
enum month
{
    Jan, Feb, ..... Dec;
}

enum Beer
{
    KF, RS, BL, BS;
}
```

Conclusion

(i) All 53 reserved keyword ~~lower~~ contains lower case value.

null → Java

NULL → C / C++

In C++, a NULL pointer is any literal zero value that is assiggned to a pointer. NULL is just a macro for 0 and macro are written in uppercase.

In java null is built in literal and Java there is no concept of literal. So no compulsion to make it uppercase

(ii) In Java we have only new keyword, no delete keyword because destruction of useless object is ~~only~~ responsibility of garbage collectors.

(iii) The following are new keyword in Java:-

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

(iv)

strictfp ✓	strictFp X
instanceof ✓	instanceOf X
synchronized ✓	synchronize X
extends ✓	extend X
implements ✓	implement X
import ✓	imports X
const ✓	constant X

Which the following list contain only Java reserved keyword ?

new, delete
goto, constant
break, continue, return, exit
final, finally, finalize
throw, throws, thrown
notify, notifyAll
implements, extends, imports
sizeof, instanceof
instanceOf, strictEp
None of the Above ✓

Spiral

Data - Types

All variables in Java has some data type.
All expression in Java has some data type.

1 byte :- 8 bits
Range :- -128 to 127

int x = 10.5; ✗
boolean b = 0; ✗
} Valid in C or more but not in Java

So Java is strongly typed language. Type checking is very strict. So each & every assignment is checked by compiler for type compatibility.

Is Java pure OOP language?

Yes compare with old language C, C++ then yes it is a pure object oriented language.

No but when compared alone, so it isn't considered pure OOP language because several OOP feature aren't supported by Java (Operator Overloading, Multiple Inheritance etc.). Moreover we are depending on primitive datatypes which isn't object.

Primitive datatypes (8)

- Numeric Datatypes
  - Integral datatype
    - byte
    - short
    - int
    - long
  - Floating data-type
    - float
    - double
- Non-Numeric Datatypes
  - char
  - boolean

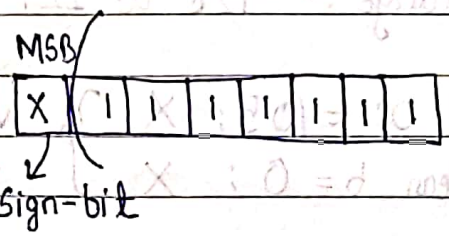
Except boolean & char remaining datatypes are considered as signed datatypes. We can represent both negative & positive number.

byte

Size:- 1 byte (8 bits)

Max-Value :- +127

Min-Value :- -128



The most significant bit act as sign-bit

0 → +ve

1 → -ve

Positive number is represented directly in memory but negative is represented as 2's complement

✓ byte b = 10;

✓ byte b = 127

✗ byte b = 128; → CE: Possible Loss of Precision found: int required: byte

✗ byte b = 10.5; → CE: PLOP Precision found: double required: byte

✗ byte b = true; → incompatible type → boolean

✗ byte b = "durga"; → CE: Possible loss of Precision ✗ Incompatible types found: Java.lang.String required byte

If we have data in form of streams so use byte.

File supported form or network supported form is byte.

Short

Size: 2 bytes

range -2^{15} to $2^{15}-1$

-32768 to 32767

Same compile type errors as bytes.

Short datatype is very suitable for 16 bit microprocessor like 8085. But these processor are completely outdated and corresponding short datatype is also outdated.

int
4 byte
-2^{31} to $2^{31}-1$
-2147483648 to 2147483648

X int x = 2147483648; → CE: Integer number too large
X int x = 2147483648l; → CE: Possible loss of precision
found: long
required: int
X int x = true; → CE: incompatible types
found: boolean
required: int

long

f.length() → Represent the no of characters in file
No of characters in big file may exceed int range. Hence 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 above data-types (byte, int, short, long) for representing integer value.

For floating point values, datatypes of floating point are required.

Floating Point Data Type

Float	Double
X Small floating point values	X Large floating point values
5-6 decimal places of accuracy	14-15 decimal places of accuracy
Single Precision → Less Accuracy	Double Precision → More Accuracy
Size 4 bytes	8 bytes
Range $-3.4e38$ to $3.4e38$ $e38 = 10^{38}$	$-1.7e308$ to $1.7e308$

Boolean → Virtual Machine

Size : NA [VM dependent]

Range : NA [but allowed values are :- true or false]

✓ boolean b= true;

X boolean b=0 → CE: incomptible types
found : int
required : boolean

X boolean b= True; → CE : ~~incompatible types~~
~~for~~ cannot find symbol
Symbol : variable True
location : class Test

X boolean b= "True"; → CE : incompatible type
found : java.lang.string
required: boolean

error in Java

Valid in C, C++, but compile time

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

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

CE: incompatible types
found: int
required: boolean

Char

C, C++ :- 1 byte
Java :- 2 bytes

Why??

Java is unicode (base) and C/C++ is ASCII (base). In C/C++ there are 256 character are there in ASCII set. So 2^8: 8 bits are enough to represent them.

But in Java there are 256 characters are ASII code values and other too, because they are unicode. They are greater than 256 and less than 65536. So 8 bits aren't enough. So we require 2 bytes or 16 bits.

Size: 2 bytes
Range: 0 to 65535

Summary

data-type	size	Range	Wrapper Class	default-Value
byte	1	-2^7 to 2^7-1	Byte	0
short	2	-2^{15} to $2^{15}-1$	Short	0
int	4	-2^{31} to $2^{31}-1$	Integer	0
long	8	-2^{63} to $2^{63}-1$	Long	0
float	4	$-3.4e^{38}$ to $3.4e^{38}$	Float	0.0
double	8	$-1.7e308$ to $1.7e308$	Double	0.0
boolean	NA	NA [But allowed values are true/false]	Boolean	false
char	2	0 to 65535	Character	0 (represents space character)

null is default value for object ~~type~~ reference
If null is applied on primitive data type then compile time occurs

Literals

```
int x = 10;
```

- int → datatype/keyword
- x → name of ~~identifier~~ variable / ~~variable~~ identifier
- 10 → constant value/literal

A constant value which can be assigned to a variable is called literal.

(i) decimal literal
int x = 10

(ii) Octal Form
int x = 010

(ii) Hexadecimal literal

Integer Literals

For Integer datatype (byte, short, int, long) we can specify literal in the following base :-

(i) decimal literals (base - 10)

int x = 10

(ii) octal form (base - 8) 0 - 7

int x = 010

(iii) Hexa-decimal form (base - 16)

int x = 0X10 0 to 9
a to f

For extra digit (a to f) we can use both lowercase & upper case characters. This is one of very rare area where Java isn't case sensitive.

✓ int x = 10

✗ int x = 0786; } 0 octal 0 - 7

✓ int x = 0777;

✓ int x = 0XFace; } 0X means Hexadecimal.

✗ int x = 0XBeer; } Only a - f

CE : Integer too Large

