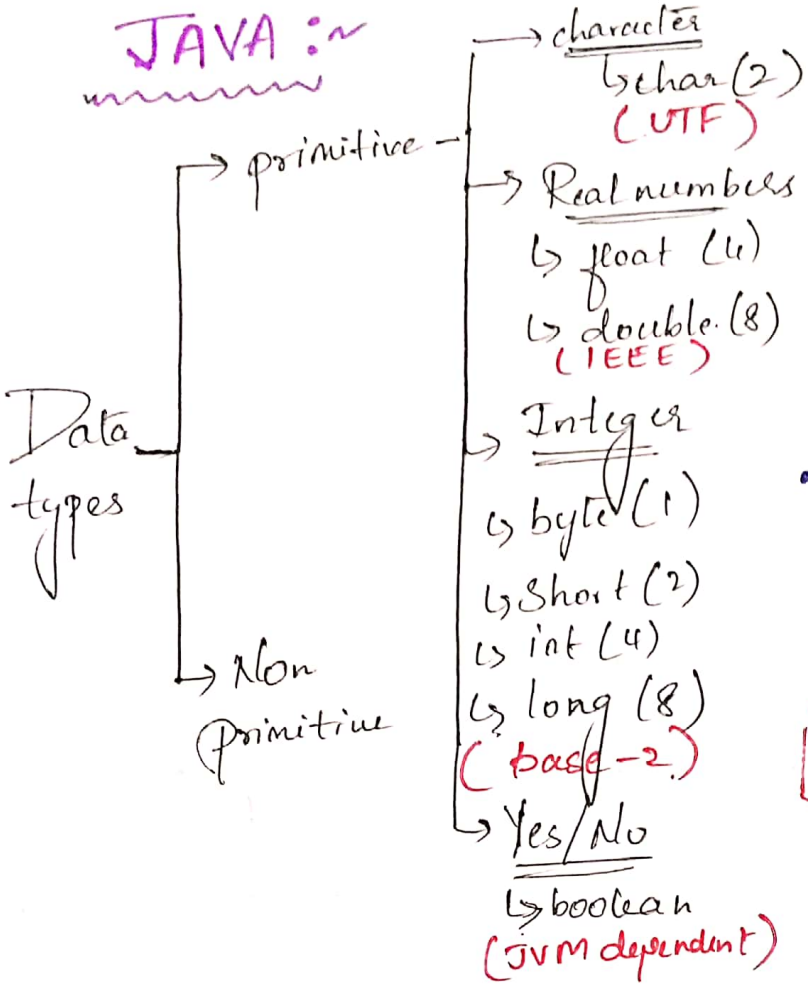
JAVA :-



defn: Datatypes in programming, is a classification that specifies the type & value the variable holds.

- And the type of mathematical, relational or logical operations that can be applied without causing an error.

range $\Rightarrow -2^{n-1}$ to $2^{n-1} - 1$

TYPE CASTING

defn assign a value from one primitive data type to another

(accurate) base-2 (not as much as IEEE)

byte(1) short(2) int(4) long(8)

UTF

char(2)

IEEE format (most efficient)

float(4) double(8)

Widening/implicit cast: (automatically) converting smaller data type to larger data type.

Narrowing/explicit cast: (manually) converting larger data type to smaller size.

Implicit :- byte → short → char → int → long → float → double

explicit :- double → float → long → int → char → short → byte

Identifiers

def: names that are used to identify a variable, method, class or label or interface.

- A-Z, a-z, 0-9, _, $ can be used
- can't use single underscore.
- cannot start with numerals. Ex 12abc X
- keywords cannot be used.

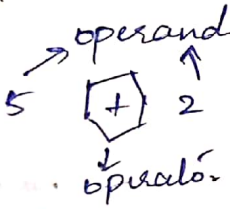
Ex: ___ , ab$$, $_ , $12 , $

invalid: _ , byte ; 123 ; int ; labe

Operators

def: spl symbols that performs operation over the operands.

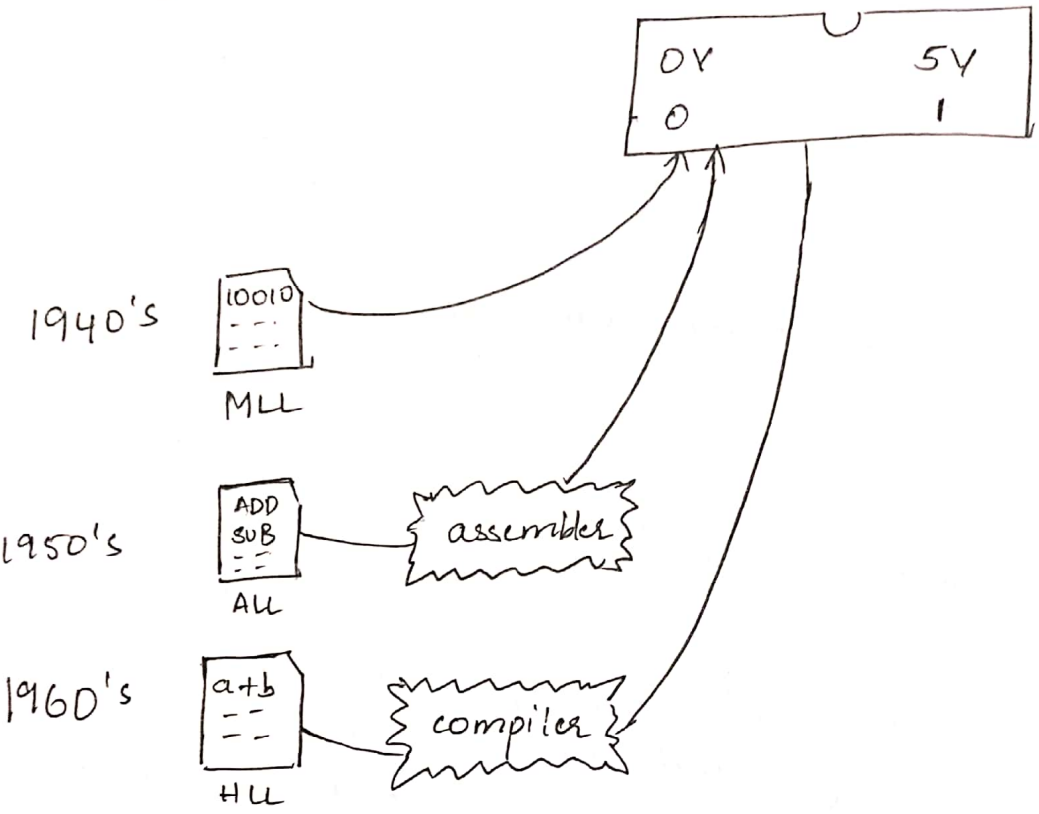
- operators dealing with → single operand (unary operator) ! ~
  → 2 operand (binary operator)
  → 3 operand. (ternary operator)



1.

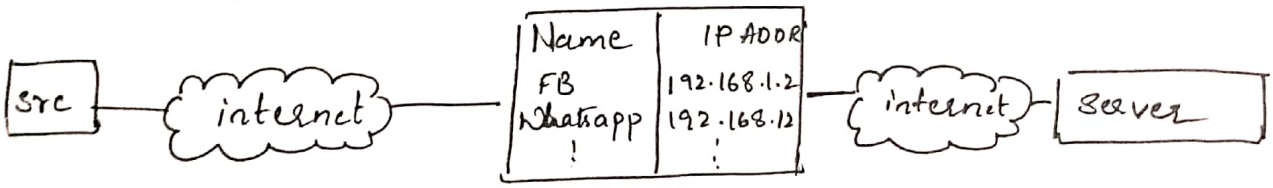
JAVA

History of languages

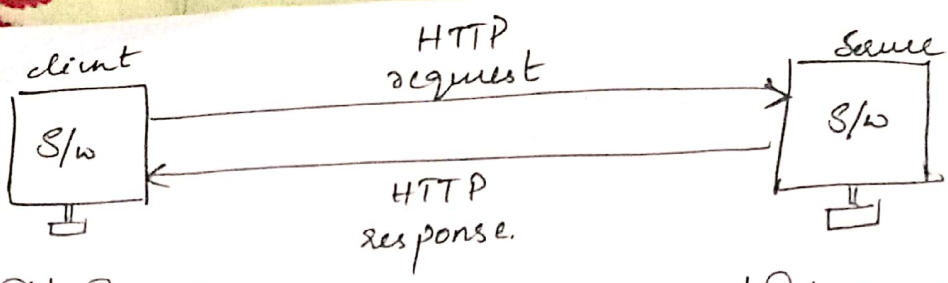


MLL	ALL	HLL
10001101	ADD	+
11011101	SUB	-
11001010	MUL	*
10110100	DIV	/

Data being exchanged



When facebook is searched IP address of facebook would be given to the server



Web Browser

chrome
Mozilla
Safari

Web server

Glass Fish
Tomcat
JBoss

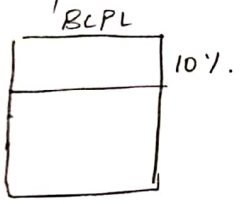
History of High level languages

1962

Martin Richards

BCPL

- occupied more memory

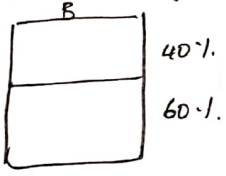


1965

Ken Thompson

B

- less memory
- unstructured



1972

Dennis Ritchie

C

- less memory
- structured
- no data security (not object oriented)

1982

Stroustrup

C++

- less memory
- object oriented
- not portable

1987

Russia-Afganistan

1989

1990

1991 [8 version-1995]

James Gosling

+ 10 members

Java.

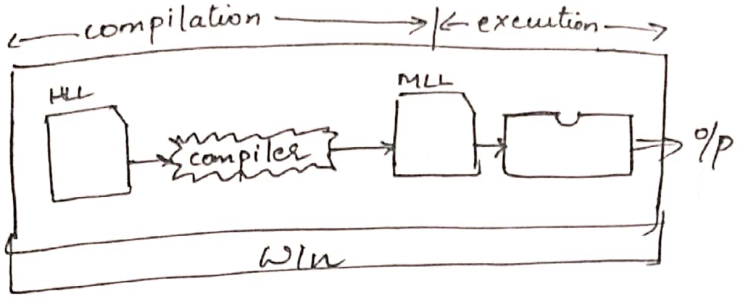
- less memory
- object oriented
- portable
- open source
- free

Platform:

- in case of h/w engineers → H/w + S/w
  microprocessor + OS
  4P
- in case of s/w engineers → ~~H/w~~ + S/w
  ~~4P~~ + OS

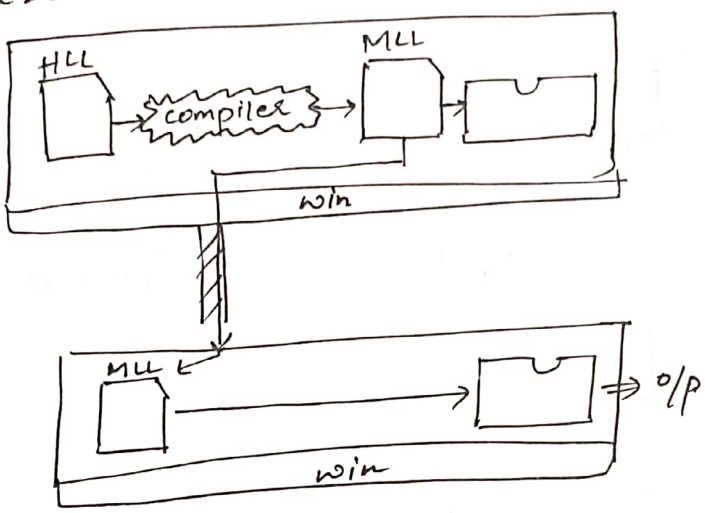
Portability

case 1:



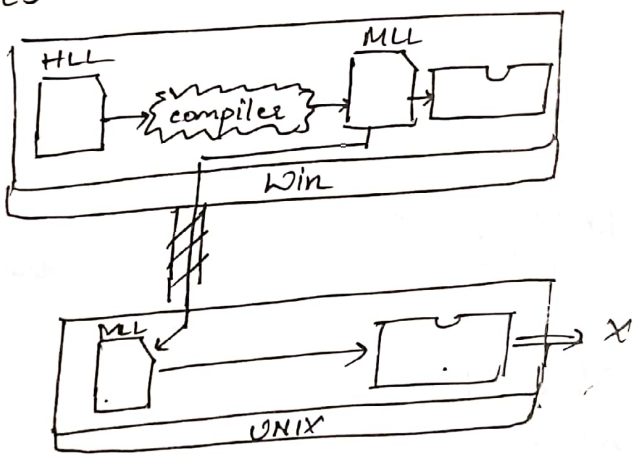
comp	exec
win	win ✓

case 2:



comp	exec
win	win ✓

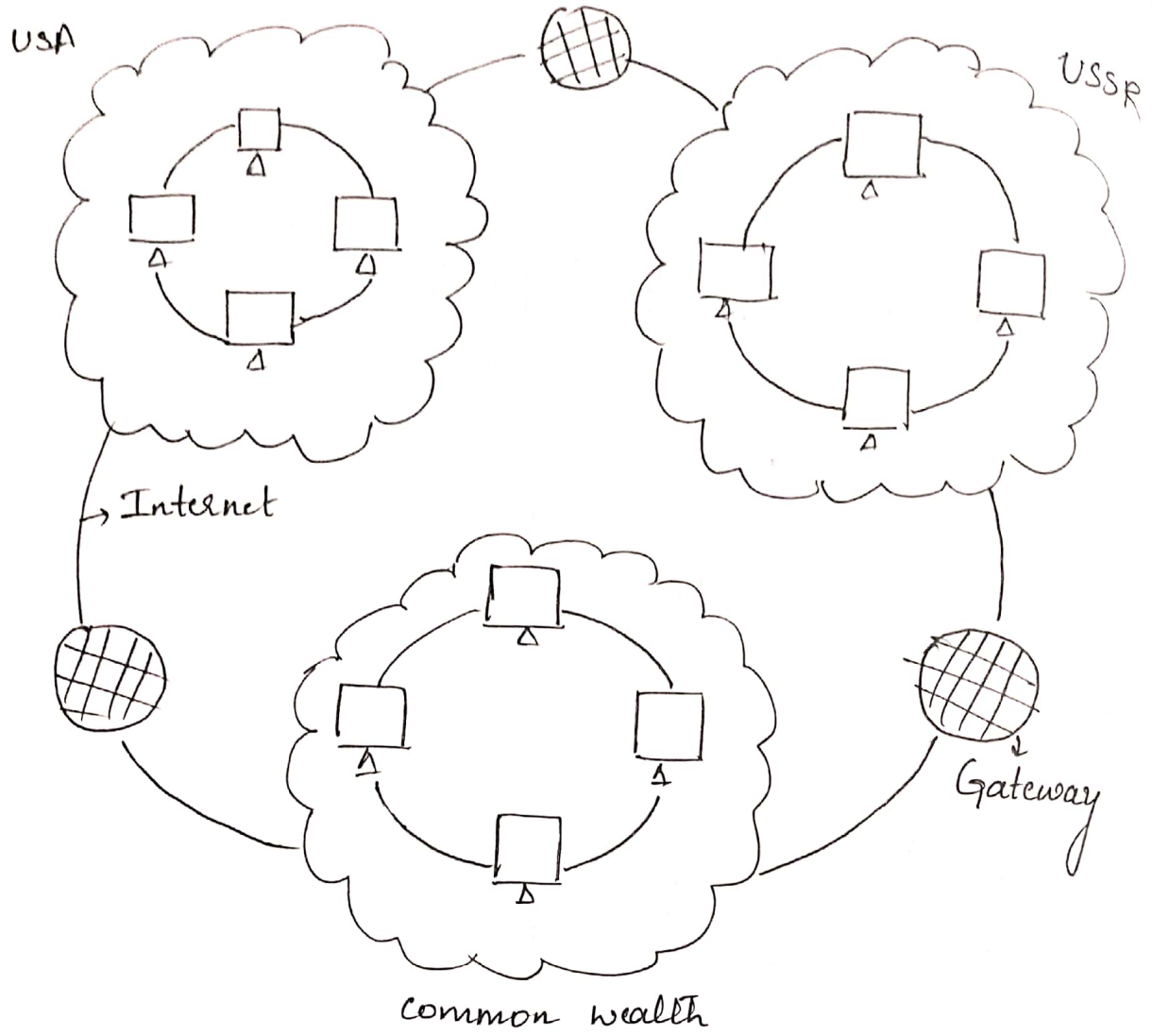
case 3



comp	exe
win	unix ✗

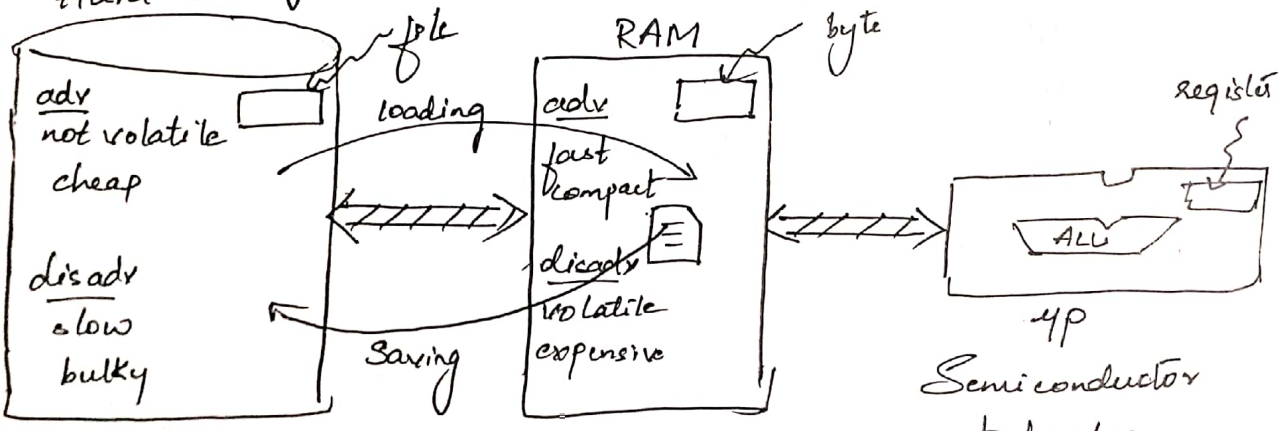
1914 → WW1
⋮
1918 → WW1 ended
1939 → WW2
⋮
1945 → WW2 ended
1945 → UNO
1950's & 1960's → peaceful
1970's superpower.

1980's → cold war
1982 → C++ released
1985 → Green team was formed.
1987 → Russia attacked Afganistan
⋮
1989 → cold war ended
1990 → Internet
1992 → B version was released
1994 → Java 1.0



Organisation of Memory :-

~~Semiconductor~~ Hard disk



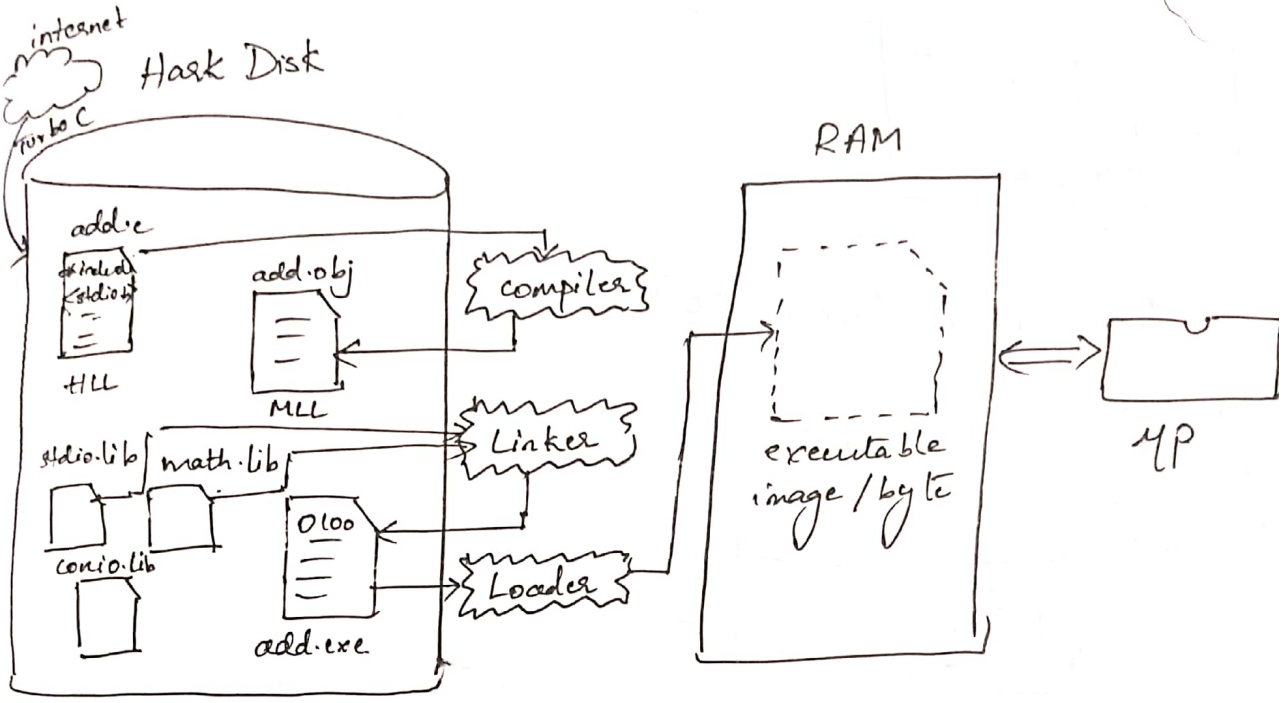
Magnetic technology (Secondary memory)

(main memory / primary memory)

Semiconductor technology

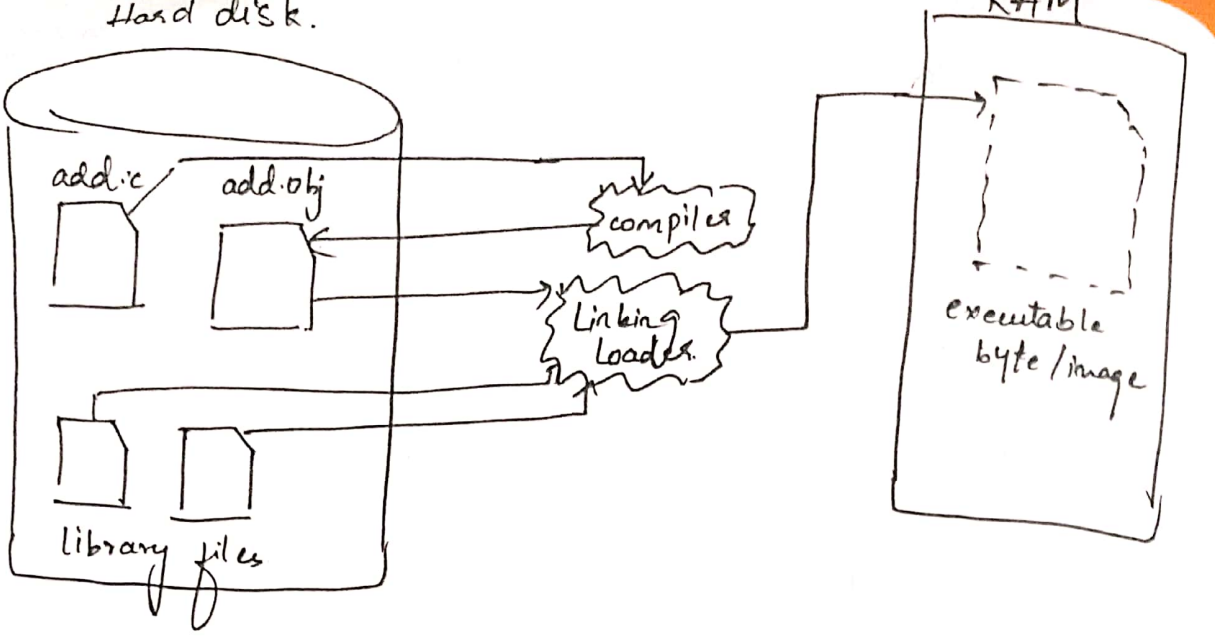
Saving is the process of copying the data RAM to hard disk

Loading is the process of taking a copy from hard disk to the RAM.

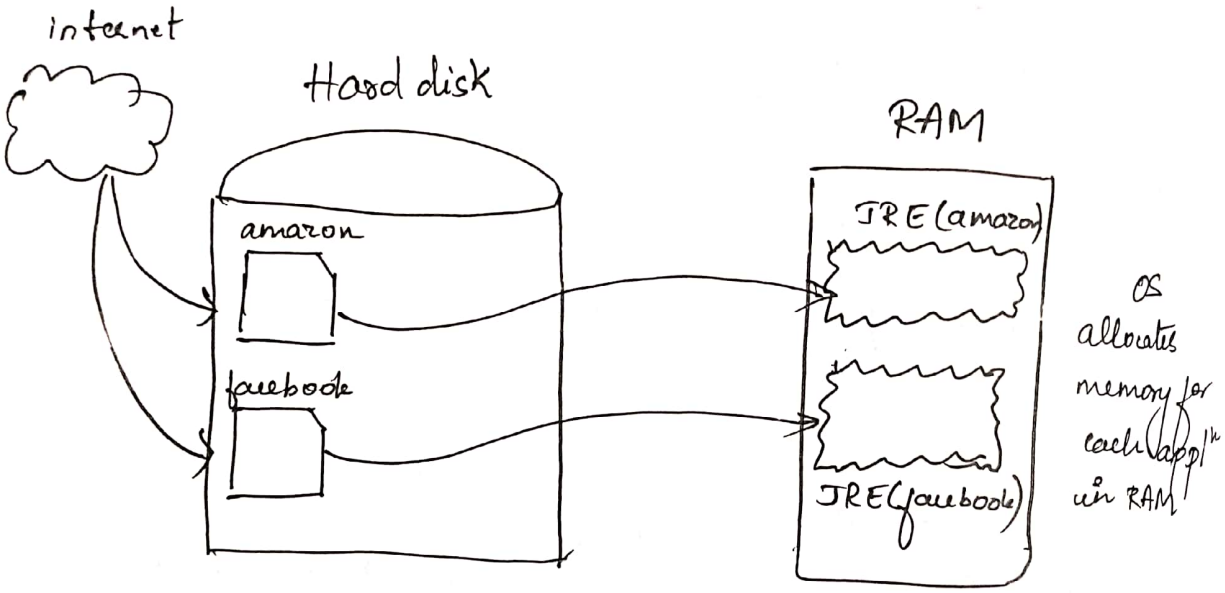


Note:

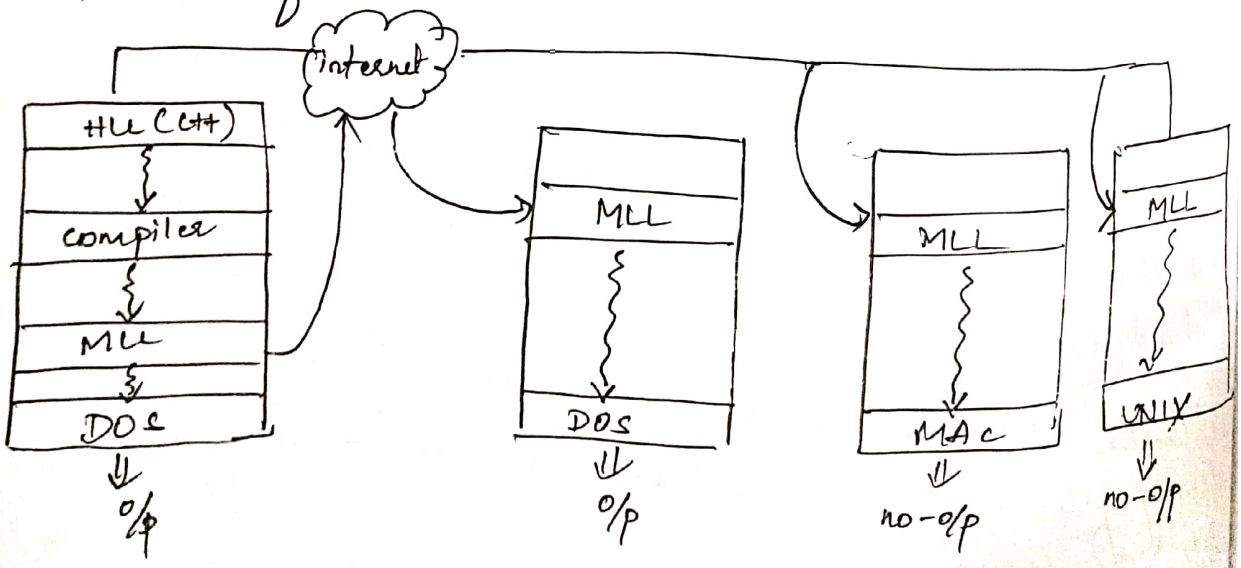
1. Assembler: convers it is a s/w wch converts assembly level lang into m/c level lang.
2. Compiler: it is a s/m s/w wch converts the HLL to MLL.
3. Linker: It is a s/m s/w wch takes object file & library files as the i/p & gives executable files as the output.
4. Loader: it is a s/m s/w wch loads the executable file or that takes a copy of the executable file from hard disk to RAM.
5. Linking Loader: It is a s/m s/w wch takes object file & library files as the i/p & loads the executable file to the RAM.



Java Runtime Envt (JRE)



Failure & C++



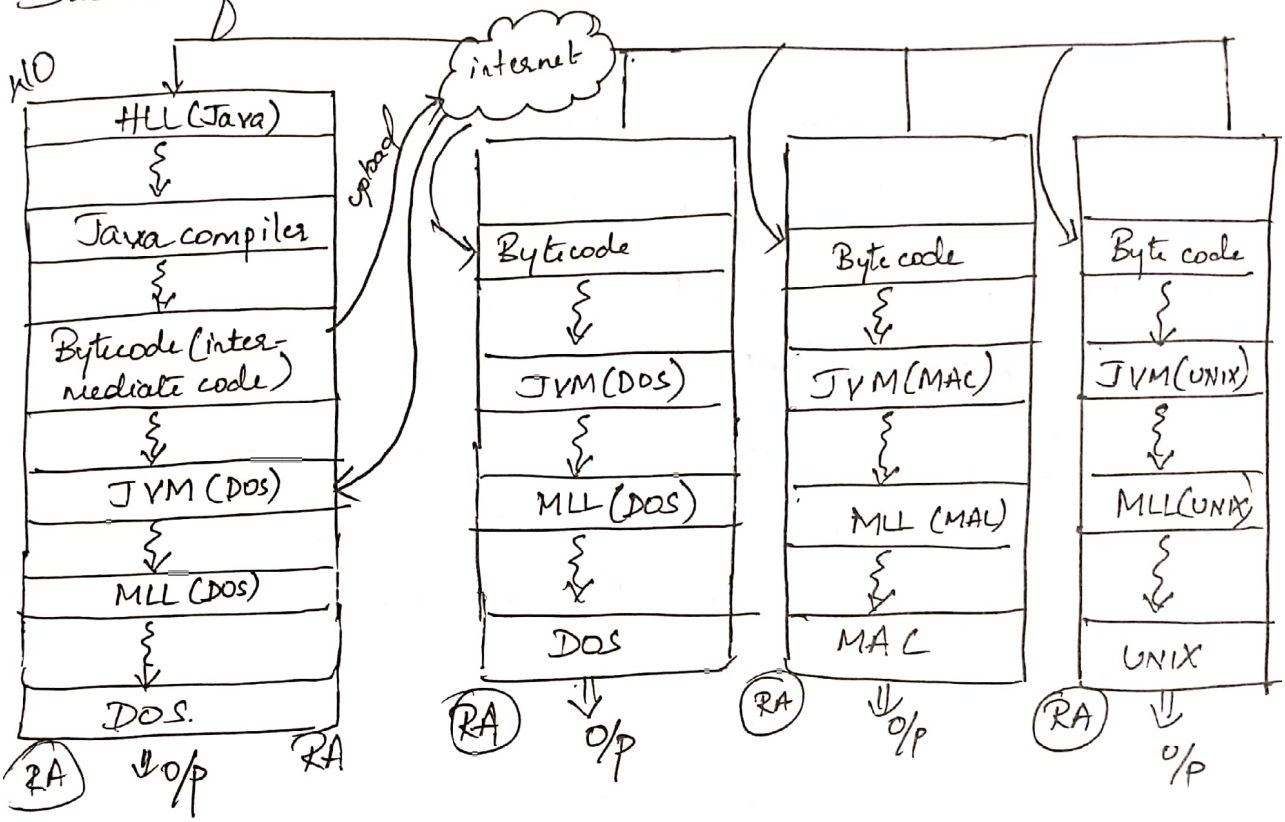
C++

case	compilation	execution	o/p
1	DOS	DOS	✓
2	DOS	DOS	✓
3	DOS	MAC	✗
4	DOS	UNIX	✗

Java

case	compilation	execution	o/p
1	DOS	DOS	✓
2	DOS	DOS	✓
3	DOS	MAC	✓
4	DOS	UNIX	✓

Success of Java.

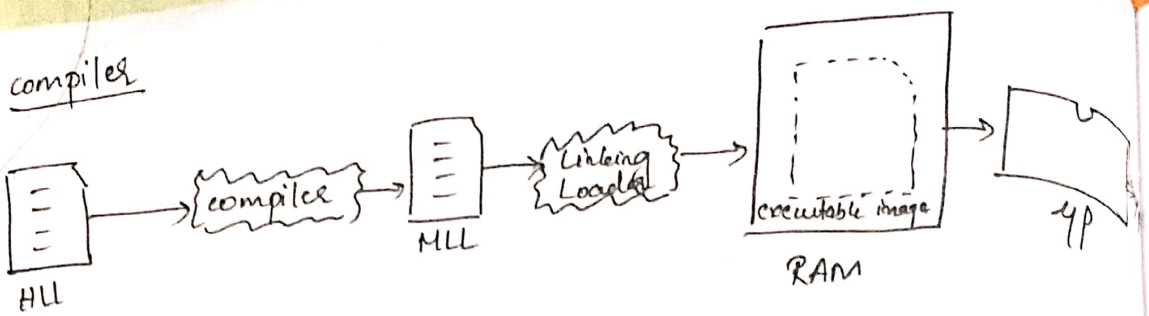


Note:-

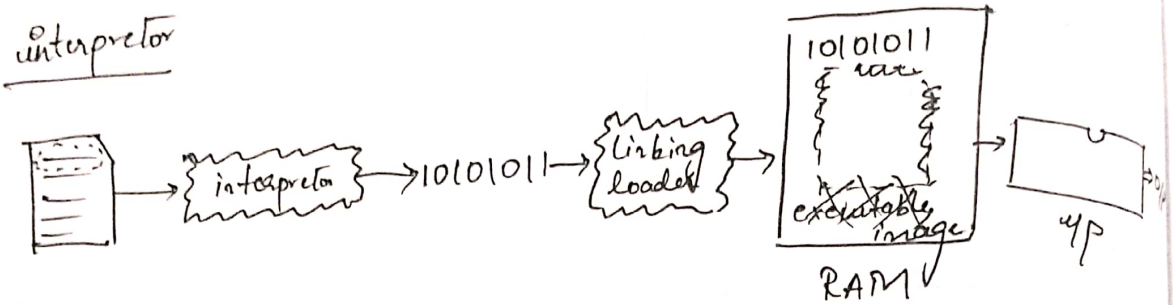
- Java achieved portability with the help of bytecode.
- Bytecode is neither in HLL nor in MLL hence called as intermediate code.
- JVM is a s/w which accepts bytecode as the i/p & produces MLL as the o/p.
- Files | Platform Independent

Files	Platform Independent
Src file	✓
Bytecode	✓
JVM	✗
MLL	✗

compiler



interpretor



Operating S/m	programming lang's
windows (microsoft)	C (Turbo C Crop)
UNIX (AT & T)	C++ (Turbo C Crop)
Macintosh (Apple)	Java (Oracle crop)

Note: The only expectation the OS has from the programming lang is the programming lang shd contain a main method. The OS hand overs the control to the main method. Once the prog is executed completely the control has to be handed back to the OS.

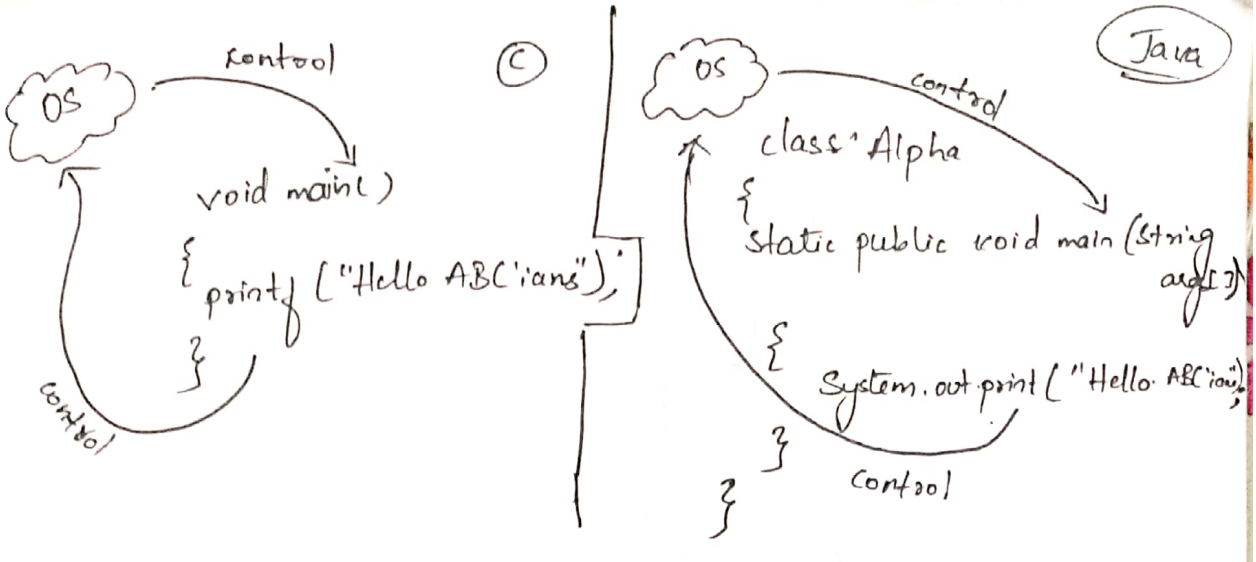
- function, method, routine, sub routine, procedure, module all mean the same with sleight differences.

- output name (input)
  {
  = activity
  }

→

datatype name (parameters)
{
= body (set of instructions)
}

- name and body are mandatory, i/p & o/p are not mandatory.



Note: • If main method is not given as static [which makes OS s/m accessible) then it pops an error specifying "Main method is not static in class Alpha".

• If main method is not made public (OS s/m visible) then it pops an error "main method not found in class Alpha".

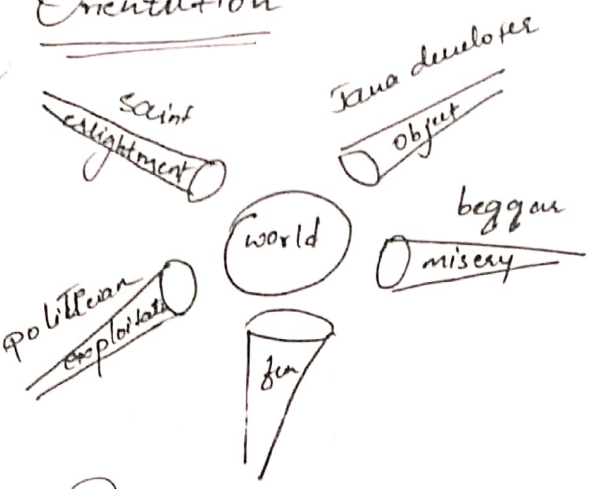
• static, public → access modifiers can be interchanged & written. (also called key words/reserved words)

• Alternatives ways of writing signature of the main method:
static public void main (String args[])
public static void main (String args[])
public static void main (String[] args)
public static void main (String... args)

• Public is a access modifier in java which will make the method or a variable accessible outside the class.