```
class Test {
    psvm ( String[] args) {
        int x = 10
        int y = 010
        int z = 0X10
        S.o.p.ln ("x + "....." + y + "....." + z);
}
}
```

JVM will print in decimal form → 10 8 16

By defaul every integer literal is of int type but we can specify explicitly as a long type by suffix with l or L.

✓ int x = 10;
✓ long l = 10 L;
int x = 10L; → Possible Loss of Precision
Found :- Long
Required :- Int

Floating Point Literals

✗ float f = 123.456 → CE : PLP
Found: double
require float

Every floating point literal is of double type.

✓ float f = 123.456 F; (Small f or Capital F)

✓ double d = 123.456

Hence, we can't assign directly to a float variable.

double d = 123.456D;

This convention isn't required because by default its double

✗ float f = 123.456d;

✓ double d = 123.456;
✓ double d = 0123.456; → It is treated as decimal literal.
✗ double d = 0X123.456
→ CE : malformed floating point literal

We can specify floating points literals only in decimal form and we can't specify in octal or hexadecimal form.

✗ double d = 0786; → CE Integer number too large
✓ double d = 0XFace;
✓ double d = 0786.0 ⇒ Output = 786.0
✗ double d = 0XFace.0 → CE: malformed floating point literal
✓ double d = 10;
✓ double d = 0777

We can assign integral literals directly to floating point variables under that integral literal can be specified either in decimal, octal or Hexadecimal form

✓ double d = 10
✗ int x = 10.0 CE: Possible loss of Precision
found : double
required : int

double d = 1.2e3; ✓
float f = 1.2e3; ✗ float f = 1.2e3F; ✓
↳ Found : double
required : float

We can specify floating point literal in exponential form

The only allowed values for boolean datatype are true or false.

✓ boolean b = true;
✗ boolean b = 0; → CE: incompatible Types
found : int
require boolean
✗ boolean b = True;
✗ boolean b = "true"; → CE: cannot find symbol
symbol : variable True
location : class Test

Spiral

Char literal

✓ char ch = 'a';
✗ char ch = a;
✗ char ch = "a";
✗ char ch = 'ab';

CE : cannot find symbol
Symbol : variable a
location : class Test

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

CE1: unclosed char literal
CE2: unclosed char literal
CE3: not a statement

char ch = 97; ✓

'a' ⇒ 97

(i) We can specify char letter as integer letter which represents unicode value of character.

Integer can be expressed in decimal, octal & Hexadecimal.

But allowed range is: 0 - 65535

char ch = 0XFace; ✓
char ch = 0777; ✓
char ch = 65535; ✓
char ch = 65536; → CE : Possible Loss of Precision
found : Int
required: char

(ii) We can represent char literal in unicode representation.

'\u$\underbrace{xxxx}_{\text{4 digit hexadecimal number}}$'

'\u0061' → a

(iii) Every escape character is a valid char literal

✓ char ch = '\n' char ch = '\t' ✓ char ch = '\m' CE : ✗
→ illegal escape character

Spiral

Escape Character	description
\n	New Line
\t	Horizontal tab
\r	Carriage return
\b	Back Space
\f	Form Feed
\'	Single quote
\"	double quote
\\	Back slash

Which of the following are valid ?

X char ch = 65536; X char ch = 0X Beer;

X char ch = \uface; ✓ char ch = '\ubaac';

X char ch = '\m'; X char ch = '\iface';

String

Any sequence of character within double quotes is treated as ~~single~~ string literals.

In 1.7 version enhancement of two literals

(i) Binary literals

For integeral datatype until 1.6 version, we can specify literal value in the following base

(1) Decimal (2) Octal (3) Hexadecimal

But from 1.7 version we can specify literal in binary form. Allowed digits are 0 and 1. Later value should be prefixed with b or B

```
int x = 0B 1111;
S.o.p(x); 15
```

(ii) Usage of underscore literals in numeric ~~character~~ literals. from 1.7 version between digit of numeric literals.

After compilation underscore is removed.

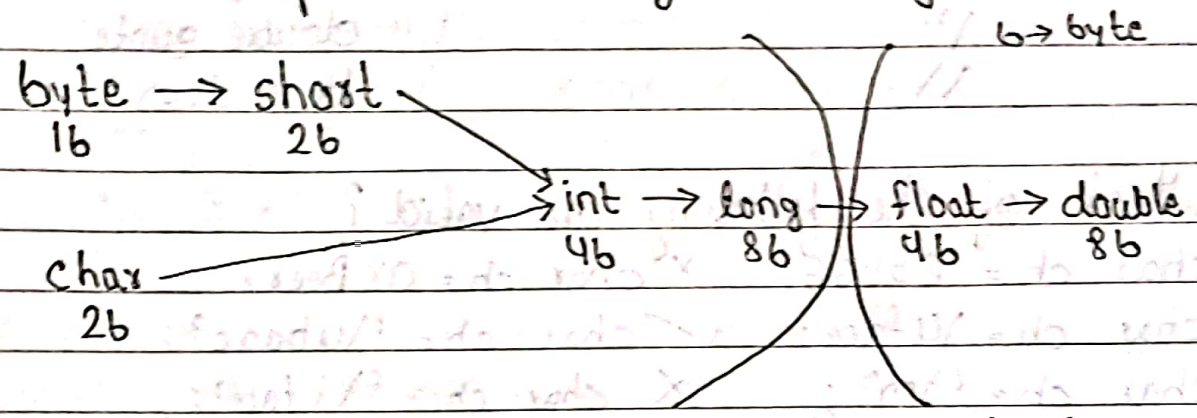
```
doubled= 123456.789
d=1_23_456.7_8_9
```

We can use more than one underscore within digit.

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

Underscore can only be used between two digits if we use it anywhere else then compile time error.

Lower datatype can be assigned to higher



- 8 byte long value can be assigned to 4 byte long variable because both are following different memory representation internally

  ✓ float f = 10l; ⇒ 10.0

- char & short are same byted. But both are different
  short is signed & char is unsigned.
  ↳ 15 ↳ 16
  1 → sign — No negative allowed
  Allowed

Arrays

(i) Introduction
(ii) Array declaration
(iii) Array creation
(iv) Array initialization
(v) Array declaration, creation & initialization in single line
(vi) length vs length()
(vii) Anonymous Arrays
(viii) Array elements assignments
(ix) Array variable assignments

Introduction

```
int[] x = new int[1000];
```

(i) Arrays are fixed in sized
(ii) Arrays are homogeneous

Array is indexed collection of fixed no of homogeneous data element.

Disadvantage

(i) Fixed in size :- Once we create array, no chance of increase or decrease of array size. To use array concept, we need to know size in advance.

Array declaration

```
int[]  x ;  → recommended, name is clearly seperated from type.
int    x[] x;
int    x[];
```

At the time of declaration, we can't specify the size we will get compile time error

```
int[6] x ;  ✗
int[]  x ;  ✓
```

2-D Array

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

Which of the following 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 → 1
int[] a,b;          a → 1 , b → 1
int[] []a,b[];      a → 2 , b → 3
int[] []a,b         a → 2 , b → 2
```

✗ int[] []a,[]b;

↳ If we are specifying dimension before the variable, the facility is available only for first variable in a declaration. If we are trying for remaining we will get compile time error

```
int[]  []a,[]b,[]c;
       ✓   ✗   ✗
```

Array Creation

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

(40:39)

Object are created using new operator.
So array is an object.

Every array type corresponding class are available.

S.O.pln(arr.getClass().getName()); → [I [[I

1D 2D

These classes are part of Java language not available to programmer level

Array Type	Class Name
int[]	[I
int[][]	[[I
double[]	[D
short[]	[S
byte[]	[B
boolean[]	[Z

- Size of array shouldn't be at time of declaration, at the time of creation compulsory we should specify size

int[] x = new int[]; ✗

int[] x = new int[6]; ✓

declaration creation

- It is legal to have size zero in java

int[] x = new int[0];

- Negative Array Size doesn't give compile time error but run time exception saying Negative Array Size Exception.

To specify array size allowed datatypes are
byte, short, char, int

int [ ] x = new int [10]; ✓
↑
'a'

byte b = 20; b ✓
short s = 30; s ✓

10l X

CE: PLP
Found : long
required : int

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

Array Max_Size

int[ ] x = new [2147483647]; ✓
int[ ] x = new [2147483648] X

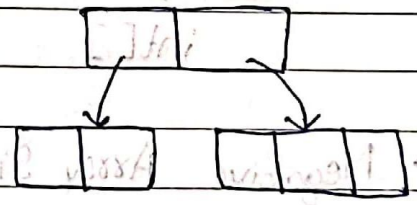
CE: integer number too large

The maximum array size allowed in Java but may give Runtime Exception due to system resources

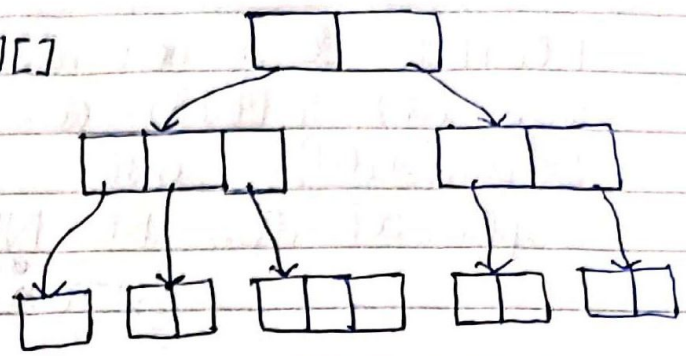
2-D Array Creation

In Java 2-D Array not implemented by using matrix style. We do this by arrays of arrays approach and this leads to a improved memory utilization.

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



```
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];
```



```
X int[][][] a = new int[3][][5]
X                   [][4][5]
```

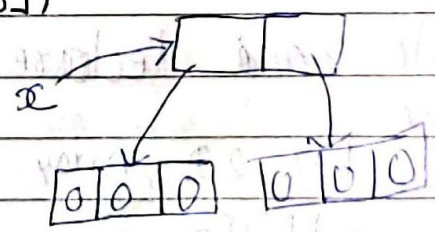
```
int[] x = new int[3]
Sopln(x); [I@3e25em5
Sopln(x[0]): 0
```

Once we create an array, every element in array by default initialised to zero.

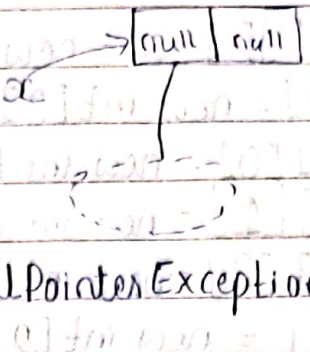
Whenever we are trying to print reference variable, internally to string method will be called, to return the string in following form.

classname @ hashCode_in_hexadecimal form.

```
int[][] x = new int[2][3];
Sopln(x); [[I@3edm53
Sopln(x[0]); [I@4892F
Sopln(x[0][0]); 0
```



```
int[][] x = new int[2][];
Sopln(x);          [[64Imed
Sopln(x[0]);       null
Sopln(x[0][0]);    RE : NPE
```



↳ java.lang.NullPointerException

```
int[] x = new int[6];
```

Runtime { x[6] = 70 ; RE : ArrayIndexOutOfBoundException
x[-6] = 80 ; RE : "

Compile time { x[2.5] = 40; CE : PLP

If we are trying to access element out of range index (+ve or -ve) int value, then we will get Runtime exception- ArrayIndexOutOfBound Exception.

Array Initialisation & Declaration

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

~~int[3]~~ int[] x = {10,20,30};

We can declare, create and initialise array in single line.