• Static is a access modifier in java. We have to make main method as static so that we dont have to create the object of that particular class to call the main method.

Object Orientation



Rules of Object Orientation

- Ever. This world is collection of objects.
- There is no object wch is useless. Every object is an useful object.
- No object is in isolation. Every object will be in constant interaction with another object.
- Every object belongs to a type and the type doesn't exist in reality. It is the objects that exist in reality. Technically we call the type as class (class is a blueprint of an object)
- Every object has few properties and thus few activities. Properties are defined using data types and activities are defined using methods.

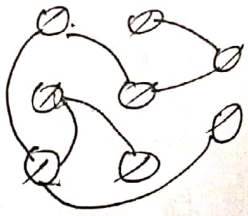
BookMyShow

customer, theatre, counter, money, seats, ticket, screen, movie.

```
class Customer {
    ==
}
class Theatre {
    ==
}
class Counter {
    ==
}
```

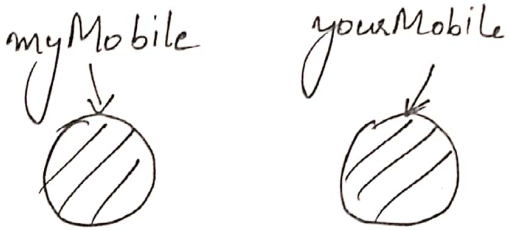
```
class Money
{
    ==
}
class Seats
{
    ==
}
class Ticket
{
    ==
}
```

```
class Screen
{
    ==
}
class Movie
{
    ==
}
```



```
class Mobile
{
  int camera;
  int RAM;
  String sim;
  void call()
  {=
  }
  void message()
  {=
  }
  void watchMovie()
{=
}
```

Mobile
- has → datatypes: camera, RAM, sim
- does → call, message, watch movie } methods

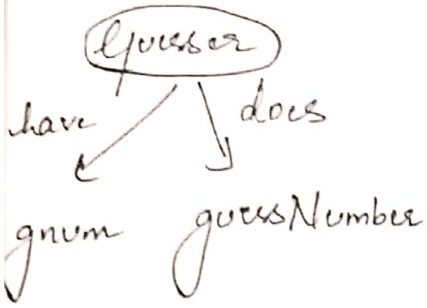


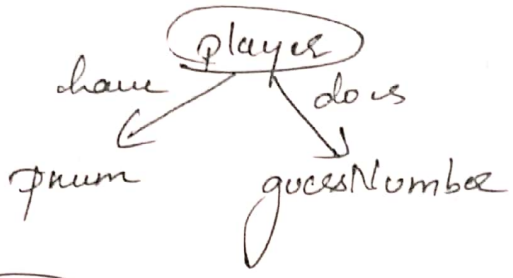

```
Mobile myMobile = new Mobile();     // giving name to an object
Mobile yourMobile = new Mobile();   // in order to create an object new is used
                                    } creating new objects
myMobile.call();
yourMobile.watchMovie();            } objects calling the method
```

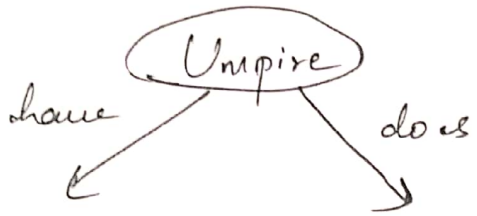
```
class Car
{
String engine
String color
 int wheels
 int seats

 void start
 {=
 }
 void stop
 {=
 }
 void playMusic
 {=
 }
 void move()
 {=
 }
```

Car
- has → engine, color, wheels, seats
- does → start, stop, playMusic, move

Guesser & player game :







gnum
playerOneNumber
playerTwoNumber
playerThreeNumber

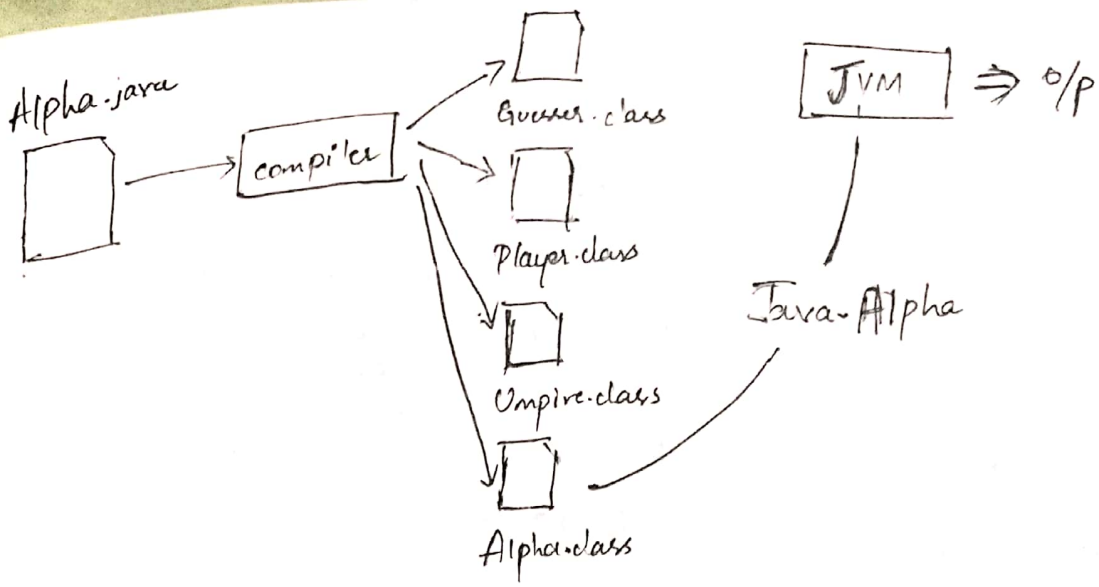
collectNumberFromGuesser
collectNumberfromPlayer
compare

```
import java.util.Scanner;
class Guesser
{
   int gnum;
   int guessNumber()
   {
    System.out.println("Guesser, guess a number");
    Scanner scan = new Scanner(System.in);
    gnum = scan.nextInt();
    return gnum;
   }
}
class Player
{
 int pnum;
 int guessNumber()
 {
   System.out.print("Player, guess a number");
   Scanner scan = new Scanner(System.in);
```

```
        pnum = scan.nextInt();
        return pnum;
    }
}

class Umpire
{
    int guessedNumber;
    int playerOneNum;
    int playerTwoNum;
    int playerThreeNum;

    void collectNumberFromGuesser()  // new obj of the Guesser class
    {
        Guesser dharmraj = new Guesser();
        guessedNumber = dharmraj.guessNumber();
        datatype
    }
    void collectNumberFromPlayer()
    {
        Player name1 = new Player();
        Player name2 = new Player();
        Player name3 = new Player();
        playerOneNum = name1.guessNumber();
        playerTwoNum = name2.guessNumber();
        playerThreeNum = name3.guessNumber();
    }
```

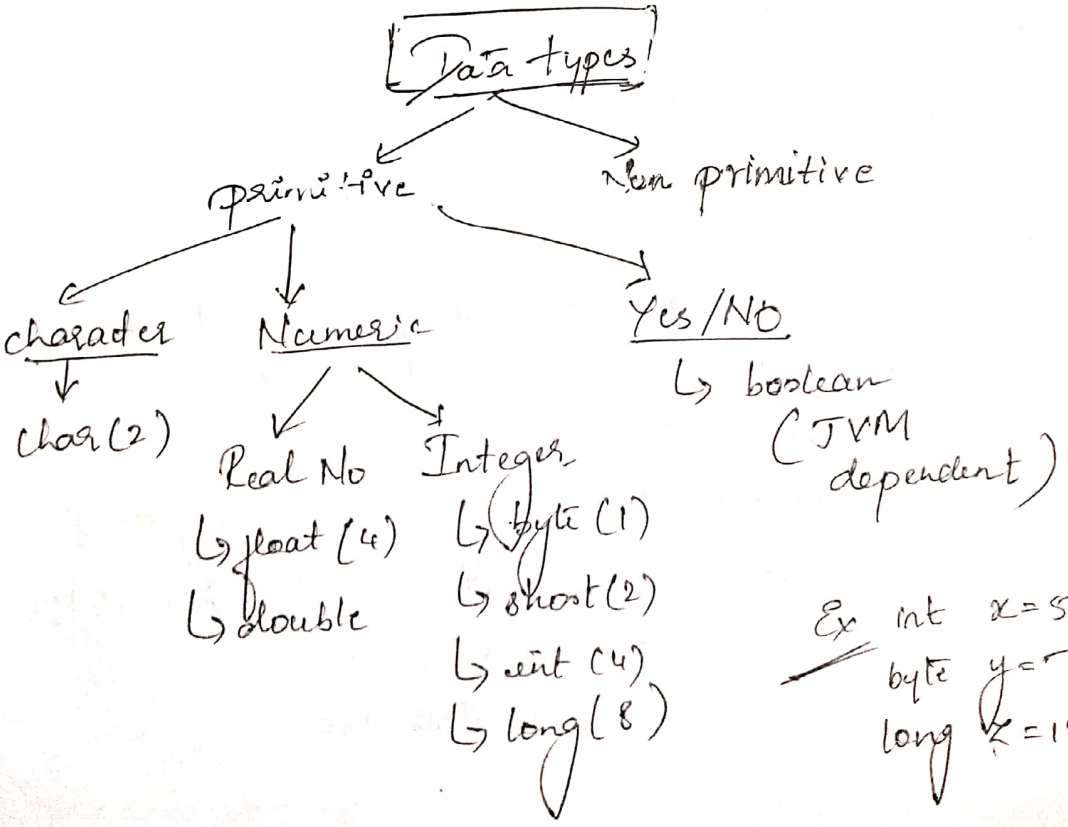
```
void compare()
{
    if (guesserNumber == playerOneNum)
    {
        System.out.print(" Player One wins ");
    }
    else if (guesserNumber == playerTwoNum)
    {
        System.out.print(" Player Two wins ");
    }
    else if (guesserNumber == playerThreeNum)
    {
        System.out.print(" Player Three wins ");
    }
    else
        System.out.print(" Better luck next time");
}
}
class Alpha{
    Public static void main(String[] args)
        Umpire somanna = new Umpire();
        somanna.collectNumberFromGuesser();
        somanna.collectNumberfromPlayers();
        somanna.compare();
    }
```

Note: camel conversion — howAreYou | Pascal Convention — HowAreYou



Architecture of JVM.

Real Life	datatypes	Format
character	char	UTF
number	byte, short, int, long	base 2
Real numbers	float, double	IEEE
Yes/No	boolean	JVM dependent
images, audio, video	} inbuilt class	



Ex int x=5; [4 bytes are allocated]
byte y=7;
long z=120;

byte(8)

-2^{8-1} to $2^{8-1}-1$

-2^{7} to $2^{7}-1$

-128 to 127

-ve → 1

+ve → 0

- for long number use l/L at the end
Ex: 922334567891OL

short (16 bits)

-2^{16-1} to $2^{16-1}-1$

-32768 to 32,767

int (32)

-2^{31} to $+2^{31}-1$

-2147483648 to

- - - -647

long (64)

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

-9223372036854

775808 to

- - - - - - - 807

float ± 3.40282347E+38F

- for float no suffix it will 'd'

double ± 1.79769313486

231570E +308

```
class DataTypes {
    public static void main(String[] args) {
        byte myData = 127;
        System.out.println("myData = " + myData);
    }
}
```

Note: When values beyond the range of that particular datatype is stored it would provide an error "possible lossy conversion".

```
class DataTypes {
    public static void main(String[] args) {
        int myData = 130;
        int yourData = myData;
        System.out.println("myData = " + myData);
        System.out.println("yourData = " + yourData);
    }
}
```

o/p myData = 130

yourData = 130

```
class DataTypes {
    public static void main(String[] args) {
        int myData = 130;
        byte yourData = myData;
        System.out.println("myData = " + myData);
        System.out.println("yourData = " + yourData);
    }
}
```

o/p: Error: possible lossy conversion from int to byte
byte yourData = myData;

sol:
```
int myData = 126;
byte yourData = (byte)myData; // explicit type.
```

∴ o/p :- myData = 126
yourData = 126

Note

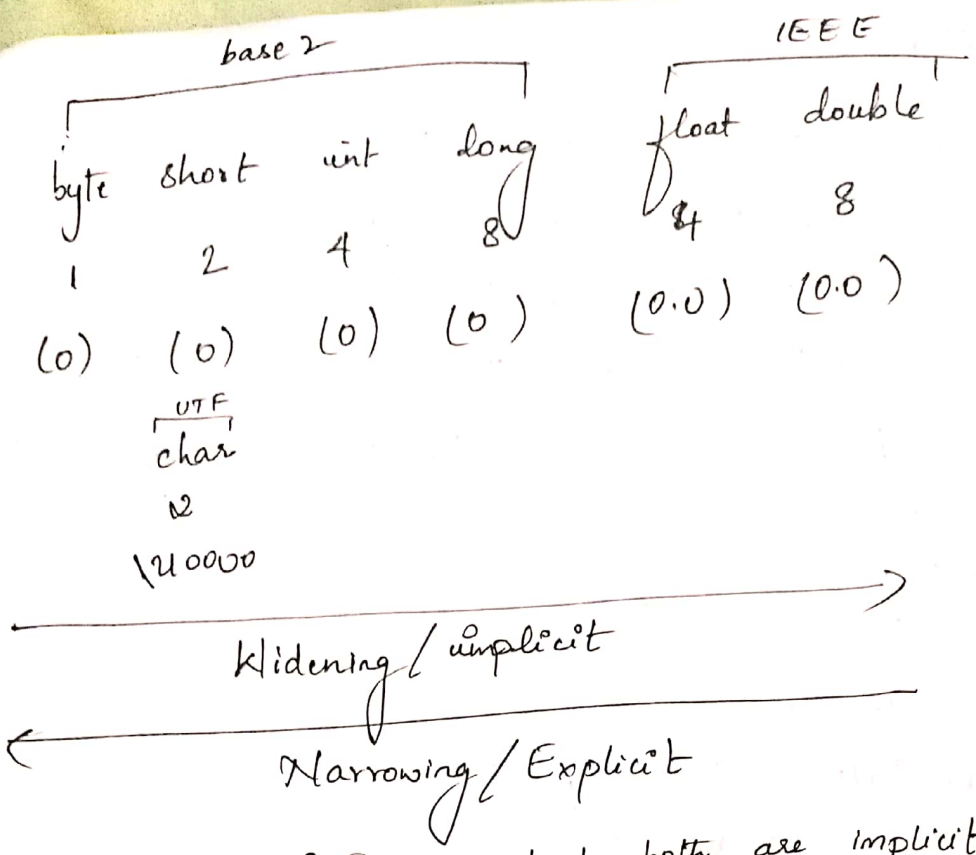
-128, -127, -126, ----- 0, 1, 2, 3 ..... 127 | 128, 129, 130 ....

```
class demo
{
    public static void main(String[] args) {
        byte b;
        byte a;
        a = 20;
        b = 30;
        byte c = a + b;
        System.out.println(c);
    }
}
```

o/p: Type mismatch: cannot convert from int to byte
when '+' operation takes place byte is converted into int

So!! int c = a + b

o/p : 50



- Short to char & char to short both are implicit
- IEEE format is the most efficient way of storing the data
- In IEEE format the values are not accurate
- Base 2 format is not that efficient compared to IEEE format
- Values are accurate in case of base 2 format.

Identifiers :-

Identifiers are used to identify a variable, method, class & label, or interface. • Identifiers are case sensitive Ex: Byte

- The characters that can be used in identifiers are A-Z, a-z, 0-9, _, $.
- We cant start a identifier with a numeric character Ex: 12abc
- We can't name an identifier as single underscore '_'
- We can't use keywords as identifiers.

valid : ___ $1$2$3 _$ $ ab$$ Byte

invalid : _ , byte } keywords 123 , int } keywords !_2_3

octal

1 - 1	9 - 11
2 - 2	10 - 12
3 - 3	11 - 13
4 - 4	12 - 14
5 - 5	13 - 15
6 - 6	14 - 16
7 - 7	15 - 17
8 - 10	16 - 20

Prefix

0b/0B	binary	- 0, 1
0 (zero)	octal	- 0, 1, 2, 3, 4, 5, 6, 7
as it is	decimal	- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
0X/0x	Hexa-decimal	- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 A, B, C, D, E, F

Formula : MAX(int, operand1, operand2)

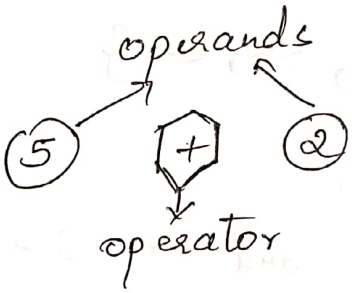
Ex int, byte, byte

The result would be in 'int'.

Operators :- are the spl symbols which does operations over the operands.

~~to Arthmetic operators :-~~

- operators dealing with single operand is classified as unary operator Ex : !, ~
- operators dealing with 2 operands is considered as binary operators.
- operators dealing with 3 operands is considered as ternary operators.



1. Arthmetic operator :

{ +, -, /, * }

- 5/2 = 2
- 5/0 = Arthmetic exception
- 5/0.0 = Infinite
- -5/0.0 = -Infinite
- 0.0/0.0 = NaN [Not a Number].
- 5.0/0 (double / int) = double
- & 5/0.0 = Infinite

2. Relational operators:

{ >, >=, <, <=, ==, !=, instanceof }

5>2 — true
5>=2 — true
5<2 — false
5<=2 — false
5<=5 — true
5==2 — false

5!=2 — true
5!=5 — false

3 Logical operators:

{ ||, &&, ! }

||

op1	op2	result
true	X	true
true		true
false	true	true
false	false	false

&&

op1	op2	result
true	true	true
true	false	false
false	X	false
false		false

!

op	result
true	false
false	true

Note: The operands for logical operators are always boolean values & results are also boolean values.

4. Bitwise operators

{ ~, |, &, ^, >>, << }

$\sim n = -(n+1)$

Ex: ~5 = -6 and ~-3 = 2

Ex: x = 5 & 3

00000101
00000011
00000001 ⇒ (1)

|

		res
0	0	0
0	1	1
1	0	1
1	1	1

(OR)

&

		res
0	0	0
0	1	0
1	0	0
1	1	1

(AND)

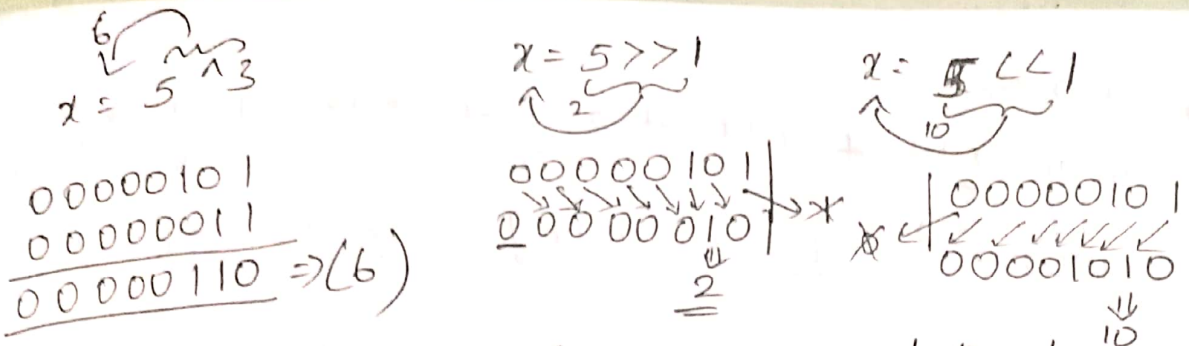
^

		res
0	0	0
0	1	1
1	0	1
1	1	0

(XOR)

~~x = 5|~~
x = 5 | 3

00000101
00000011
00000111 = (7)



Note:- Bitwise operation happens only for byte, short, int, long, char datatypes. For rest of the datatypes the compiler would give an error.

5. Assignment operators:

{=, +=, -=, *=, /=, %=, |=, &=, ^=, >>=, <<=}

Ex: x = x + y (must be same) → x += y

x = x ^ y → x ^= y

x = y - x (not same)

6. Ternary operator: { ? : }

op1 ? op2 : op3 (true / false)

The first operand must always be a boolean value.

Ex:
- int x = true ? 5 : 2
- int y = false ? 5 : 2
- int x = (5>2) ? 5 : 2 (true)
- int y = (5<2) ? 5 : 2 (false)

7. increment, decrement operator: {++, --}

Ex
- int x=5; x++; SOP(x); => 6
- int x=5; ++x; SOP(x); => 6
- int x=5; int y=x++; SOP(x); => 6 SOP(y); => 5 (x 6, y 5)
- int x=5; int y=++x; SOP(x); => 6 SOP(y); => 6 (x 6, y 6)

- int x=5;

int y = x++ + ++x + x++ + x++ + ++x;
(5) 5 6 7 7 8 8 9 10

x 5 6 x 6 7 x 7 8 x 8 9 x 9 10

- int x=5;

int y = x++ + --x + x++ + x-- + --x
(5) (5) (5) (6) (4)
(25)

x 5 6 5 6 5 5 4

y 25

For right shift = $\frac{N}{2^{\text{No of bit}}}$

For left shift = $N \cdot 2^{\text{No of bit}}$

int x=0;

int y = x++ + ++x + --x;
(0) (2) (1)

sop(x)

sop(y)

x = 0 1 2 1

y = 3

int x=3;

int y = x++; 3

int a = ++x + --x + --y + y-- - ++x;
5 4 2 2 5

sop(x);//5

sop(y);//1

sop(a);//8

x = 3 4 3 4

y = 3 2 1

a = 7

x = 3 4 5 4 5

y = 3 2 1

a = 8

```
int x=0;
int y=5-1.2;
int a = x++ - --x + y-- + ++y - ++x + y--;
SOP(x); // 1
SOP(y); // 0
SOP(a); // 1
```

(0) (1) (1) (1) (1) (1)

x = 0 1 0 1

y = x 0 x 0

a = 1

Control Statements : is used to control execution of the prg

1. if statement

case 1:

```
if (condition) {
    body of if
}
else {
    body of else
}
```

true / false

if is used to specify a block of code to be executed if a specified condition is true.
else is used to specify a block of code to be executed if the same condition is false.

case 3:

```
if (condition-1) {
}
else if (condition-2) {
}
else if (condition-3) {
}
```

else if - to specify a new condition to test, if the first condition is false

```
import java.util.*;
class Result {
    public static void main (String[] args) {
        SOP ("Enter your marks");
        Scanner scan = new Scanner (System.in);
        float marks = scan.nextFloat();
```

```
int x = 0;
int y = 5*1.2;
int a = x++ - --x + y-- + ++y - ++x + y--;
        (0)  (1)  (1)  (1)  (1)  (1)
SOP(x); // 1
SOP(y); // 0
SOP(a); // 1
```

x = | 0 1 0 1 |

y = | 1 0 1 0 |

a = | 1 |

Control Statements : is used to control execution of the prg

1. if statement

case 1:

```
if (condition) {
    = body of if
}
```

(condition → true; false → skip)

case 2:

```
if (condition) {
    = body of if
}
else {
    = body of else
}
```

(condition → true; false → else)

case 3:

```
if (condition-1) {
}
else if (condition-2) {
}
else if (condition-3) {
}
```

else if - to specify a new condition to test, if the first condition is false

```
import java.util.*;
class Result {
    public static void main (String[] args) {
        SOP ("Enter your marks");
        Scanner scan = new Scanner (System.in);
        float marks = scan.nextFloat();
```

```
if (marks>=70) {
    SOP ("Distinction");
}
else if (marks >=60) {
    SOP (" First class");
}
else if (marks >=50) {
    SOP (" Second class");
}
else if (marks >=35) {
    SOP (" lucky person");
}
else
    SOP (" Study well");
}
}
```

```
import java.util.*;
class Chocolates {
  public static void main (String[] args) {
    SOP (" Enter number of friends ");
    Scanner scan = new Scanner (System.in);
    int n = nextInt();
    SOP (" Enter no of chocolates in a
        packet ");
    int m = nextInt();
    if ( m % n == 0 ) {
      SOP (" Yes");
    }
```

```
        else{
            SOP("NO");
        }
    }
}
```

l = 24/2 = 12

r = 50/2 = 25

switch st: • is a multiway branch stat
- provides an easy way to dispatch execution to different parts of the code based on the value of expression (byte, short, char data types)
- (OR) select one of the many code blocks to be executed

Switch statement :-

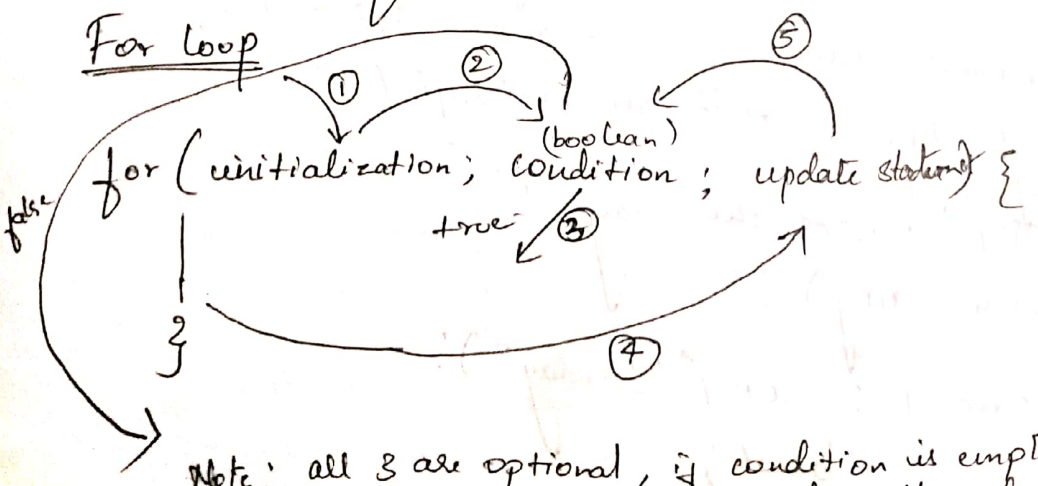
```
class test    switch(expression){
                case exp1 :  body-1
                    break;
                case exp2 :  body-2
                    break;
                case exp3 :  body-3
                    break;
                default : body4
```

```
class Week {
    public static void main (String args[]) {
        Scanner scan = new Scanner (System.in);
        byte day = next scan.nextByte();
        SOP("Enter the day");
        switch(day) {
            case 1 : SOP("Sunday");
                break;
```

```
case 2 : SOP("Monday");
         break;
case 3 : SOP("Tuesday");
         break;
case 4 : SOP("Wednesday");
         break;
case 5 : SOP("Thursday");
         break;
case 6 : SOP("Friday");
         break;
case 7 : SOP("Saturday");
         break;
default : SOP("Get lost");
}
}
}
```

Note:
- The expression can be of integer type data (int, byte, short, long), character, String
- break & default are optional.

For loop



Note: all 3 are optional, if condition is empty its default value is "true"

- for (; ;) { — defn: executes part of the Prog repeatedly based on basis of given boolean condition

optional

}

- for (, , , ; ; , , , ,) {

any no of initialization — any no of update statements

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

SOP("*");

}

- for (int i=-1; i>-4; i--) {

SOP("*");

}

- for (int i=-6; i<0; i=i+2) {

}

- for (int i=0; i<5; i++) {

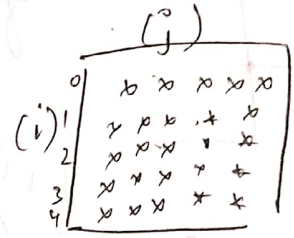
for (int j=0; j<5; j++) {

SOP('*');

}

SOPln();

}



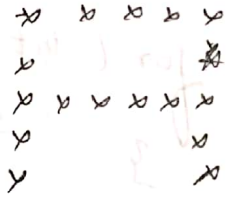
```
import java.util.Scanner;
public class Alpha {
    public static void main(String args[]) {
        System.out.println("Enter n: ");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
```

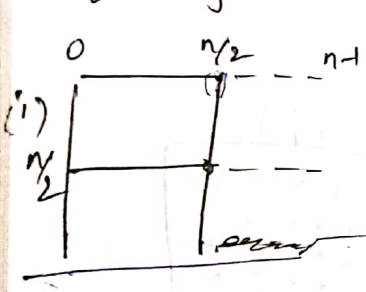
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    if(i==0 || i==n-1 || j==0 || j==n-1){
      SOP("*");
    }
    else{
      SOP(" ");
    }
  }
  System.out.println();
}
```

O/P

```
Enter n: 5
* * * * *
*       *
*       *
*       *
* * * * *
```

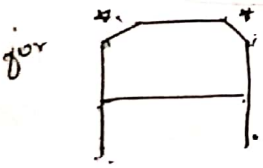
```
if(i==0 || i==n-1 || j==0 || j==n/2)
```

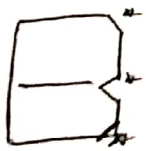




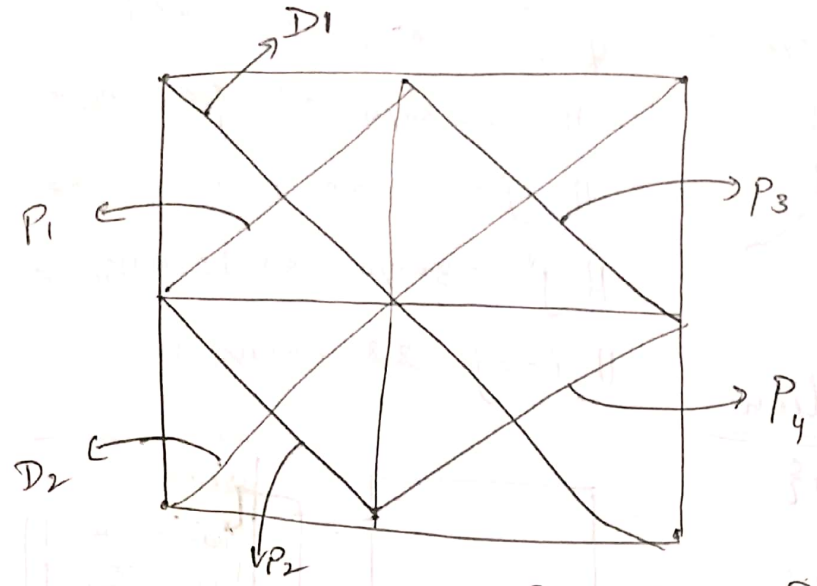

```
if(j==0 || j==n/2 || (i==0 && j<=n/2)
    || (i==n/2 && j<=n/2))
```

```
if((i==0 && j!=0 && j<n/2) ||
  (
```





```
if(j==0 || (i==0 && j<n/2) || (i==n/2 && j<n/2)
    || (i==n-1 && j<n/2) ||
    (j==n/2 && i!=0 && i!=n/2 && i!=n-1))
```



D1 D2 P1 P2 P3 P4

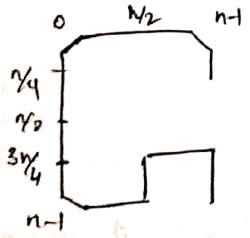
i==j i+j==n-1 i+j==n/2 i-j==n/2 j-i==n/2 i+j==n+n/2-1

0,0	0,10	0,5			
1,1	1,9	1,4			
2,2	2,8	2,3			
3,3	3,7	3,2			
4,4	4,6	4,1			
5,5	:	5,0			
:	:				

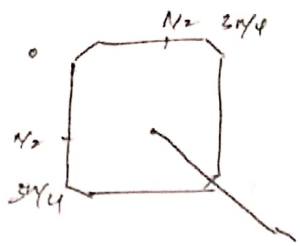
i==0
||j==0
||j==n-1
||i==n-1
||i==j
||i+j==n-1
||i==n/2
||j==n/2

||i+j==n/2
||i-j==n/2
||j-i==n/2
||i+j==n+n/2-1

2/2)



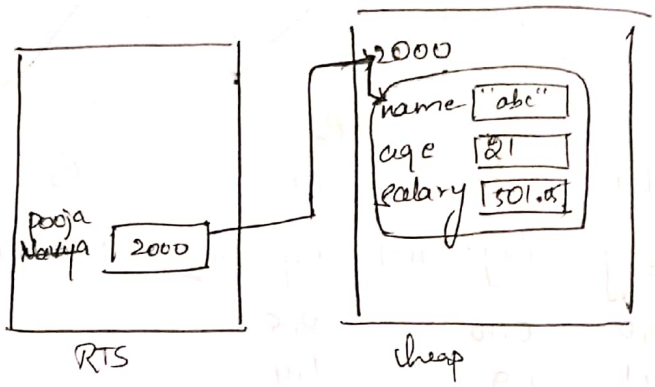

```
if (j==0 && i!=0 && i!=n-1
 || i==0 && j!=0 && j!=n-1
 || i==n-1 && j<=n/2 && j!=0
 || i==3*n/4 && j>=n/2
 || j==n-1 && i<=n/4 && i!=0
 || j==n/2 && i>=3*n/4
 || j==n-1 && i>=3*n/4)
{ sop("*");
}
```




```
if (i==0 && j<3*n/4 && j!=0
 || i==3n/4 && j<3n/4 && j!=0
 || j==0 && i<3*n/4 && i!=0
 || j==3*n/4 && i<3*n/4 && i!=0
 || i==j && i>=n/2)
```

Object Creation

```
class Programmer{
    String name;
    byte age;
    double salary;
}
```




```
class Alpha {
    public static void main (String args[]) {
        Programmer pooja = new Programmer();
        pooja.name = "abc";
        pooja.age = 21;
        pooja.salary = 501.05;
        SOP(pooja.name);
        SOP(pooja.age);
        SOP(pooja.salary);
    }
}
```

dy: Object is created from a class (blueprint that describes behavior/state of that object)
state (fields) behavior (methods)

Not
The objects will get created in the heap area.

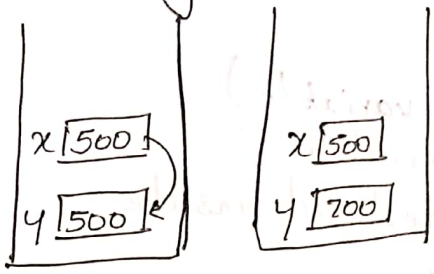
The reference to the object will be created in the stack.

All the class names are either noun/pronoun the name & the methods are usually verb.

Value type assignment & reference type assignment.

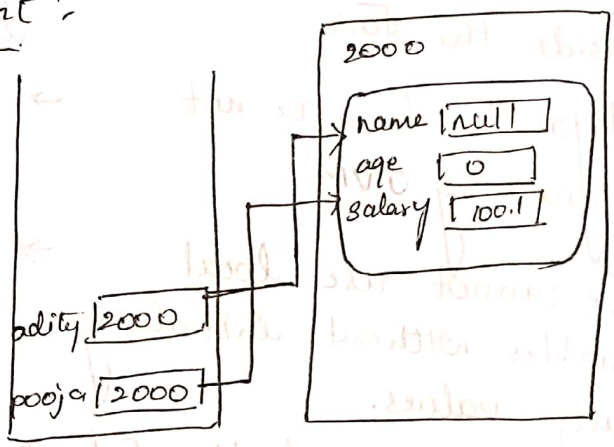
Value type assignment:

```
class Alpha {
    public static void main (String args[]) {
        int x = 500;
        int y;
        y = x
        y = 700;
```



Reference type assignment:

```
class Programmer {
    String name;
    byte age;
    double salary;
}
class Alpha {
    public static void main (String args[]) {
        Programmer pooja = new Programmer();
        Programmer aditya;
        aditya = pooja;
        aditya.salary = 100.1;
        SOP(pooja.salary); // 100.1
    }
}
```



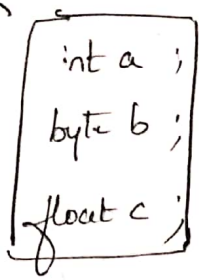
Types of variables :-

variable : name given to a memory location (dyn)

(i) local variable (ii) instance variable (iii) Static variable / class variables

instance variables

```
class Alpha
{
  int a;
  byte b;
  float c;
}
```



local variables

```
class Alpha
{
  void add()
  {
    int a;
    byte b;
    float c;
  }
}
```

(fields)

(Automic variable)

local variable :

→ are declared inside methods

→ local variables are created inside the stack

→ default values are not assigned by JVM

→ We cannot use local variables without initializing any values.

→ Memory will be deallocated one execution of method gets over.

instance variable

→ are declared outside method & inside the class

→ are created inside the heap area.

→ default values are assigned by the JVM.

→ We can use instance variables without initializing any value.

→ [Demon thread] Garbage collector deallocates the memory of instance variables when there is no reference to the object.

local: variables declared inside methods, constructors or blocks

instance: variables declared inside the class, outside any method.

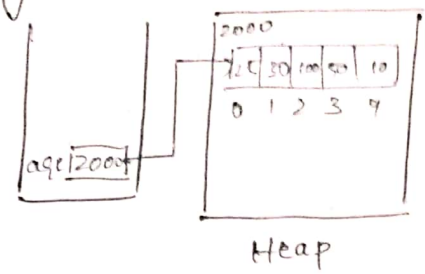
class/static: inside class, outside methods along with static keyword

```
for(int i=0; i<n; i++)
{
  SOP(*age[i] + " ");
}
```

Arrays

declaration of an array :→ `byte age[] = new byte[5];`

```
age[0] = 25;
age[1] = 30;
age[2] = 100;
age[3] = 50;
age[4] = 10;
```



→ Array is a collection of homogeneous data
→ In java array is an object.
→ array gets created inside the heap area.
→ default values are assigned by the JVM to the whole array.
→ while creating an array size should be mentioned.
→ array size is fixed. We can neither enlarge nor decrease.
→ We can create an array of any type.
→ Every element in an array has the index & the index value starts from zero.
→ we can create an array of size zero as well but cant create an array of negative size.
→ maximum size of an array is $2^{31} - 1$.