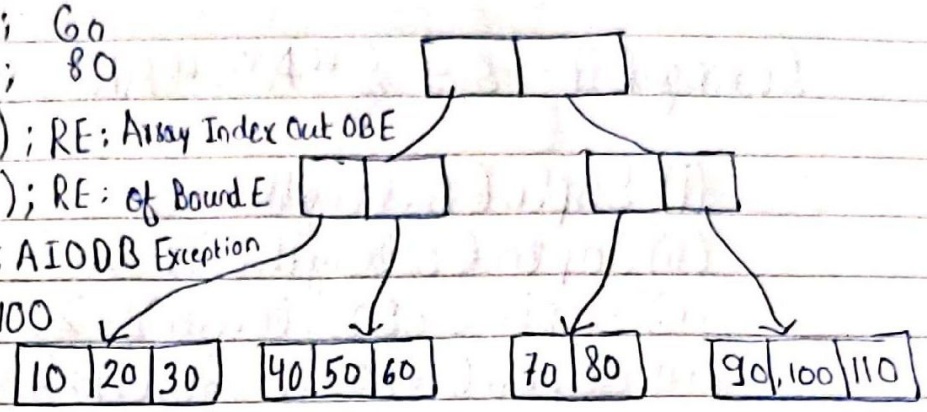
For 2D Array

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

3-D Array

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

```
Sopln(x[0][0][0]); 60
Sopln(x[1][0][1]); 80
Sopln(x[1][0][0]); RE: Array Index Out OBE
Sopln(x[1][2][0]); RE: of Bound E
Sopln(x[2][1][1]); AIOOB Exception
Sopln(x[1][1][0]); 100
```



✓ `int [] x = {10,20,30};`

✗ `int[] x;`
`x = {10,20,30};`
CE: Illegal sort of expression

length vs length()
↳ Variable ↳ method

```
int [] x = new int[6];
```

✗ `Sopln(x.length());` → CE: Cannot find symbol
symbol : method length
location : class int[]

✓ `Sopln(x.length);` ⇒ 6

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

```
String s = "saurav";
```

✗ `Sopln(s.length);` → CE: cannot find symbol
symbol : variable length
location : class java.lang.String

✓ `Sopln(s.length());` ⇒ 6

length method is a final method applicable for string object. length method returns no of character in string

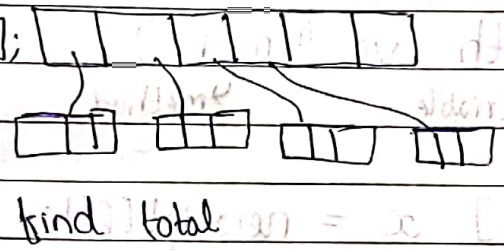
length :- Applicable for arrays
length() :- Applicable for String
↳ method
→ variable

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

(i) Sopln(s.length); ✓
(ii) Sopln(s.length()); ✗
(iii) Sopln(s[0].length); ✗
(iv) Sopln(s[0].length()); ✓

In multi-dimensional length variable returns base size not total length.

int[][] x = new int[6][3];



Sopln(x.length); 6
Sopln(x[0].length); 3

There is no direct way to length of multi-dimensional array.

Anonymous Array

Sometime we can declare an array without name, such type of nameless array is called Anonymous Arrays. The main purpose of Anonymous Array is just for Anonymous use.

We can create anonymous array as follows:-

new int[]{10, 20, 30, 40}; ✓
new int[3]{10, 20, 30}; ✗ CE

While creating anonymous array we can't specify size. Otherwise we will get compile time error.

Spiral

2-D Anonymous Array

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

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

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

```
class Test {
    public static void main(String[] args) {
        sum(new int[] {10,20,30});
    }
    public static void sum(int[] x) {
        int total=0;
        for(int x1 : x) {
            total+=x1;
        }
        Sopln("The sum is" + total);
    }
}
```

In the above example to call ~~some~~ sum method we required array but after calling sum method we don't require array anymore. Hence anonymous array is best choice.

In the case of object type arrays as array element we can provide either declare type object or child class object

eg1

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

eg2

abstract

```
Number[] n = new Number[10];
n[0] = new Integer(10);
n[1] = new Double(10.5);
n[2] = new String("Saurav");
```

CE: incompatible type
found: java.l.string
required: java.l.number

Number
B S I L F D

Runnable :- Interface type
Interface type array object can be created but interface object can't be created.

Every implementation class by default is of type interface class.

Runnable is an interface. So we can't create object. It's implementation class Thread is of Runnable type.

Runnable

~~Runnable[] r = new~~

Thread

For interface type arrays, as array elements it implementation class objects are allowed.
type

```
Runnable [ ] r = new Runnable[10];
r[0] = new Thread();
r[1] = new String("Saurav");
```

CE: incompatible types
found: java.l.string
required: j.l.runnable

Array Type	Allowed element types
primitive Arrays	Any type which can be implicitly promoted to declared type
object type Arrays	Either declared type or it's child class objects
abstract class type arrays	All child class object
interface type array	It implementation class objects are allowed.

Arrays Variable Assignment

Element level promotion aren't applicable at array level.

for eg:-

char element can be promoted to int type
char array can't be promoted to int array

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

~~int E~~
✓ int [] b = x;
✗ int [] c = ch;
```

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

In the case of object type arrays, child class type array can be promoted parent class type array.

String → Object ✓
String[] → Object[] ✓

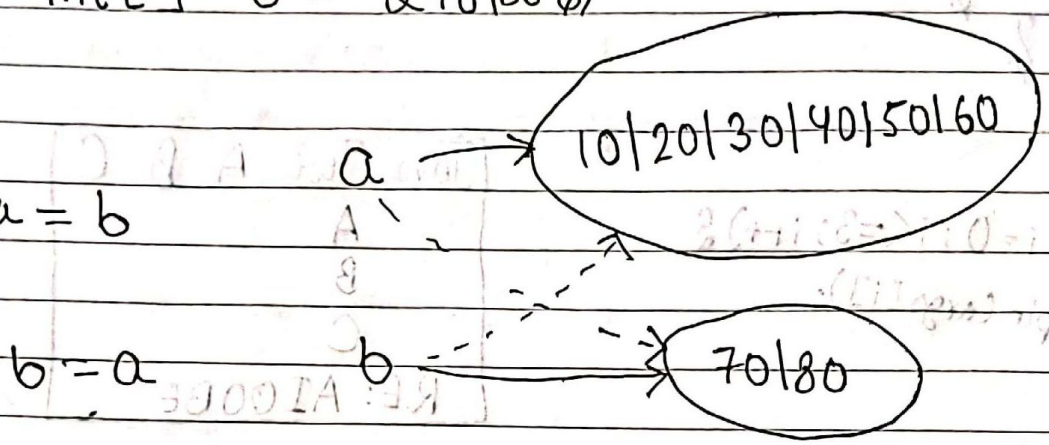
```
String[] s = {"A", "B", "C"};
Object[] a = s;
```

char → int ✓ int → double ✓
char [] → int [] ✗ int [] → double [] ✗

float → int ✗
float[] → int[] ✗

Whenever we are assigning one array to another array. Internal elements aren't copied just reference variable will be reassigned.

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



21:40

Whenever we are assigning one array to another array the ~~Intern~~ dimension must be matched.

For ex:- In the place one-dimensional int array we should provide one-dimensional array only. If we are trying to provid another dimension we will get compile time error.

```
int[][] a = new int[3][]
X a[0] = new int[4][3]
```

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

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

CE:
found: int
required: int[]

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

Whenever we are assigning one array to another array both dimension and types must be matched, but sizes aren't required to match.

```
class Test {
    p s v main( String[] args){
        for(int i=0 ; i<=args.length ; i++) {
            Sopln(args[i]);
        }
    }
}
```

```
for (int i=0 ; i<=3; i++) {
    Sopln(args[i]);
}
```

Java Test A B C
A
B
C
RE: AIOOBE

```
args[0] = A
args [1] = B
args [2] = C
args [3] = AIOOBE
```

Java Test A B
A
B
RE: AIOOBE

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

Java Test
RE: AIOOBE

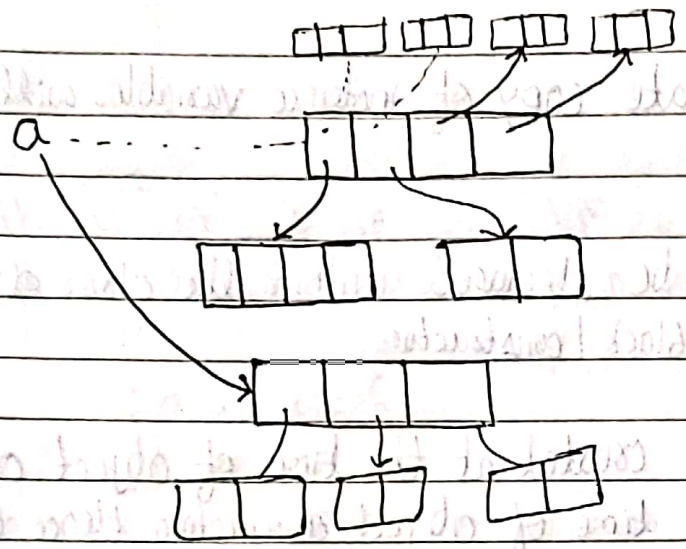
X
Y
Z

in all three cases

```
int [][] a = new int [4] [3]; ---(5)
a[0] = new int[4] ; ---(1)
a[1] = new int[2] ; ---(2)
a = new int[3] [2]; ---(4)
```

(i) Total how many object created? 11

(ii) Total how many object eligible for garbage collection? 7



Types of Variables (10)

(i) int x = 10 (ii) Student s = new Student();

s →

Based on type of values represented by ~~this~~ a variables all variables are divided into two types.

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

(ii) Reference Variables can be used to refer objects.

Based on the position of declaration and the behaviour all variables are divided into three types:-

(i) instance variables

(ii) static variables

(iii) local variables

Instance Variable

```
class Student{
    String name;
    int rollno;
    :
    :
}
```

If the value of a varied from object to object such ~~that~~ type of variables are called instance variables

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

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

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

Instance variable will be stored in the heap memory as a part of object.

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 = 10;
    psv main(String[] args) {
        Sopln(x);
        Test t = new Test();
        Sopln(t.x);
    }
    public void m1() {
        Sopln(x);
    }
}
```

→ CE: non-static variable x cannot be reference from a static context

For instance variables JVM will always provide default values and we aren't required to perform initialisation explicitly

```
class Test {
    int x;
    double d;
    String s;
    boolean b;
psv main(String[] args) {
        Test t1 = new Test();
            Sopln(t1.x); ⇒ 0
            Sopln(t1.d); ⇒ 0.0
            Sopln(t1.b); ⇒ false
            Sopln(t1.s); ⇒ null
    }
}
```

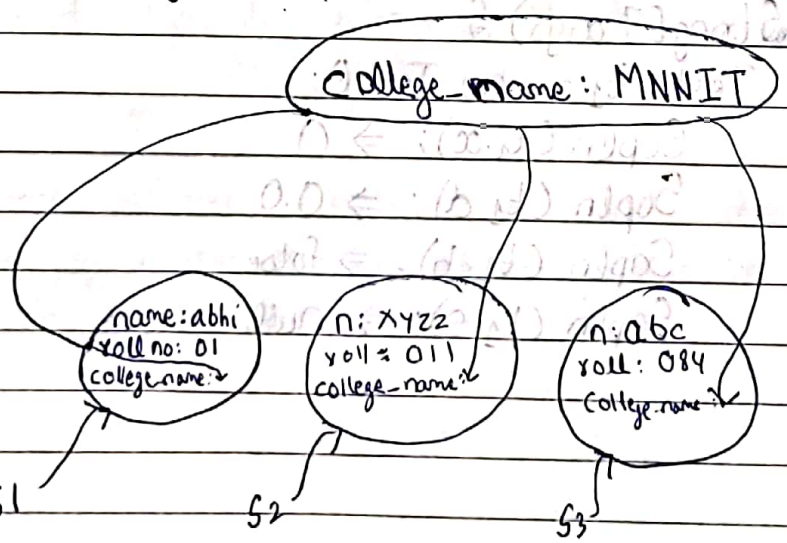
Instance variables are also known as object level variables or attributes.

Static Variables

If the value of the variables aren't varied from object to object, then it isn't recommended to declare variable as ~~static~~ instance variable. We have to declare such type of variable at class level by using static modifier.

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

```
class Student {
    string name;
    int rollno;
    static String college_name;
}
```



Static variable should be declared within the class directly but outside of any method block or constructor. Static variable will be created at the time of class loading and destroyed at the time of class unloading. Hence, scope of static variable is exactly same as scope of ~~object~~ .class file.

Java test ↵

(i) Start JVM
(ii) Create & start main thread
(iii) Locate test.class file
(iv) Load test.class ⟶ Static variables creation
(v) Execute main() method
(vi) Unload test.class ⟶ Static variables destruction
(vii) Terminate main thread
(viii) Shutdown JVM

Static variable will be stored in method Area.
NOTE :- Local variables are stored in stack.

We can access static variable either by object reference or class name but recomended to use class name but within the class it isn't required to use class name and we can access it directly.

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

We can access static variable directly from both instance & static areas.

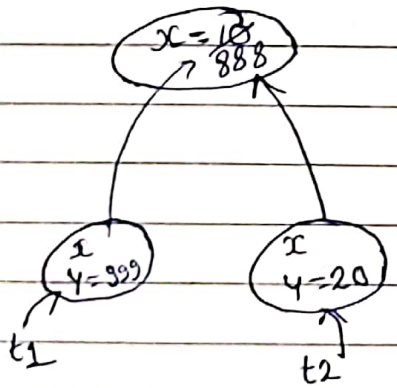
```
class Test {
    static int x = 10;
    p s v main (String[] args) {
        Sopln(x);
    }
    public void m1() {
        Sopln(x);
    }
}
```

Even in static variable JVM will provide default values, we aren't required, the initialisation isn't performed explicitly

Static variables are also known as class level variables or fields

Static variable has only one copy. By using any object reference if we change, the change will reflect automatically for all.