```
class MyArray {
    public static void main (String[] args) {
        Scanner scan = new Scanner (System.in);
        int n = scan.nextInt();  → sop ("Enter size");
        int age[] = new int[n];  → sop ("Enter elements");
        for (int i=0; i<n; i++) {
            int age[] = scan.nextInt();
        } sop (age[i]);
    }
```

→ If we try to access element beyond the limit it would give an exception saying "ArrayIndexOutOfBoundsException"

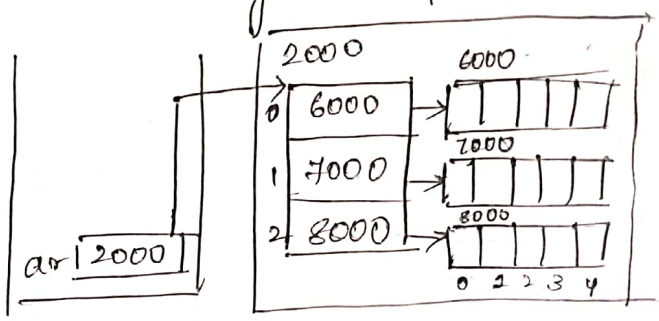
→ If we want to get the size of an array we can get it from variable name called 'length'
[arrayname.length]

Two dimentional array

declaration : `int ar[][] = new int[3][5];`

Memory map

classroom	no of students
0	0 1 2 3 4
1	0 - - - 4
2	0 - - - 4



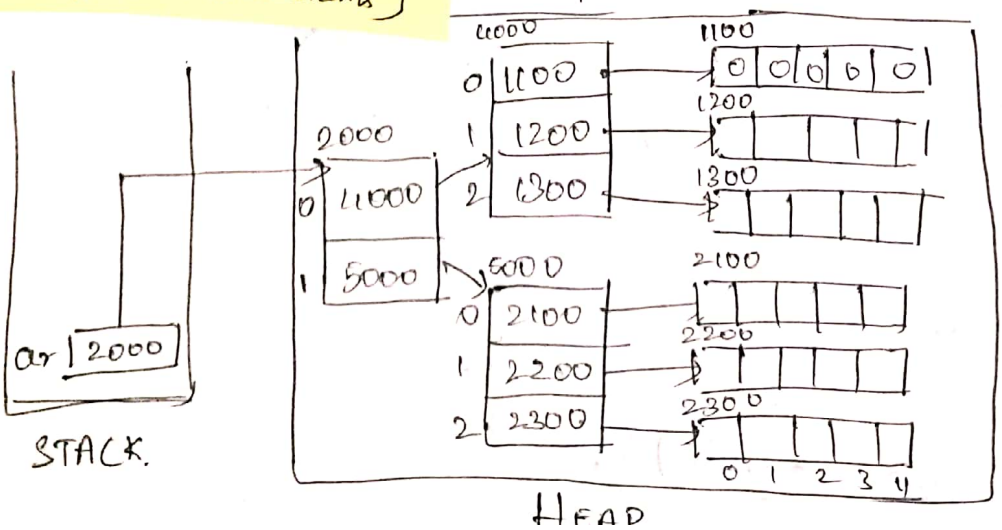
Three dimentional array

declaration : `int ar[][][] = new int[2][3][5]`

1-D/single : elements represented by single subscript

2-D : elements represented by 2 subscripts [has mxn elements]

3-D : by 3 subscripts [has mxnxl elements]



→ If we try to access element beyond the limit it would give an exception saying "ArrayIndexOutOfBoundsException"

→ If we want to get the size of an array we can get it from variable name called 'length'

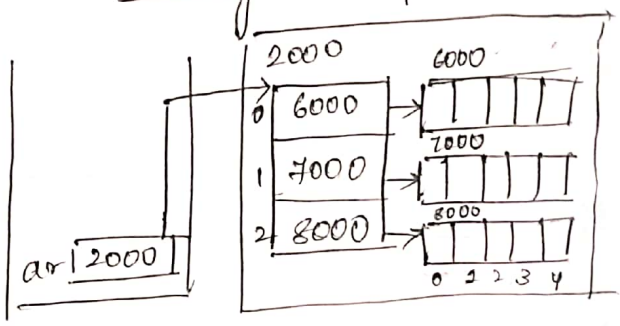
[arrayname.length]

Two dimentional array

Declaration: `int ar[][] = new int[3][5];`

classroom	no of students
0	0 1 2 3 4
1	0 - - - 4
2	0 - - - 4

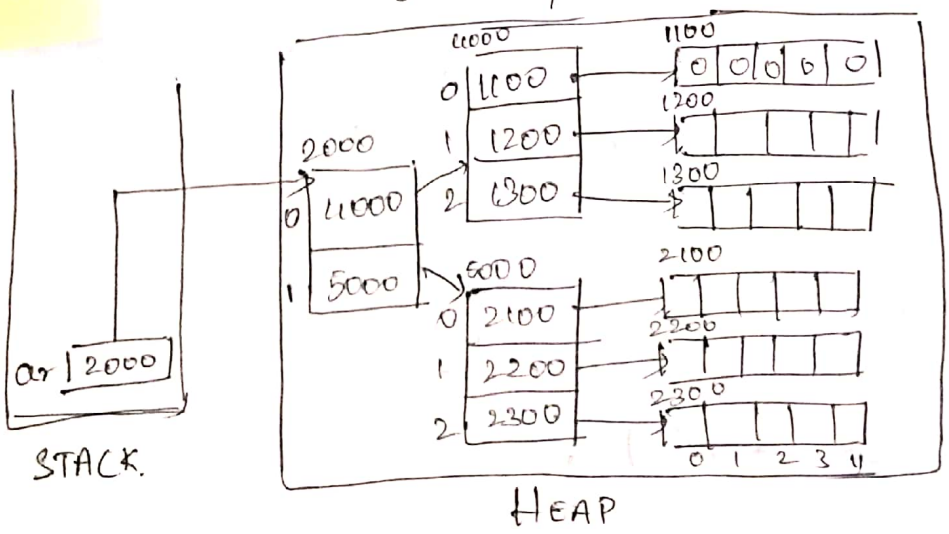
Memory map



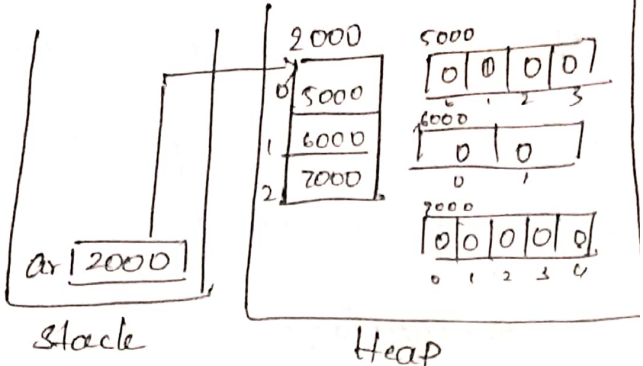
Three dimentional array

Declaration: `int ar[][][] = new int[2][3][5]`

class room	no of students
0	0 1 2 3 4
1	0 - - - 4
2	0 - - - 4
0	0 - - - 4
1	0 - - - 4
2	0 - - - 4



class room	students
0	0, 1, 2, 3
1	0, 1
2	0, 1, 2, 3, 4



Jagged array / ragged arrays → arrays containing arrays of different length

```
public class JaggedArray {
    public static void main(String args[]) {
        int ar[][] = new int[3][];
        SOPln(ar[0]);
        SOPln(ar[1]);
        SOPln(ar[2]);   // prints null (: null is the default reference value)
        ar[0] = new int[4];
        ar[1] = new int[2];
        ar[2] = new int[5];
        SOPln(ar[0]);
        SOPln(ar[1]);
        SOPln(ar[2]);   // prints reference of each array.
    }
}
```

Possibilities of declaring an array

```
int ar[];
int []ar;
int[] ar;
int ar[][];
int [][]ar;
int[][] ar;
int []ar[];
int[] []ar;
int [][] ar;
```

Adv of an array

- Can store multiple elements under a same name
- Easy to access an element in an array

Disadvantage

- Size of an array is fixed, when array is created.
- Heterogeneous type of data cannot be stored in an array
- memory shd be contiguous in an array.

Sum in an array

```
class Sum {
    public static void main (String args[]) {
        Scanner scan = new Scanner (System.in)
        int n = scan.nextInt();
        int ar[] = new int[n];
        for (int i=0; i<ar.length; i++) {
            ar[i] = scan.nextInt();
        }
        int sum=0;
        for (int i=0; i<ar.length; i++) {
            sum = sum+ar[i];
        }
        SOP (sum);
    }
}
```

ar → | 2 | 1 | 5 | 4 | 3 |
0 1 2 3 4 < 5
ar.length

Note =

```
for ( , ) {
    if (ar[i] % 2 == 0) {
        sum = sum+ar[i];
    }
```

Program to find maximum no in an array.

```
int max = ar[0];
for(int i=0; i<ar.length; i++){
    if(ar[i] > max){
        max = ar[i];
    }
}
SOP(max);
```

Shortcuts

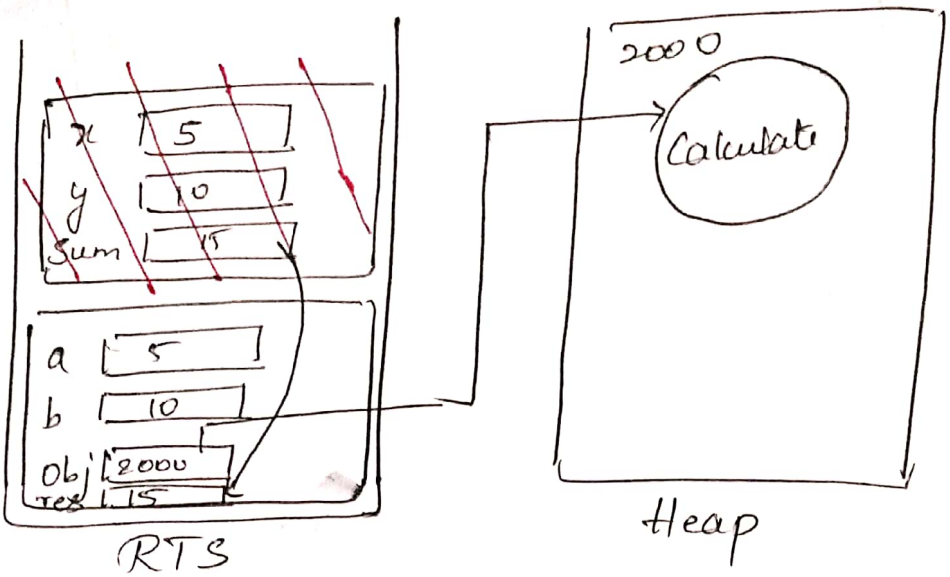
ctrl + m ⇒ maximize/minimize screen
ctrl + d ⇒ to delete the particular line
ctrl + n → new package

Methods

```
output name(input)
{
  =
  =
}
```

```
return_type method_name(parameters)
{
    // body.
}
```

def: Method is a collection of statements which performs a specific task & returns the data to the caller (main method).

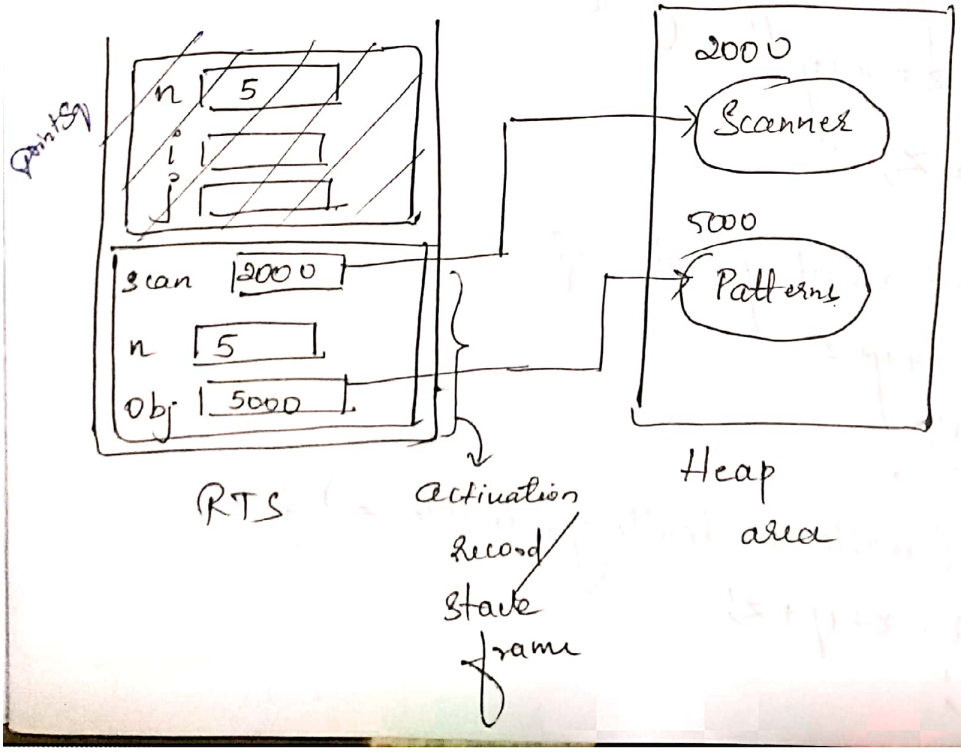



```
class Calculate{
  int add(int x, int y)
  {
    int sum = x+y;
    return sum;
  }
}

class Alpha{
  public static void main(String args[]){
    Scanner scan = new Scanner(System.in);
    int a = scan.nextInt();
    int b = scan.nextInt();
    Calculate obj = new Calculate();
    int res = obj.add(a,b);
    SOP(res);
  }
}
```

```
class Patterns {
    void printSquare(int n)        → parameters
    {
    {
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++){
                SOP("*");
            }
            SOPln();
        }
    }
}

class Alpha {
    public static void main (String args[]) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        Patterns obj = new Patterns();
        obj.printSquare(n);
    }
}
```

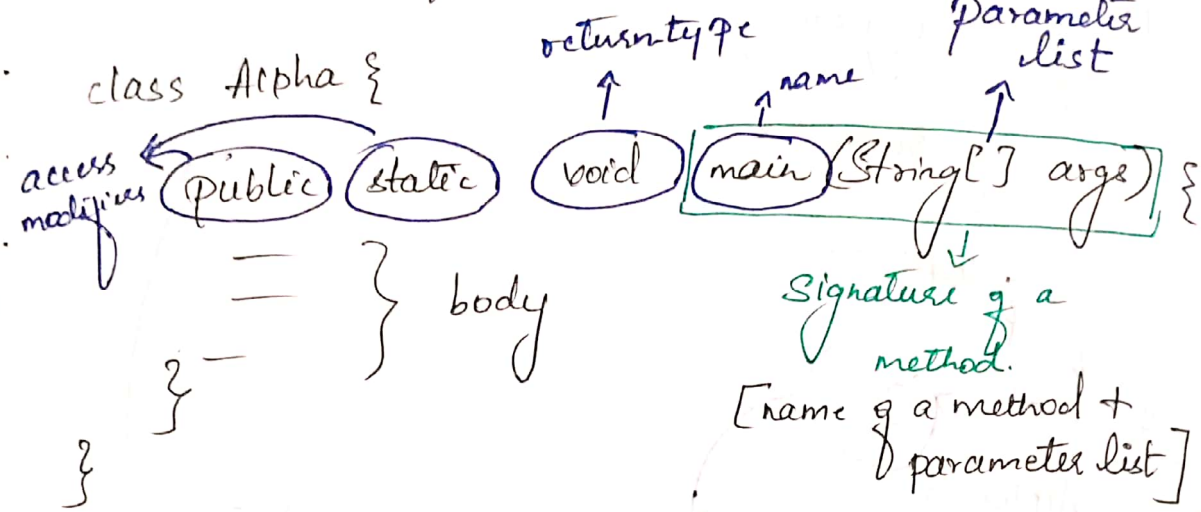
OS

arguments



Note The variables/data passed in the method call is called as arguements.

The data present in the method body is called as parameters.



Method Overloading:

defn: feature that allows to have same method name with different parameters in a single class.

```
class Calculate {
    int add(int x, int y){
        int z = x+y;
        return z;
    }
    float add(float x, float y){
        float z = x+y;
        return z;
    }
    int add(int x, int y, int z){
        int a = x+y+z;
        return a;
    }
    double add(double x, double y, double z) {
        double a = x+y+z;
        return a;
```

```
public class MethodOverloading {
    public static void main(String args[]) {
        Calculate obj = new Calculate();
        int res = obj.add(3,7);
        float res2 = obj.add(5.2f, 7.6f);
        int res3 = obj.add(10,20,30);
        double res4 = obj.add(2.1, 3.6, 6.6);
        SOPln(res);
        SOPln(res1);
        SOPln(res2);
        SOPln(res3);
    }
}
```

Def^n: Method Overloading is methods having same name inside a single class

Adv

→ Developer need not remember the method names while calling.

Note 1. We can overcome naming conflict in method overloading with a help of

→ no of parameters.
→ datatype of the parameters.
→ order of the datatype.

2. Method Overloading is considered as false polymorphism.

→ compile time polymorphism.
→ Static polymorphism.
→ virtual polymorphism.

Different names

Method Overloading with Type Promotion:

- In method overloading when a method is called when there is no exact match
- The compiler finds the closest match if there is no exact match by performing implicit type casting & call that method

```
class Calculate {
    long add (long x, long y) {
        long sum = x+y;
        return sum;
    }
    double add (double x, double y) {
        double sum = x+y;
        return sum;
    }
}

class Alpha {
    public static void main (String[] args) {
        byte x = 10, y = 20;
        Calculate obj = new Calculate();
        obj.add(x, y);
    }
}
```

byte → long, byte → long.

```
class Calculate2 {
    long add(long x, long y) {
        SOPln("method-1");
        long sum = x+y;
        return sum;
    }
    double add(double x, double y){
        SOPln("method-2");
        double sum = x+y;
        return sum;
    }
class TypePromotion {
    public static void main(String args[]) {
        byte x = 10;
        float y = 20.7f;
        Calculate obj = new Calculate();
        double res = obj.add(x,y);
    }
}
```

byte → double, float → double : method-2

byte → long, float → ~~long~~ : method-1

Problems with type promotion in method overload

```
class Calculate {
    float add (float x, int y) {
    float sum = x+y;
    return sum;
    }
    float add (int x, float y) {
        =
    }
}
class Alpha {
    psvm (String args[]) {
        int x = 10;
        int y = 20;
        Calculate obj = new Calculate ();
        obj.add (x, y);
    }
}
```

int int ⟹ int, float ✗
float, int ✗

Note: It shows an error (unambiguous)

" Can we have multiple main methods inside a same class? " - Yes we can overload the main method provided the signature of the main methods are different.

```
public class OverloadMainMethod {
    psvm (String args[]) {
        SOPln ("main-1");
    }
    psvm() {
        SOPln ("main-2");
    }
```

O/P
main-1

Var-arg methods (method available from version 1.5 onwards)

→ In this case the method can accept any number of parameters of same datatype.

```
class Calculate {                          → accepts any no of parameters
    void display(int... a) {
        SOP(a);
    }
}
class Alpha {
    public static void main(String args[]) {
        Calculate obj = new Calculate();
        obj.display();
        obj.display(10);
        obj.display(10,20,30,80,...);      → any no of arguments can be passed.
    }
}
```

O/P
returns of each array

```
class Calculate {
    void display(int... x) {
        for(int i=0; i<x.length; i++) {
            SOP(x[i]+" ");
        }
        SOPln();
    }
}
public class TypePromotion {
    p s v m(String args[]) {
        Calculate obj = new Calculate();
        obj.display();
        obj.display(10);
        obj.display(10,20,30,40,50);
    }
}
```

O/P

```
10
20 20
10 20 30
10 20 30 40
10 20 30 40 50
```

While & do-while

```
while (condition)
{
    body
}
```

true → body; false → exit

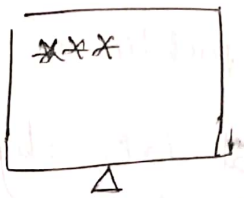
```
do {
    body
} while (condition)
```

true → body; false → exit

executes the target stmt as long as the given condition is true (OR) **While**

the condition is evaluated first & if it returns true then stmt inside while loop is executed

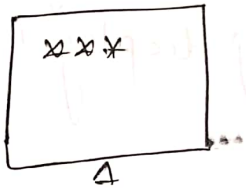
x = 0 x x 3 < 3



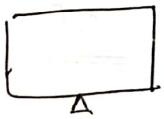
similar to while, except that the condition is checked after the stmts are executed

do-while

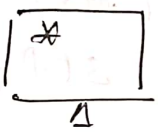
x = 0 x x 3 < 3



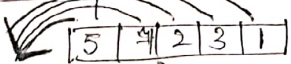
if x=10 in while



if x=10 in do while



For each loop / Enhanced for loop




```
for (int x : ar) {
    SOP(x);
}
```

for each : to traverse each element one by one

alternate approach to traverse an array/collection

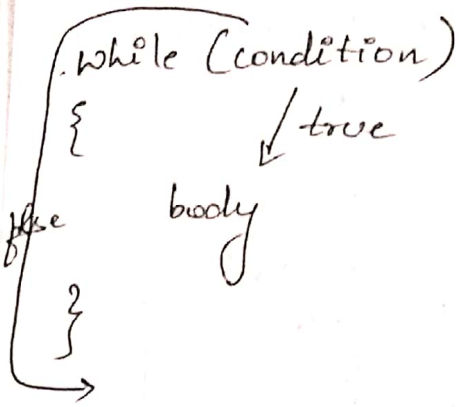
enhanced for loop

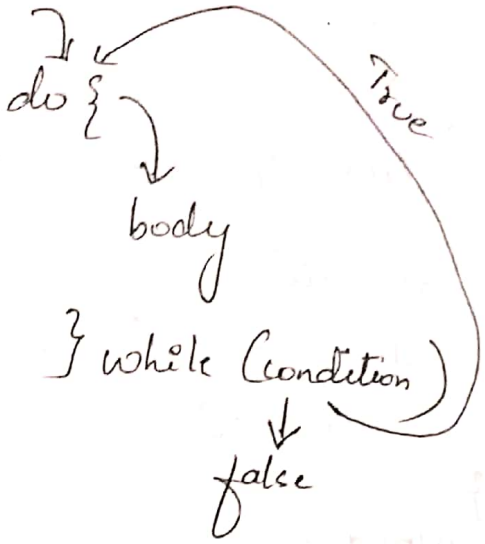
5

→ Can be traversed only from left to right

→ The above loop executes 5 times as there are 5 elements in the array.

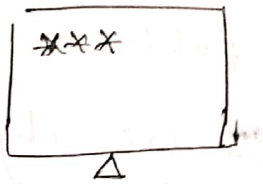
while & do-while





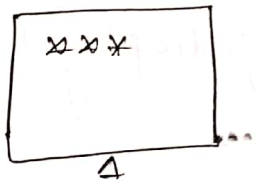

```
int x = 0
while (x < 3) {
    SOP("*");
    x++;
}
```

x = 0 < 3 ✓ x = 1 < 3 ✓ x = 2 < 3 ✓ x = 3 < 3 ✗



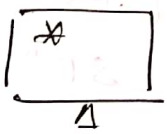

```
int x = 0;
do {
    SOP("*");
    x++;
} while (x < 3);
```

x = 0 < 3 ✓ x = 1 < 3 ✓ x = 2 < 3 ✓ x = 3 < 3 ✗

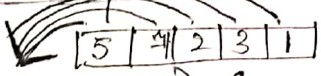


if x = 10 in while

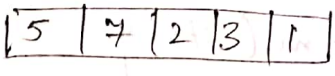
if x = 10 in do while



For each loop / Enhanced for loop




```
for (int x : ar) {
    SOP(x);
}
```



< ar.length

→ Can be traversed only from left to right

→ The above loop executes 5 times as there are 5 elements in the array.

→ In for each loop no' of iterations = no' of elements in the collection.

no of elements

no of iterations = no of elements

Note: Alternative way of declaring & initializing an array:

```
int var[] = {5, 10, 15, 20}
```

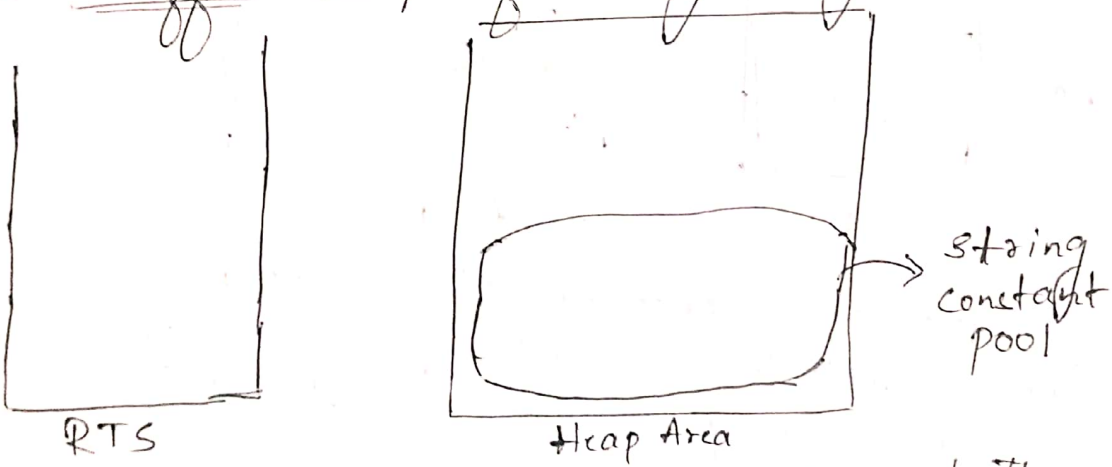
STRING

def: String is a collection/sequence of characters enclosed with double quotes (" ")

Ex: "1234", "*#123", "abcd", "Java@123".

- String is treated as an object in case of java.
- → String is created using `String s = new String()`
- → String is immutable in java.
- → Immutable strings are the strings that cannot be changed or altered.

Different Ways of Creating Strings

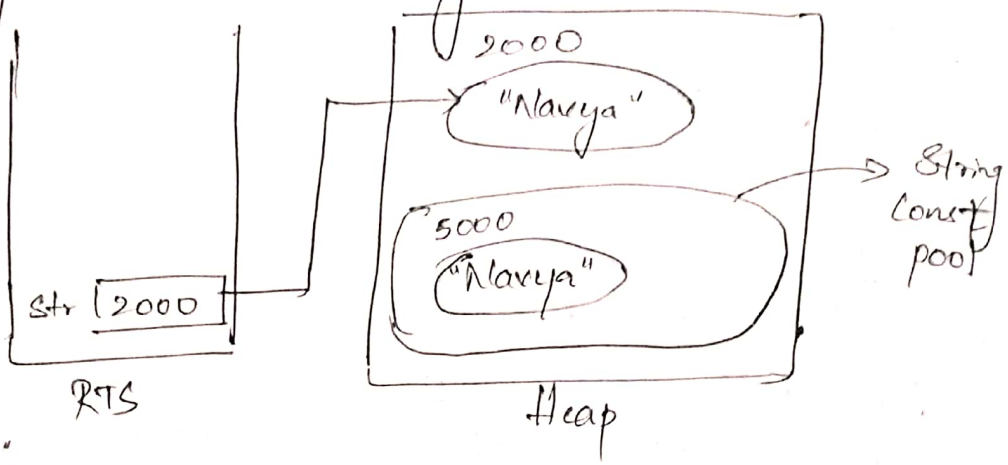


- → String constant pool is a region present inside the heap area
- → Only string objects gets created inside this region.
- → No duplicates are allowed inside this region.

1. String str = new String("Navya"); // in heap & string const pool (if copy is not present)

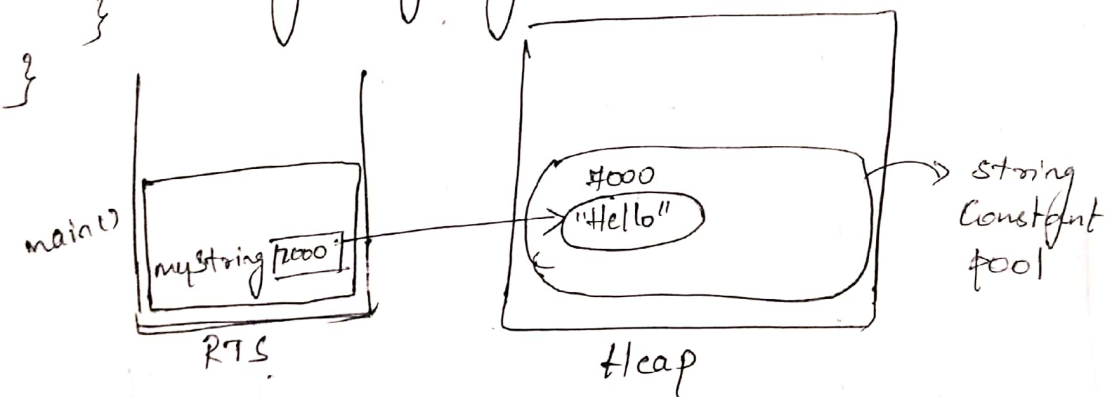
2. String str2 = "Navya"; // in string const pool

3. char ar[] = {'A', 'B', 'C'}; // same as ①
String str3 = new String(ar);



Prog 1:

```
class StringPrg {
    public static void main(String[] args) {
        String myString = "Hello";
    }
}
```

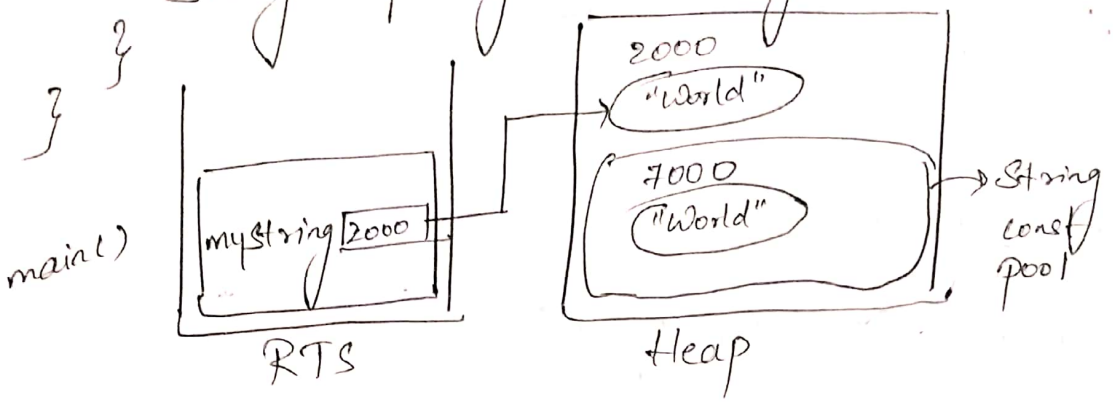


1. Object gets created in string const pool (no duplicates are allowed in case. String myString = "Hello")

2. Checks if there is "new" keyword while creating, if it is present it creates a copy in the heap area and returns the address of it.

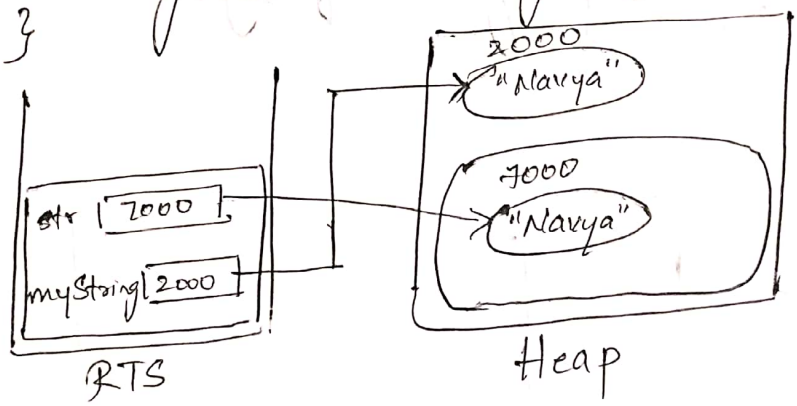
Prg-2:

```
class StringPrg2{
    public static void main(String[] args){
        String myString = new String("World");
    }
}
```



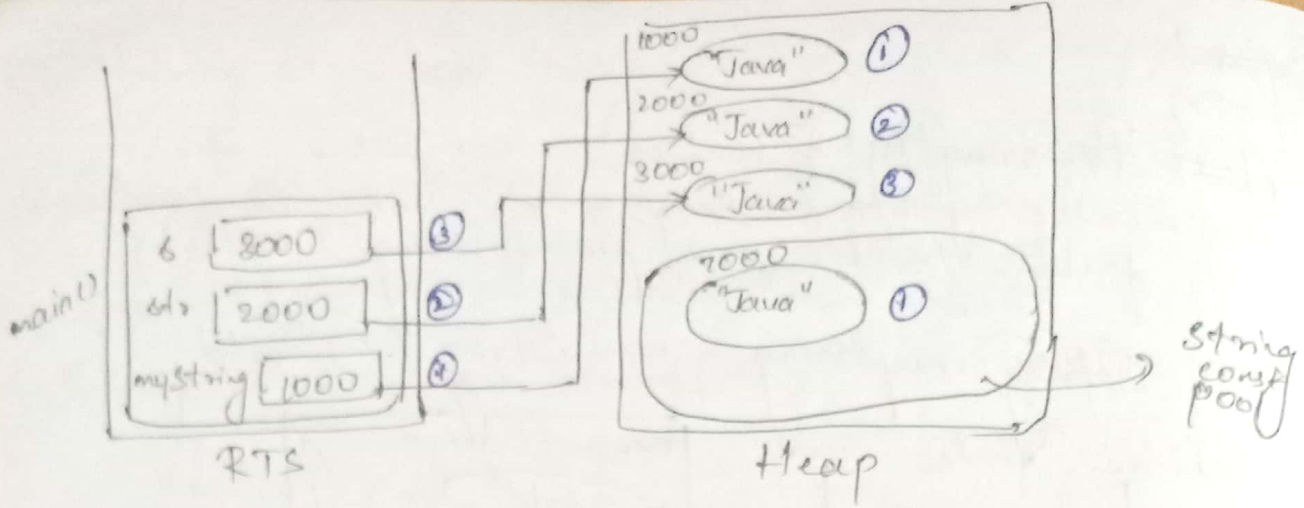
Prg-3

```
class StringPrg3{
    psvm(String[] args){
        String myString = new String("Navya");
        String str = "Navya"; // address of the scp object is stored.
    }
}
```



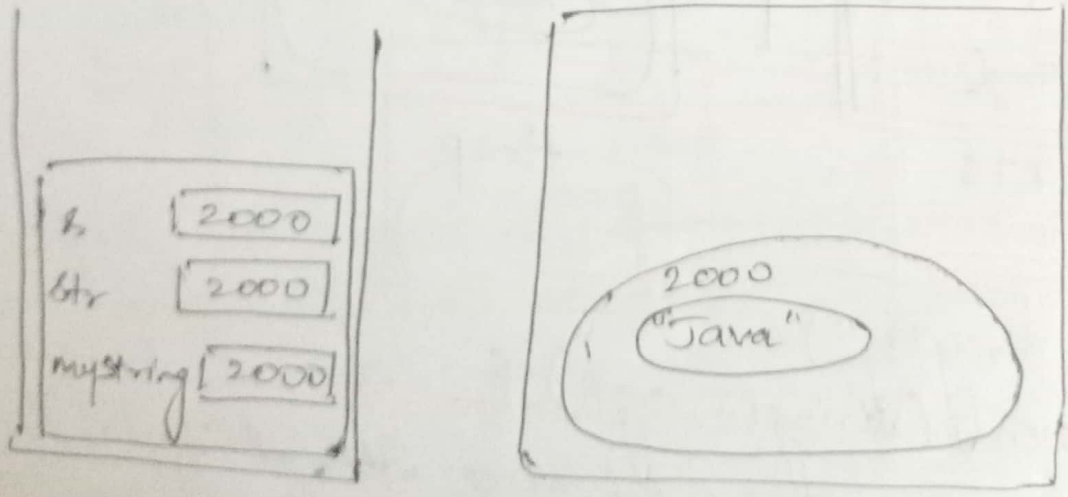
Prg-4

```
class StringPrg4{
    psvm(String[] args){
        ① String myString = new String("Java");
        ② String str = new String("Java");
        ③ String s = new String("Java");
    }
}
```



Prg-5

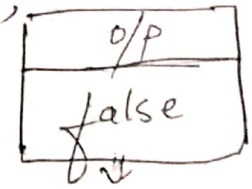
```
class StringPrg5 {
    psvm (String[] args) {
    String myString = "Java";
    String str = "Java";
    String s = "Java";
    SOPln(myString) // print "Java"
    SOPln(str)  //} reference of the
    SOPln(s)    //} object
  }
}
```



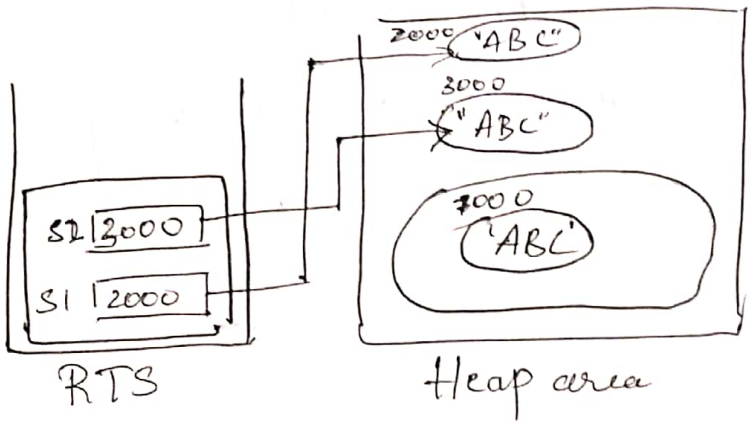
Equals

Prg-6 Equal to operator compares the reference of the string & not the values

```
class Alpha {
    public static void main (String args[]) {
        String s1 = new String ("ABC");
        String s2 = new String ("ABC");
        SOPln (s1 == s2)
    }
}
```

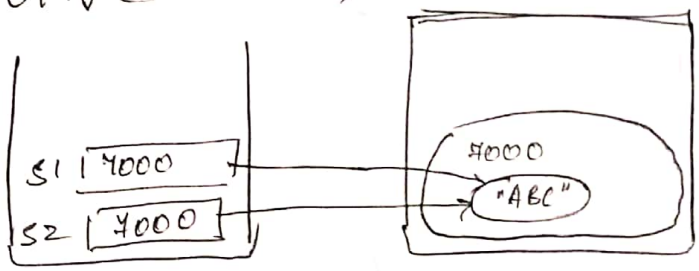


Since references aren't same



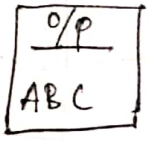
Prg-7

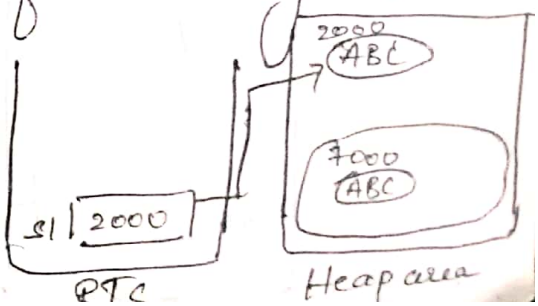
```
String s1 = "ABC";
String s2 = "ABC";
SOPln (s1 == s2)
```



Prg-8 In case of string when we print object reference it prints the value of the string.