But instance variable, every object seperate copy, if i change for any object instance variable it won't reflect for other object because every object has own instance variable.




```
class Test {
    static int x = 10;
        int y = 20;
    psv main (String[] args) {
        Test t1 = new Test();
        t1.x = 888;
        t1.y = 999;
        Test t2 = new Test();
        Sopln(t2.x + "....." + t2.y);   888 ..... 20
    }
}
```

Local Variables

Sometimes to meet temporary requirement of programmer we can declare variables inside a method or block or constructor. Such type of variables are called local variables or temporary variables or stack variables or 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 completes automatically local variables will be destroyed. Hence Scope of local variables is the block in which we declare it.

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

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

For Local variables JVM wont provide default values compulsorily we should provide initialisation explicitly before using that variable that is if we aren't using it initialisation isn't required

```
class Test{
    psv main(String[] args){
        int x;
        sopln(x);
    }
}
```

CE: variable x might not have been initialized

Spiral

```
class Test {
    psv main(String[] args) {
        int x;
        if (args.length >0) {
            x=10;
        }
        Sopln(x);
    }
}
```

CE: Variable x might not have been initialized.

```
class Test {
    psv main( String[] args) {
        int x;
        if (args.length > 0) {
            x=10;
        } else {
            x = 20;
        }
        Sopln(x);
    }
}
```

Java Test A B
O/P 10

Java Test
O/P 20

It isn't recommended to perform initialisation for local variable inside logical blocks, because there is no guarantee of ~~initialization~~ execution of these block always at run time.

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

The only applicable modifier for local variable is final. By mistake if we are trying to apply any other modifier then we will get compile time error.

```
class Test {
    psv main (String[] args) {
        public int x = 10;
        private int x = 10;
        protected int x = 10;
        static int x = 10;
        transient int x = 10;
        volatile int x = 10;
        final int x = 10;
    }
}
```

CE : illegal start of expression

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

Conclusion

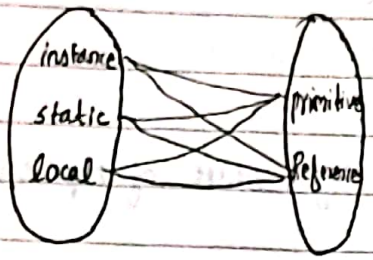
(i) For instance & static variable JVM will provide default values and we aren't required to perform initialisation explicitly but for local variables initialisation is mandatory before use.

(ii) Instance & static variables can be accessed by multiple thread simultaneously and hence these aren't thread safe but incase of local variable a seperate copy are created hence they are thread safe.

Type of variable	Is Thread Safe
instance variable	✗
static variable	✗
local variable	✓

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

Hence various possible combination of variables in Java are :-




```
class Test{
    int x = 10; ←— instance-primitive
    static string s = "dasgali"; ←— static reference
    psv main (String[] args){
local reference → int[] y = new int[3];
    }
}
```

```
class test{
    int [] x;
    psv main (String[] args){
        Test t = new Test();
        Sopln(t.x); null
        Sopln(t.x[0]; X  RE : null pointer exception
    }
}
```

Instance level

(i)

```
int[] x;
obj.x; ⇒ null
obj.x[0]; ⇒ RE : NP
```

(ii)

```
int [] x = new int[3];
Sopln (obj.x); [I@3e25a
Sopln (obj.x[0]); 0
```

Spiral

① Static Level

```
Static int[] x ;
Sopln(x) ; null
sopln(x[0]; RE : NPE

static int[] x = new int[3]
sopln(x) ; [I@e35
Sopln(x[0]); 0
```

③ Local Level

```
psv main (String[] args) {
    int [] x = new int[3];
    Sopln (x) ; [I @3e 25a5
    Sopln (x[0]); 0
```

NOTe :- Once we creat an array, every array element initialise by default with zero irrespective of local, instance or static array.

Var-arg methods (Variable number of argument methods)

1.5 version, this concept came

Until 1.4 version, we can't declare a method with the variable number of argument. For changes in argument we need to go for new method. It increases length of the codes & reduces readability. To overcome this problem Sun people introduced var-arg methods in 1.5 version. According to this we can declare a method which can take variable number of argument. such type of method are called var-arg method.

We can declare a var-arg method as follows

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

We can call this method by passing any no of values including zero no.

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

```
class Test {
  public static void m1(int... x) {
      System.out.println("Var-arg Method");
  }
  public static void main(String[] args) {
      m1();
      m1(10);
      m1(10,20);
      m1(10,20,30,40);
  }
}
```

Internally var-arg parameter will be converted into 1-D Array. Hence we can differentiate values by using index.

```
class Test {
    public static void main (String [] args) {
        sum ();
        sum (10,20);
        sum (10,20,30,40);
    }
    public static void sum(int... x) {
        int total=0;
        for(int x1 : x) {
            total = total + x1;
        }
        System.out.println("The sum is ": + total);
    }
}
```

Case 1 : Which of the following are valid var-arg declaration?

- ml(int[] x) ✗
- ml(int x[]) ✗
- ml(int [] x) ✗
- ml(int... x) ✓
- ml(int ...x) ✓
- ml(int...x) ✓
- ml(int x...) ✗ dots should be before.
- ml(int .x..) ✗
- ml(int. ..x) ✗ Three dot are bestfried. Can't be seperated

Case 2 : We can mix var arg parameter with normal parameter

```
ml (int x, int ... y)
ml (String s, double... y)
```

Case 3 : If we mix normal parameter with var-arg parameter Then var-arg parameter should be last parameter.

```
ml(double... d, string s) ✗
ml( char ch, string... s) ✓
```

Case 4: Inside var-arg method, we can take only one var-arg parameter.

m1(int... x , double... d) ✗

Case 5: Inside a class we can't declare var-arg method and corresponding 1-D array simultaneously. If we do so we get compile time error.

CE: ~~can~~ cannot declare both m1(int[]) & m1(int...) in test.