```
String s1 = new String ("ABC");
System.out.println (s1);
```





Note: To compare the value of 2 strings we have equals() method. If the strings are equal it returns true otherwise false.

Prg-9

```
String s1 = new String("ABC");
String s2 = new String("ABC");
SOP(s1.equals(s2));
```

O/P
True

Note: Strings are case sensitive.

Prg-10

```
class Alpha{
    psvm(String args[]){
        String s1 = "ABC";
        String s2 = "abc";
        SOP(s1.equals(s2));
    }
}
```

O/P
false

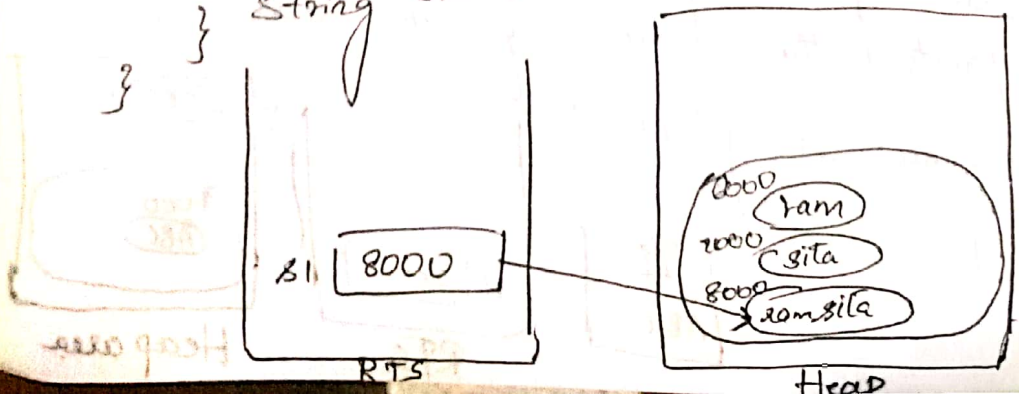
Note: In case of strings "+" does concatination

Ex: 1+1 = 2
"1"+1 = "11"
1+"1" = "11"
"1"+"1" = "11"

1+1+"1" = "21"
1+"1"+1 = "111"
"1"+1+1 = "111"

Prg-11

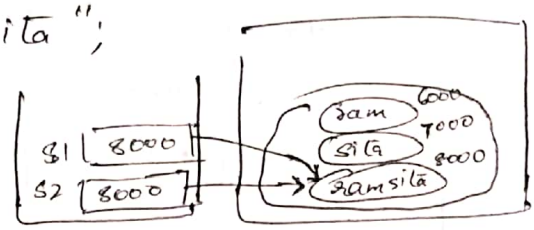
```
class Alpha{
    psvm(String args[]){
        String s1 = "ram"+"sita";
    }
}
```



Prog-12

```
class Alpha {
    psvm(String args[]) {
        String s1 = "ram" + "sita";
        String s2 = "ram" + "sita";
        SOPln (s1 == s2)
    }
}
```

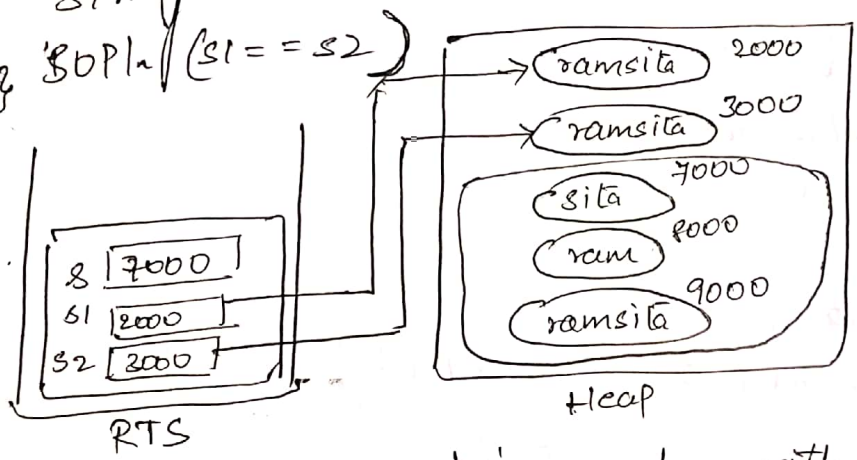
O/P
True



Prog-13

```
class Alpha {
    psvm(String args[]) {
        String s = "sita";
        String s1 = "ram" + s;
        String s2 = "ram" + s;
        SOPln (s1 == s2)
    }
}
```

O/P
false



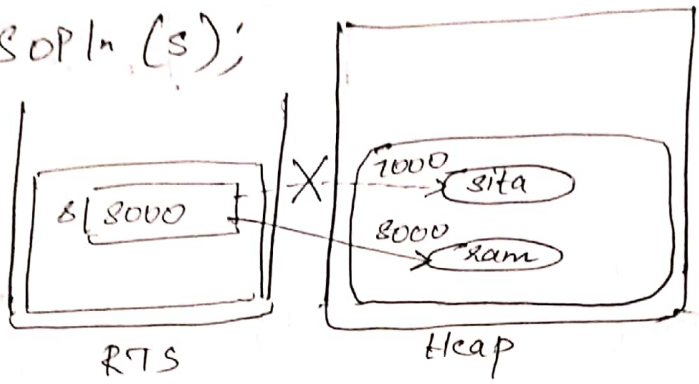
→ When we try to use '+' operator with object (variable) reference always the memory will be allocated in the heap area along with string constant pool.

→ When we try to use '+' operator only with string literals the memory will be allocated in string constant pool.

Pg 14

```
class Alpha{
    Psvm(String args[]) {
String s = "sita";
s = "ram";
SOPln (s);
    }
}
```

O/P
ram



Methods of strings

class String {

1. equals()
2. equalsIgnoreCase()
   :

}

Ex-1

```
class Alpha {
    psvm (String args[]) {
String s1 = "sita";
String s2 = "Sita";
SOPln (s1.equalsIgnoreCase(s2));
    }
}
```

O/P
True

1. equalsIgnoreCase() - it compares whether the content in both of the strings is equal or not by ignoring their case
   ↳ if equal it returns "True"
   ↳ otherwise "False"

3. length()

↳ returns the no of characters present in the string.

```
String s = "siTA";
SOPln(s.length());
```

O/P
4

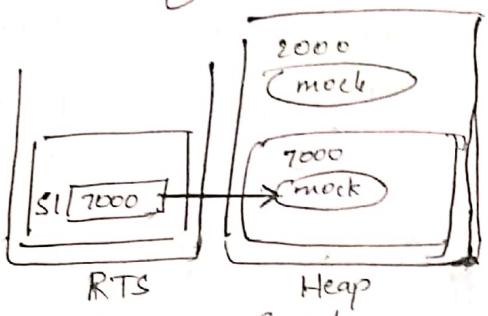
4. repeat()

```
String s = "mock";
SOPln(s.repeat(5));
```

↳ repeats the string for given no of times

5. intern()

```
String s1 = new String("mock");
s1 = s1.intern()
SOPln(s1);
```



↳ intern method creates a copy of the string inside string constant pool and returns the reference/address of it.

[if doesn't create the copy if the object is already present.]

```
String s1 = new String("mock");
String s2 = "mock";
SOPln(s1==s2) // prints "false"
s1=s1.intern
SOPln(s1==s2) // prints "true"
```

O/P
false
true.

ASCII Values

'A' - 65	'a' - 97	'0' - 48
'Z' - 90	'b' - 122	'9' - 57

6. compareTo()- compares if 2 strings are equal or not

```
String s1 = "abc";
String s2 = "abc";
SOPln (s1.compareTo(s2));
```

returns int value

o/p
0

a b c
c a b
97 98 99
99 97 98
-2

abcd

ex1 abc
abc

97 98 99
97 98 99
0 0 0

ex2 abc
abd

97 98 99
97 98 100
0 0 -1

ex3 abc

Since length itself is different
4-3 = 1

i) CompareTo() first compares the length of each string and then returns the difference in it

ii) if they are equal it returns '0' otherwise a non zero value

iii) If the length is same then it compares the ASCII values and return its value.

7. compareToIgnoreCase():

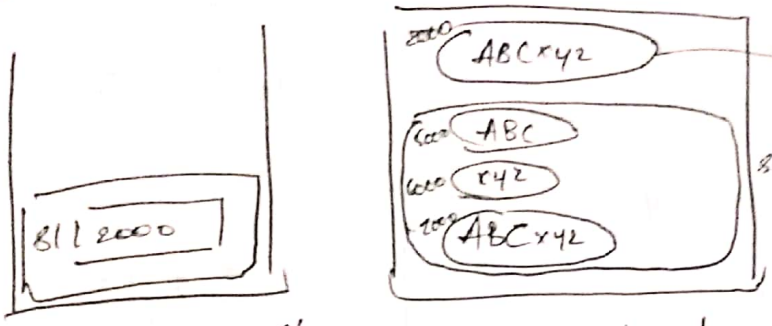
```
String s1 = "ABC";
String s2 = "abcd";
SOPln (s1.compareToIgnoreCase(s2));
```

o/p
-1

8. concat()

```
String s1 = "ABC";
s1 = s1.concat("xyz");
SOPln (s1);
```

o/p
ABCxyz



case1: for variable s1.concat(xyz) is used.

case2: if 'abc'.concat('xyz') it will not create

Note: if s1.concat('xyz') is not stored in s1 then

9. toUpperCase():
   ↳ converts the string to uppercase characters

```
String s1 = "abc";
s1 = s1.toUpperCase();
SOPln(s1)
```

o/p = ABC

```
String s1 = "abc"
s1.toUpperCase();
SOPln(s1)
```

o/p: abc

10. toLowerCase():
    ↳ converts alpha characters to lowercase

```
String s1 = "ABC";
s1 = s1.toLowerCase();
SOPln(s1);
```

o/p: abc

```
String s1 = "Hello";
s1 = s1.toLowerCase();
SOP(s1);
```

o/p: hello

11. contains();
    ↳ checks if particular string is present or not
    ↳ if present returns "true" otherwise "false"

```
String s1 = "wake up"
SOPln(s1.contains("up"));
```

o/p: true

```
String s1 = "wake up"
SOP(s1.contains("UP"))
```

o/p: false

12. startsWith():

↳ checks if the particular string starts with particular/set of characters

```
String s1 = "wake up";
SOP(s1.startsWith("wa"));
```

o/p: true

wa → true
aa → false

13. endsWith():

↳ checks if the string ends with particular/set of characters.

```
String s = "wake up";
SOP(s.endsWith("up"));
```

o/p: true

14. trim():

↳ removes extra spaces at start and end of the string.

```
String s = "    wake up  ";
SOPln(s); // prints along with the spaces
SOPln(s.trim); // prints without any space.
```

o/p:

```
    wake up
wake up
```

15. charAt():

↳ fetches the character from the given index value

```
String s = "hello";
SOPln(s.charAt(1));
```

o/p: e

hello (index value starts from zero)
0 1 2 3 4

Note: s.charAt(6);

o/p: Exception: StringOutOfBounds

16. indexOf()

↳ returns the ~~character or set~~ index of particular/set of characters if present

```
String s = "hello";
SOP(s.indexOf("e"));
```

op: 1

'l' → returns 2

'ell' → returns 1

(returns index of 1st occurance of the character)

- For s.indexOf("a"): → returns -1.

If the character is not present in the particular string it returns -1.

{write the character in single quotes
write the string in double quotes}

17. subString(x, y):

↳ exclusive
↳ inclusive

↳ returns the substring of particular string.

```
String s = "mississippi";
           0123456789 10
SOP(s.subString(4,8)); // returns string from index 4-7
```

o/p: issi

(9,11) → returns 'pi'

(9,12) → Exception: StringOutOfBounceException

(4) → prints from index 4 to end "issippi".

18. split():

↳ splits the given string.

```
String s = "How are you";
SOP(s.split(" "));
```

returns the reference of the string

Ex
```
String s = "mango, apple, banana, orange, jackfruit";
String[] fruits = s.split(",");
for(String fruit : fruits) {
    SOPln(fruit);
}
```

output
mango
apple
banana
orange
jackfruit

Note
↳ ~~There are~~ Strings can be mutable by creating the object of StringBuffer or StringBuilder class

↳ StringBuffer is thread safe.
↳ StringBuilder is not thread safe.

convert int to String

```
~~String s~~ int x = 565;
String s = x + "";
SOP(s);
```

scan.next() ⟶ takes only one word
scan.nextLine() ⟶ takes 1 or multiple words.

Methods to convert String to a different datatype:

1. String to Byte:
```
String s = "456";
byte n = Byte.parseByte(s);
SOPln(n);
```

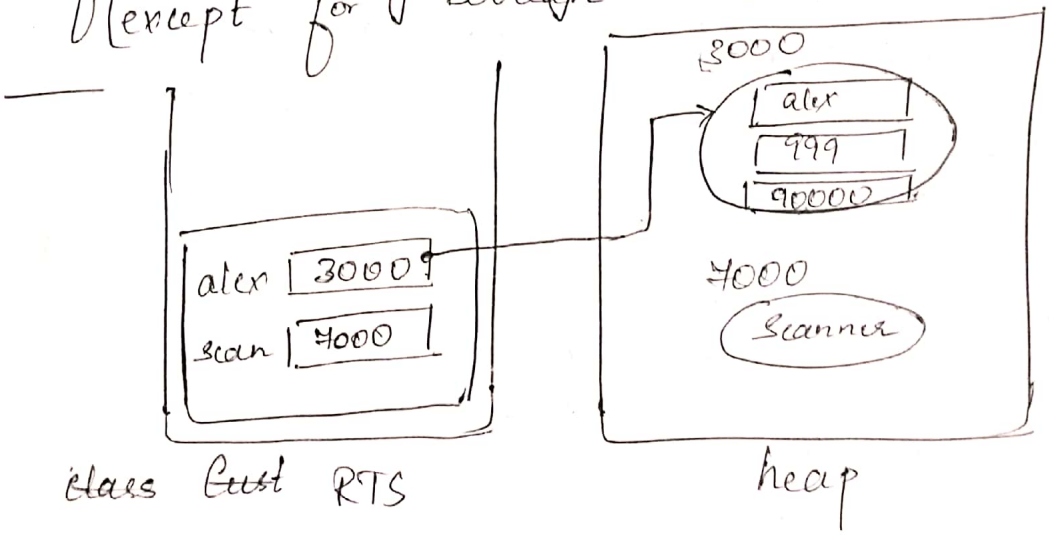
2. String to int → Integer.parseInt(s);
3. String to short → Short.parseShort(s);
4. String to long → Long.parseLong(s);

5. String to float → Float.parseFloat(s);
6. String to double → Double.parseDouble(s);
7. String to Boolean → Boolean.parseBoolean(s);
returns 'true' only if the input string is 'TRUE', 'true', 'True', 'TrUE' and 'false' otherwise ["abcd..."]

```
String s = "true";
Boolean n = Boolean.parseBoolean(s);
SOP(n); // prints true.
```

for → s = "abhi"; // prints false.
→ s = "false"; // prints false

- If String shd contain only numeric characters for converting string to different data types [except for Boolean it shd have alphabets].



Note: • In the program given we are able to access the data of customer of cls. from outside the customer class, because of this there is no security given to the data
• We can secure the data by declaring the variables as private

```
class Customer {
    String name;
    long cnNum;
    double balance;
}
```

OS → JVM

```
class Anonymous {
    public static void main(String args[]) {
        Customer alex = new Customer();
        Scanner scan = new Scanner(System.in);
        alex.name = scan.nextLine();
        alex.cnNum = scan.nextLong();
        alex.balance = scan.nextDouble();
        SOPln("Name :" + alex.name);
        SOPln("cn No :" + alex.cnNum);
        SOPln("Balance :" + alex.balance);
    }
```

- The data declared private can be accessed only within the class and it is hidden from outside the class. This process is called as data hiding.
- If we have to give access to the data we have to give the controlled access.
- Controlled access can be given by using setters & getters [ie accessors & mutators]
- If declared as public it can be accessed outside the class.

class Customer

- The setters and getters has to be declared as public so that it is accessible from outside the class.

Encapsulation :

The process of binding the code & the data into a single unit is called as encapsulation.

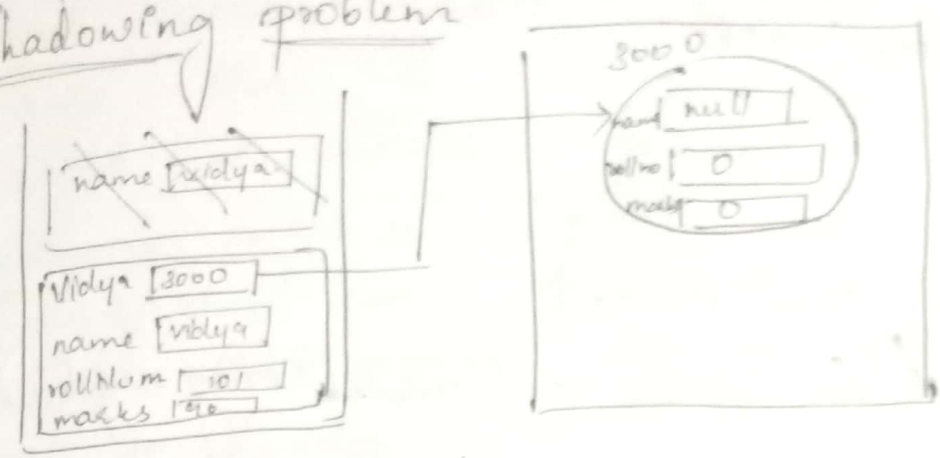
code + data → single unit

```
class Customer {

    private String name;
    private long conNum;
    private double balance;

    public void setName(String s) {
        name = s;
    }
    public void setConNum(long cn) {
        conNum = cn;
    }
    public void setBalance(double bal) {
        balance = bal;
        if (bal > 0) {
            balance = bal;
        }
        else {
            SOPln("invalid input");
        }
    }

    public String getName() {
        return name;
    }
```

```
public long getCrnNum() {
        return crnNum;
    }
public double getBalance() {
        return balance;
    }

class Alpha {
    public static void main(String args[]) {
    SOPln("Enter the details");
    Customer alex = new Customer();
    Scanner scan = new Scanner(System.in);
    String nm = scan.nextLine();
    alex.setName(nm);
    long crn = scan.nextLong();
    alex.setCrnNum(crn);
    double bal = scan.nextdouble();
    alex.setBalance(bal);
    SOPln("Name:" + alex.getName());
    SOPln("CRN:" + alex.getCrnNum());
    SOPln("Balance:" + alex.getBalance());
    }
}
```

import

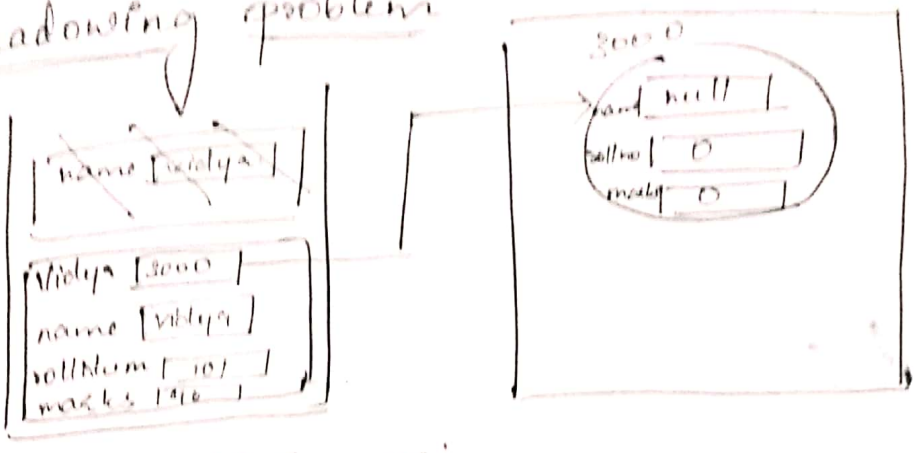
Shadowing problem




```
import java.util.Scanner;
class Student {
    private String name;
    private int rollNum;
    private byte marks;
    public void setName(String name) {
        name = name;
    }
    public void setRollNum(int rollNum) {
        rollNum = rollNum;
    }
    public void setMarks(byte marks) {
        marks = marks;
    }
    public String getName() {
        return name;
    }
    public int getRollNum() {
        return rollNum;
    }
}
```

Important

Shadowing problem




```
import java.util.Scanner;
class Student {
    private String name;
    private int rollNum;
    private byte marks;
    public void setName(String name) {
        name = name;
    }
    public void setRollNum(int rollNum) {
        rollNum = rollNum;
    }
    public void setMarks(byte marks) {
        marks = marks;
    }
    public String getName() {
        return name;
    }
    public int getRollNum() {
        return rollNum;
    }
}
```

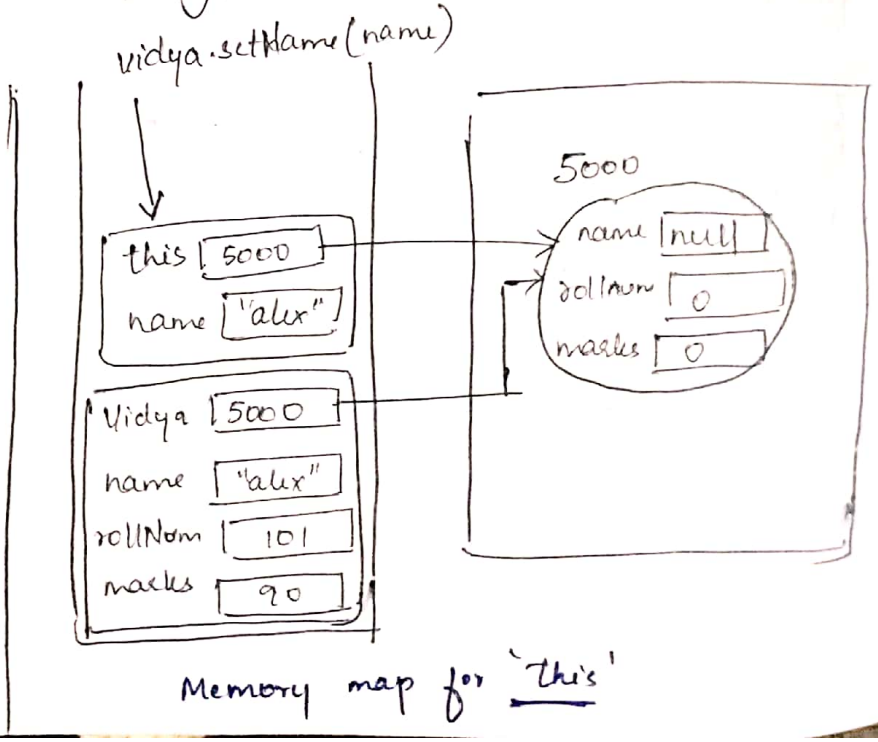
```
    public byte getMarks(){
        return marks;
    }
}
```