Case 6: In general var-arg method will get least priority. If no other method matches then var-arg will get chance. It is exactly same as default case in switch.

```
Class Test{
    public static void m1(int... x){
        Sopln("var-arg method");
    }
    public static void m1(int x){
        Sopln("General method");
    }
    psv main(String[] args){
        m1();      var-arg method
        m1(10,20); var-arg method
        m1(10);    General method
    }
}
```

Equivalence between var-arg parameter & one dimensional array

Case 1 : Wheseever ~~var-arg~~ one dimensional array present we can replace with var-arg parameter.

✓ m1(int[] x) ⇒ m1(int... x)

Even main, String[] args can be replaced String... args

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

m1(int[] x) ⟹ m1(int... x)

↓

Values passed

m1(new int[] {10})

m1 (new int[] {10,20})

m1 (new int[] {10,20,30})

m1()

m1(10,20)

m1 (10,20,30)

But reverse is not possible

m1(int... x) ⟹ m1 (int[] x)

m1() ✗

m1(10,20) ✗

Case 2 :

m1(int... x). We can call this method by passing a group of int values. And x will become one-dimensional array.

m1 (int[]... x), We can call this method by passing a group of 1-D int arrays. And x will become two-dimensional int array.

```
class Test {
    psv main(String[] args) {
        int[] a = {10,20,30};
        int[] b = {40,50,60};
        m1(a,b);
    }
    public static void m1(int[]... x) {
        for (int[] x1 : x) {
            Sopln(x1[0]);      => 10, 40
        }
    }
}
```

Main Method

Whether the class contain main method or not, these things won't be checked by compiler. At run time JVM is responsible to check these things. If JVM unable to find main method then we will get run time exception saying

RE: NoSuchMethodError: Main.

```
class Test {

}
```

Javac Test.Java ✓

Java Test => Exception in thread main Java.lang.NoSuchMethodError: main

At the run time JVM always searches for the main method with the following prototypes :-

Dt.
Page

public static void main (String[] args)

- public → To get called by JVM from anywhere
- static → At the time of execution of main method my class don't have any object. So without existing object also JVM has to call this method
- void → JVM calls main method. Main method won't return anything to Java
- main → This is the name which is configured inside JVM
- (String[] args) → command line arguments.

JVM {

psv main(String[] args) → JVM is configured with this prototype. So if we change here then we can use else.

}

main ⇒ xyz

change is JVM is must

The above syntax is very strict and if we change anything then Runtime error

RE : NoSuchMethodError : main .

Changes Allowed

(i) Order of modifiers isn't important. public static or static public

(ii) String array can be declared in any acceptable form.

main (String [] args) , main (String [] args) , main (String args[])

(iii) main(String [] sourav)

(iv) main (String... args) :- We can replace string array with var-arg parameter.

Spiral

- We can declare main method with the following modifiers final, synchronized, strictfp

```
class Test {
    static final synchronized strictfp public void main(String... sau)
    {
        System.out.println("Valid main method");
    }
}
```

Which of the following will give Compile time error & Run time error?

```
public static void main(String args)
public static void Main(String[] args)
public void main(String[] args)
public static int main(String[] args)
~~public~~ final synchronized strictfp public void main(String[] arg)
final synchronized strictfp public static void main(String [] args)
public static void main(String... args)
```

RE

No compile time error in any case

* Overloading of main method is possible. But JVM will call String[] args only. The other overloaded method we have to call explicitly like normal method call.

```
class Test {
    psv main(String[] args) {
        Sopln("String[]");
    }
    psv main(int[] args) {
        Sopln("int[]");
    }
}
```

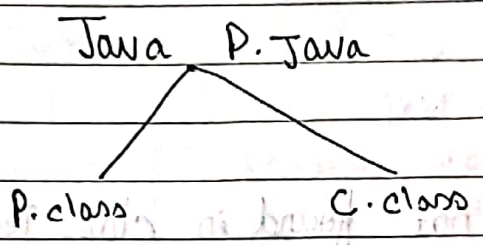
Java Test
↳ String []

* Inheritance concept applicable for main method. Hence while executing child class, if child doesn't contain main method then parent class main method will be executed.

```
class P {
    psv main (String [] args) {
        Sopln ("parent main");
    }
}
class C extends P {

}
```

P.Java



Java P
parent main

Java C
parent main

*
```
class P {
    psv main (String [] args) {
        Sopln ("parent main");
    }
}
class C extends P {
    psv main (String [] args) {
        Sopln ("child main");
    }
}
```

It is method hiding not overriding.

Java P
parent main

Java C
child main

It seems overriding concept applicable for main method but it isn't overriding but it is method hiding.

1.7 version enhancement wrt main method

- Until 1.6 version if the class doesn't contain main method then we will get Runtime exception
  NoSuchMethodError: main

But from 1.7 version we will get more ellaborated error information

```
class Test
{
}
```

1.6v :- Javac Test.java
Java Test
RE : NoSuchMethodError : main

1.7v :- Javac Test.java
Java Test
RE : Main method not found in class Test, please define the main method as
public static void main(String[] args)

- From 1.7 version onwards main method is mandatory to start the program execution. Hence even though class contain ~~main method~~ static block, it won't be executed if the class doesn't contain main method.

1.6v	1.7v
Java Test.java	Java Test.java
Java Test	Error : Main Metho .....
O/P :- Static Block	.....
RE : No such Method Error : Main	

```
class Test
{
static
{
System.out.println("static block");
}
}
```

Spiral

```
class Test {
    static {
        Sopln("static Block");
        System.exit(0); // Shutdown JVM
    }
}
```

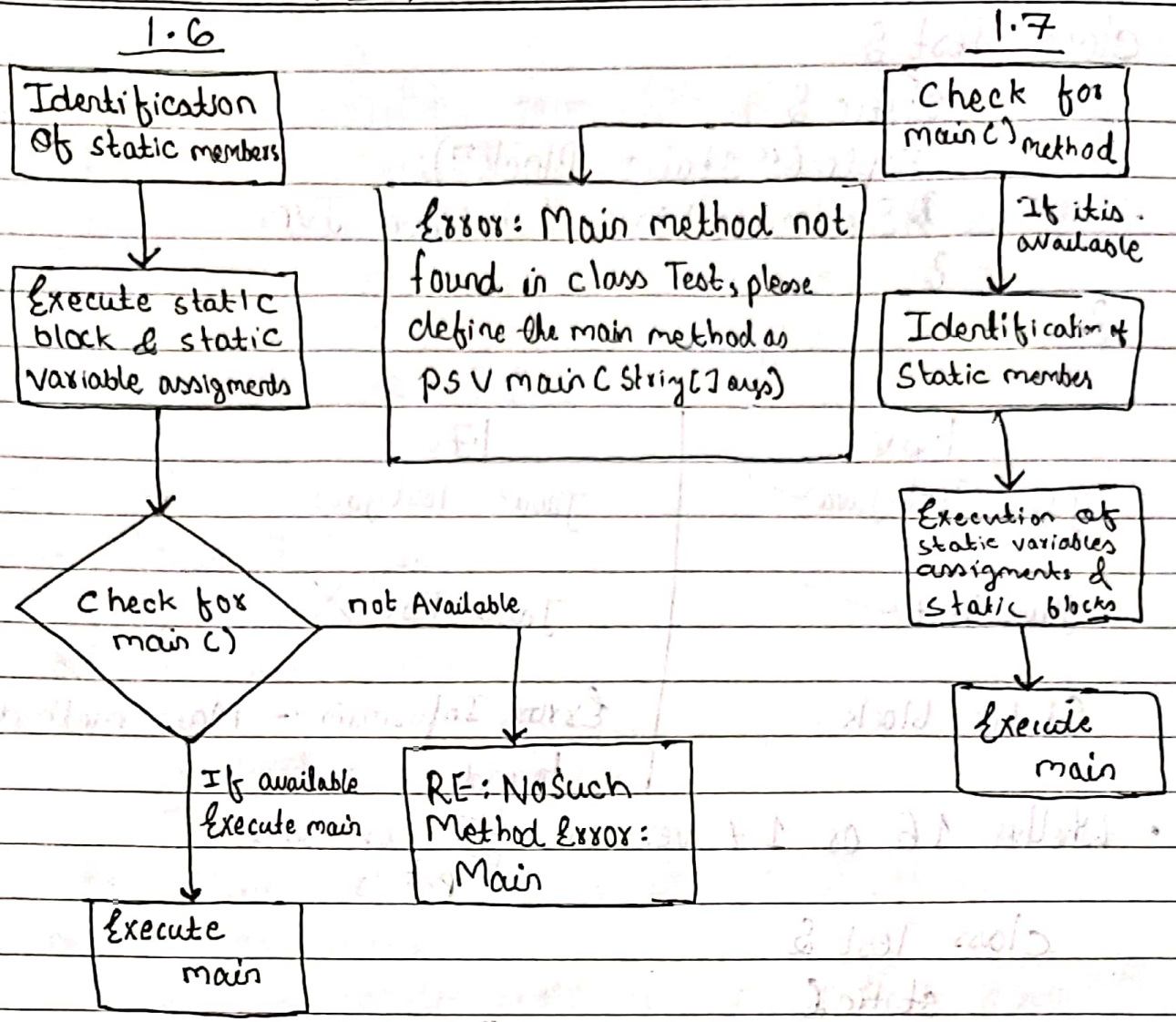
1.6v	1.7v
Javac Test.Java ✓	Javac Test.java
Java Test ✓	Java Test ✗
Static block.	Error Informain :- Main method not found;

- Whether 1.6 or 1.7 version, ~~flow~~ output is same.

```
class Test {
    static {
        Sopln("Static Block");
    }
    psv main(String[] args) {
        Sopln("main method");
    }
}
```

1.6v	1.7v
Javac Test.Java	"
Java Test	"
Static Block	"
main method	

Flow in 1.6 & 1.7



Yes, by using static block we can print statement in console without main till 1.6 version, after 1.6 ie 1.7 it is impossible to print statement in console.

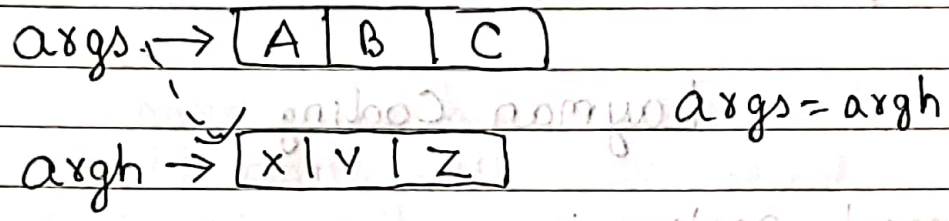
Command line Arguments

The arguments which are passing from command prompt are called command line arguments. With these command line arguments, JVM well create an Array and by passing that array as argument JVM will call main method.

* The main objective of command line argument is we can customize behaviour of the main method.

"2" → 2

Integer.parseInt(args[0]);



Java Test 10 20

```
class Test{
    p s v main(String[] args){
        Sopln(args[0] + args[1]); 1020
    }
}
```

* Within main methods command line arguments are available in string form.

* Usually space itself is the seperator between command line arguments, if our command line arguments contain space then we need to enclose command line argument within space.

Java Coding Standard

When we are writing Java code it is highly recommended to follow coding standard.

When we are writing any component, it name should reflect the functionality. The main approach of this is it improves readability and maintainability of code.

```
class A {
    public int m1(int x, int y) {
        return x+y;
    }
}
```

Layman Coding

```
import package.io
public class Calculator {
    public static int add (int number1, int number2) {
        return number1 + number2;
    }
}
```

Smart Coding

It is a normal utility class. Every Java class shouldn't contain main method. Only stand alone application to be executed from command prompt it requires main method.

Coding Standards for :-

Class

Usually class name are nouns. It should start with uppercase character. If it contains multiple words every innerword should starts with uppercase character.

23:00

Interfaces

Adjectives ⇒ Runnable, Serializable, Comparable

Usually interface names are adjective and should start with uppercase character. And if it contains multiple word, every innerword should starts with uppercase characted

Methods

Usually method names are either verbs or verbs-noun combination should starts with lowercase alphabet section, if it contains multiple words, every innerwords should start with uppercase character.

verbs ⇒ print(), sleep(), run(), eat(), start()

verbs-noun ⇒ getName(), setSalary()

Variables

Usually variable names are noun, starts with lower case alphabet symbol and if it contains multiple words, every inner word should start with upper case character. (Camel Case Convention)

name, age, MobileNumber

Constant

Usually constant names are noun should contain only uppercase character, if it contains multiple word, they are operated by _.

MAX_VALUE, MAX_PRIORITY, NORM_PRIORITY.

Spiral

Java Bean

A Java bean is a simple Java class with private properties and public getter and setter method.

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

Java listener

To register a listener, method name should be prefixed with add.

✓ public void addMyActionListener (myActionListener l)
✗ public void registerMy " " " "

To unregister a listener, method name should be prefixed with remove

✓ public void removeMyActionListener (myActionListener l)
✗ public void remove MyActionListener (ActionListener l)

same then valid