```
et public class Alpha{
    public static void main(String args[]){
        Student vidya = new Student();
        Scanner scan = new Scanner(System.in);
        String name = scan.nextLine();
        int rollNum = scan.nextInt();
        byte marks = scan.nextByte();
        vidya.setName(name);
        vidya.setRollNum(rollNum);
        vidya.setMarks(marks);
        SOPln(vidya.getName());
        SOPln(vidya.getRollNum());
        SOPln(vidya.getMarks());
    }
}
```

Output

i/p: Vidya
101
90

o/p: null
0
0



Memory map for 'this'

- In the above program since instance variables and local variables are having the same name there is a naming conflit called as shadowing problem.
- Shadowing refers to a practice where 2 variables having the same name with 2 different scope, the variable having higher scope will be hidden.
- We can overcome shadowing problem using the keyword called 'this'
- 'this' refers to the current object inside a method or a constructor

Program using 'this' keyword:

```
import java.util.Scanner;
class Student {
    private String name;
    private int rollNum;
    private byte marks;

    public void setName (String name) {
        this.name = name;
    }
    public void setRollNum ( int rollNum) {
        this.rollNum = rollNum;
    }
    public void setMarks ( byte marks) {
        this.marks = marks;
    }
```

this.name → instance variable
name → local variable

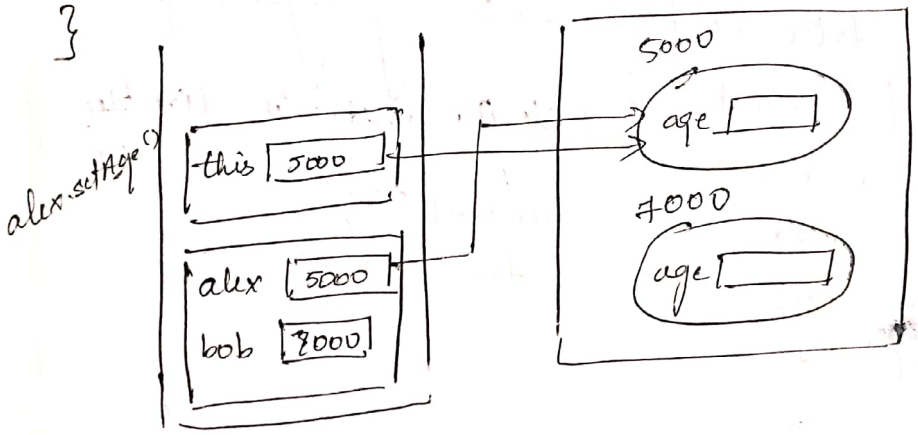
```
public String getName(){
    return this.name;
}
public int getRollNum(){
    return this.rollNum;
}
public byte getMarks(){
    return this.marks;
}
}
public class Alpha{
    public static void main(String args[]){
    SOPln("Enter the details");
    Student vidya = new Student();
    Scanner scan= new Scanner(System.in);
    String name = scan.nextLine();
    int rollNum = scan.nextInt();
    byte marks = scan.nextByte();
    vidya.setName(name);
    vidya.setRollNum(rollNum);
    vidya.setMarks(marks);
    SOPln(vidya.getName());
    SOPln(vidya.getRollNum());
    SOPln(vidya.getMarks());
    }
}
```

returns instance variable

Output

```
i/p  Vidya
     101
     90

o/p  Vidya
     101
     90
```

```
class Person {
    int age;
    public void setAge(int age){
        SOPln(this);
        this.age = age;
    }
}
public class Alpha {
    public static void main(String args[]) {
        Person alex = new Person();
        Person bob = new Person();
        alex.setAge(20);
        SOPln(alex);
        bob.setAge(30);
        SOPln(bob);
    }
}
```



Advantages of encapsulation :-

- We can make the fields / instance variables of a class as read only or write only by having only setters & getters
- The user would not get to know what exactly is happening behind the scenes (Abstraction)
- Improves maintainability & reusability.

Abstraction: Hiding certain details and showing only essential info to the user. It can be achieved by
↳ abstract classes
↳ interfaces

Constructor :-

```
class Student {
    private String name;
    Student(){
        SOPln("Student object created");
    }
}
public class Constructor {
    public static void main(String args[]){
        SOPln("start");
        String alex = new Student();
        Student alex = new Student();
        Student bob = new Student();
        SOPln("Stop");
    }
}
```

calls the constructor directly.

Output

```
Start
Student object created
Student object created
Stop.
```

- Constructor is a block of code that initializes newly created object
- Constructor has same name as the class.
- Constructor doesn't have return type.
- Constructors are called when a new instance of object is created.

When there is no constructor written by a programmer then the default constructor will be created by the compiler (ie zero parametarized constructor)

```
class Student{
    private String name;
    private byte marks;
    Student() {
        SOPln("student object created");
        name = "unknown";
        marks = 40;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public byte getMarks() {
        return marks;
    }
}
public void setMarks(){
    this.marks = marks;
}

public class Constructor {
    public static void main(String args[]) {
        SOPln("Start");
        Student bob = new Student();
        SOPln("bob details are");
        SOPln(bob.getName());
        SOPln("Marks: " + bob.getMarks());
```

```
        SOPln("stop");
    }
}
```

output

```
start
Student object is created
bob details are
Name: Unknown
Marks: 70
stop.
```

Alt + shift + s → to generate getters & setters

Constructor Overloading :-

Constructor : is block of code that initializes a newly created object, resembles an instance method but doesn't have any return type.

```
class Student {
    private String name;
    private int marks;

    Student() {
        SOPln("no values assigned");
    } ------ (1)

    Student(String name) {
        this.name = name
        SOPln("name assigned");
    } ------ (2)

    Student(int marks) {
        SOPln("marks assigned");
        this.marks = marks;
    } ------ (3)

    Student(String name, int marks) {
        SOPln("name & marks assigned");
        this.name = name;
        this.marks = marks;
    } ------ (4)
```

```
public class ConstructorOverload {
    public static void main(String args[]){
        Student alex = new Student();          (1)
        Student bob = new Student("bob",35);   (4)
        Student carl = new Student("carl");    (2)
        Student david = new Student(99);       (3)
    }
}
```

Output

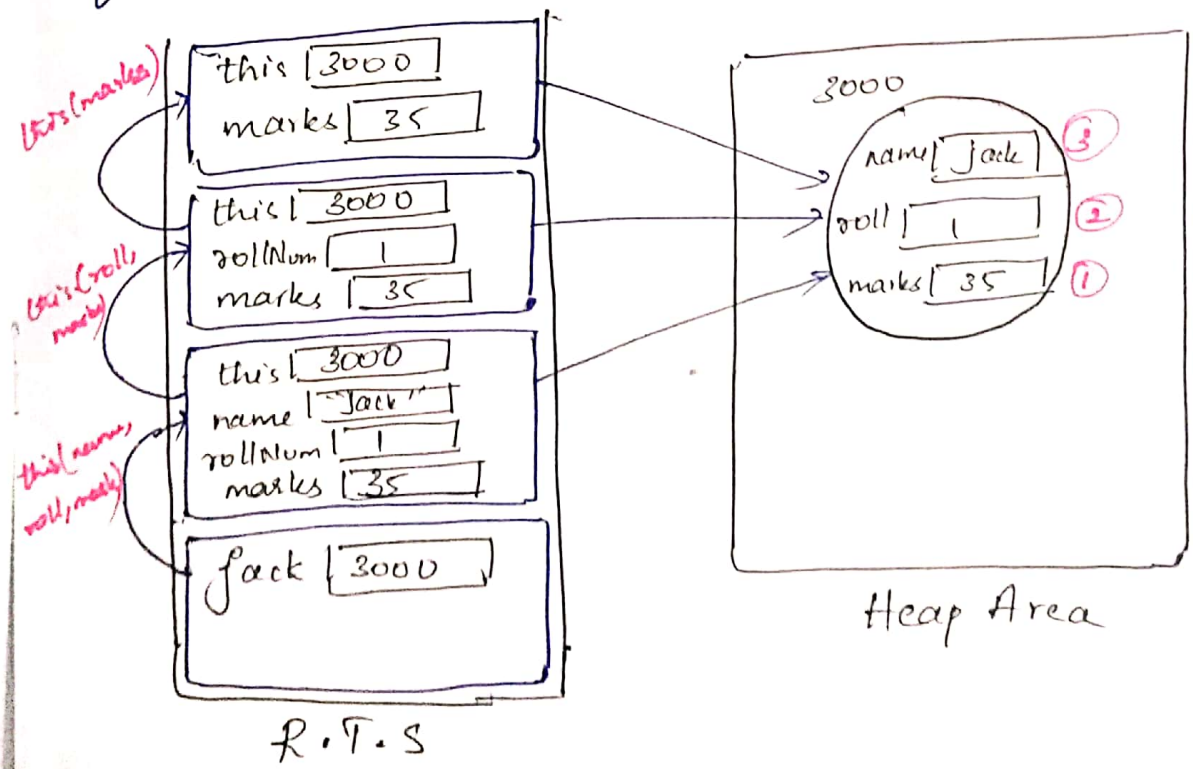
(1) --- no values assigned

(4) --- name & marks assigned
- bob
- 35

(2) --- name assigned
- carl

(3) --- marks assigned
- 99

Refer Local chain concept [in JS]



Local Chain

```
class Student {
    private String name;
    private int rollNum;
    private int marks;
    public Student() {
        SOPln("default constructor");
    }
    public Student(int rollNum, int marks) {
        this(marks);
        this.rollNum = rollNum;
    }
    public Student(int marks) {
        this.marks = marks;
    }
    public Student(String name, int rollNum, int marks) {
        this(rollNum, marks)
        this.name = name;
    }
}
class Alpha {
    public static void main(String args[]) {
        Student jack = new Student("jack", 1, 35);
    }
}
```

OS → JVM

Note • We can call other constructor of the same class using 'this()' keyword
• While using 'this()' it has to be placed at the first of the constructor

Customizing default values: by initializing the instance variables

```
class Student {
    private String name;
    private int marks = 35;
}
```

O/P
name : null
marks : 35

```
class Alpha {
    public static void main(String[] args) {
        Student jack = new Student();
    }
}
```

Command line arguments :-

```
class Sample {
    psvm(String args[]) {
        SOPln("no of arguments:" + args.length);
    }
}
```

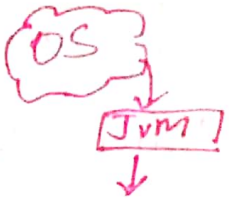
OS → JVM

compilation : javac Sample.java
execution : java Sample Sachin Dravid
no of arguements : 2

args → | Sachin | Dravid |
0 1

def: arguments passed at the time of running the java program
• args are passed from the console
• to check behaviour of the prg • any no of args can be passed

JVM gives control to the main method which accepts strings




```
class Sample{
    public static void main(String[] args){
        SOPln("main accepting Strings");
    }
    public static void main(int[] args){
        SOPln("main accepting integers");
    }
}
```

compilation : javac Sample6.java
execution : java Sample Jack Jill

main accepting Strings

execution : java Sample 120 320

main accepting strings.

In eclipse ▷ → run configuration → command line agruements

Pass the arguements wch you wish to ← Prog arguements ← Virtual args

no of bytes that you want for the prog is passed

Static :
- It is a modifier in java.
- can be used in different elements in java.

↳ variables
↳ blocks
↳ methods
↳ nested class

Static used for variables

```
class Indian {
String name;
int adhaar_no;
static String prime_minister;
```

prime_minister | null

Method area

- Static variables gets the memory allocated only once in the method area at the time of class loading.
- Variables declared as static does not belong to any specific object rather belongs to the entire class, so it is also refered as class variable
- It makes the program memory efficient.

Whenever class is loaded static memory is created.

```
class StaticPrg {
  int x,y;
  static int a,b;
}
```

```
class Alpha {
  psvm (String args[]) {
    StaticPrg sj1 = new StaticPrg();
    StaticPrg sj2 = new StaticPrg();
    SOPln(sj1.x); // 0
    SOPln(sj1.y); // 0
    SOPln(sj1.a); // 0
    SOPln(sj1.b); // 0
    SOPln(sj2.x); // 0
    SOPln(sj2.y) // 0
    SOln(sj2.a) // 0
    SOPln(sj2.b) // 0
```

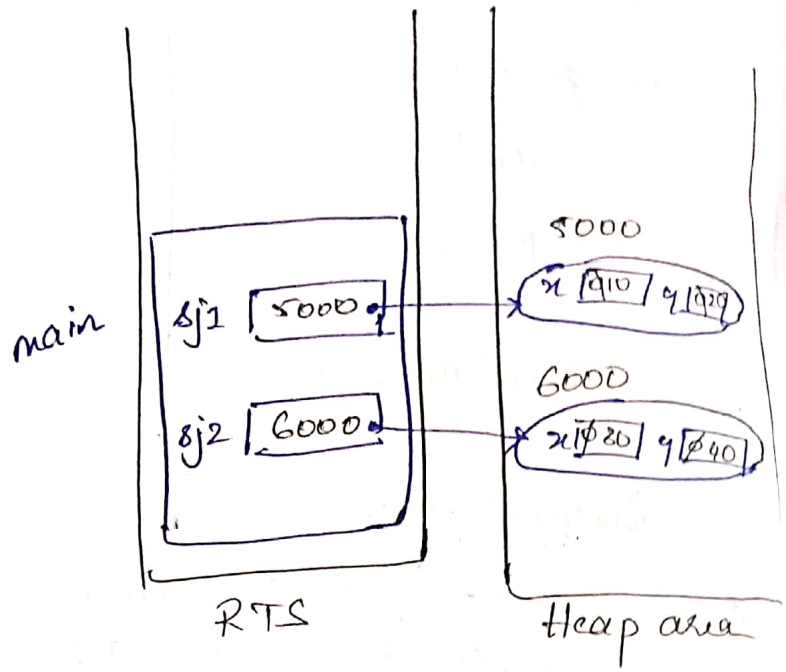
Permanent generation

a | ~~0~~ 777

b | ~~0~~ 888666

Method area

```
sj1.x = 10;
sj1.y = 20;
sj1.a = 999;
sj1.b = 888;
sj2.x = 30;
sj2.y = 40;
sj2.a = 777;
sj2.b = 666;
SOPln(sj1.x);//10
SOPln(sj1.y);//20
SOPln(sj1.a);//777
SOPln(sj1.b);//666
SOPln(sj2.x);//30
SOPln(sj2.y);//40
SOPln(sj2.a);//777
SOPln(sj2.b);//666
```



→ Method Area

Blocks

Whenever class is called

```
{
  // Instance block
}
```

```
static {
  // static block
}
```

Ex

```
class Blocks{
    {
      SOP("Instance block");
    }
  static {
      SOP("static block");
    }
```

- Static block is invoked during class loading.
- Instance block is invoked during the time of object creation.
- Whenever class is loaded first static block is executed & then the main method.
- Any no of static blocks can be created.

```
public class StaticControlFlow {
    static {
        SOPln("static block-1"); ①
    }
    psvm(String args[]){
        SOPln("main method"); ③
    }
    static {
        SOPln("static block-2"); ②
    }
}
```

O/P
Staticblock-1
~~Main method~~
Static block-2
Main Method

class loading → static block is in stack frame → initialization → main method.

```
public class InstanceBlock {
    {
        SOPln("instance-block 1");
    }
    psvm(String args[]){
        SOPln("main method"); ①
    }
    {
        SOPln("instance-block 2");
    }
}
```

O/P
main method

instance block is called when object is created.

```
public class InstanceBlock{
  {
   SOP("Instance Block-1"); ②
  }
  psvm (String[] args){
    SOPln("main method start"); ①
    new InstanceBlock();
    SOPln("main method stop"); ④
  }
  {
   SOPln("Instance block-2"); ③
  }
```

O/P

```
main method start
Instance Block-1
Instance Block-2
main method stop
```

Methods

```
void myMethod() {
}
```

```
static void yourMethod() {
}
```

Instance method is called with the help of object reference

Static method can be called with the help of class name & object reference.

Static variables
Static blocks → class loading
Static methods [main]

Instance variables
Instance blocks → object creation
Instance methods → when method is called
Instance constructors

- Main method will be executed during the time of initialization i.e., after execution of static variables & static blocks.
- Static methods will be executed only when it is called except main method.
- Instance methods are executed only when it is called.

Order of execution during the time of object creation :-

↳ Instance variables
Instance blocks
Constructors
Instance method

Order of execution during the time of class loading :

static variables
static block
main method

```
class Demo {
    {
    SOPln("Instance block");  (2)
    }

  public Demo() {
      SOPln("constructor");  (3)
  }
}

  class Alpha {
      psvm(String args[]) {
      SOPln("main method start");  (1)
      new Demo();
  } SOPln("main method stop");  (4)
```

O/P

main method start
Instance block
Constructor
main method stop

Difference b/w static & instance variable

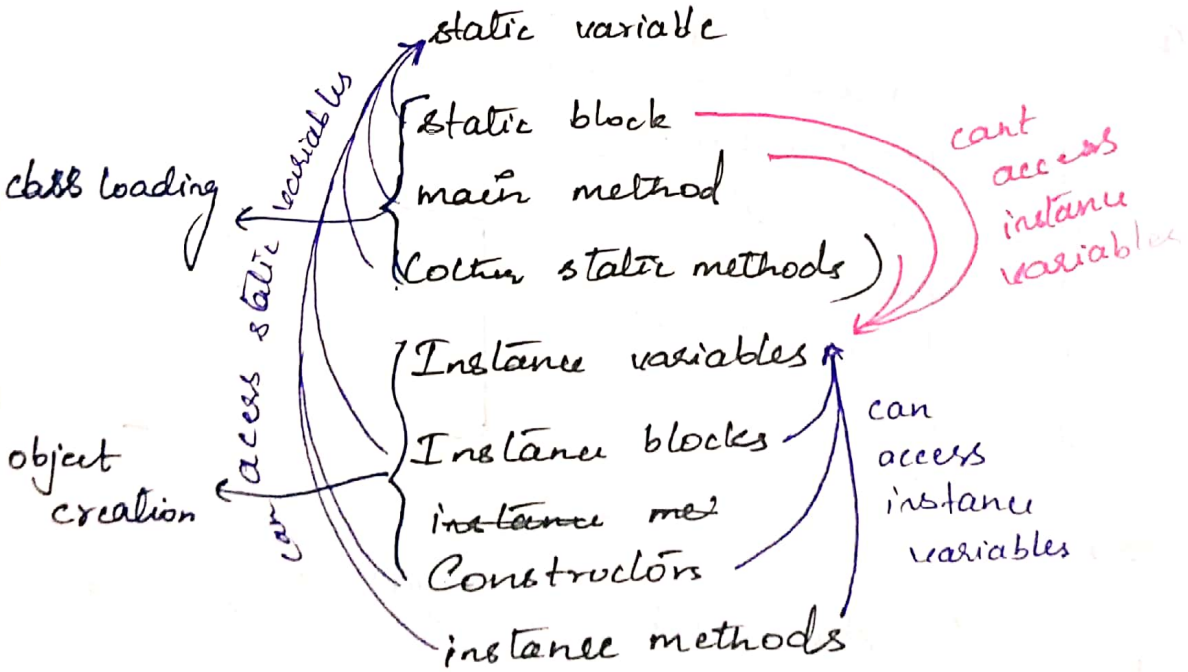
Static	Instance
• static int x;	• int x;
• There will be only one copy of static variable for the whole class	• Every object has one copy of instance variable.
• Static variable is declared inside the class & outside the method	Instance variable is declared outside the method & inside the class
• Static variable can be accessed with the help of class name	• Instance variable can be accessed with the help of object reference.
• Memory is allocated in method area	• Memory is allocated in heap area.
• During class loading memory is allocated	• During object creation memory is allocated.
• Default values are assigned during memory allocation	• Default values are assigned during memory allocation
• Static variables are also called as class variables	• Instance variables are also called fields.
• Can be accessed from static block, instance block, static methods & instance methods	• Can be accessed from instance block & instance method.

Difference b/w static block & instance block

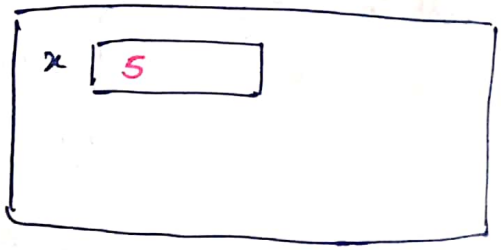
Static block	Instance Block

static {
 =
 }
	{
	 =
	 }

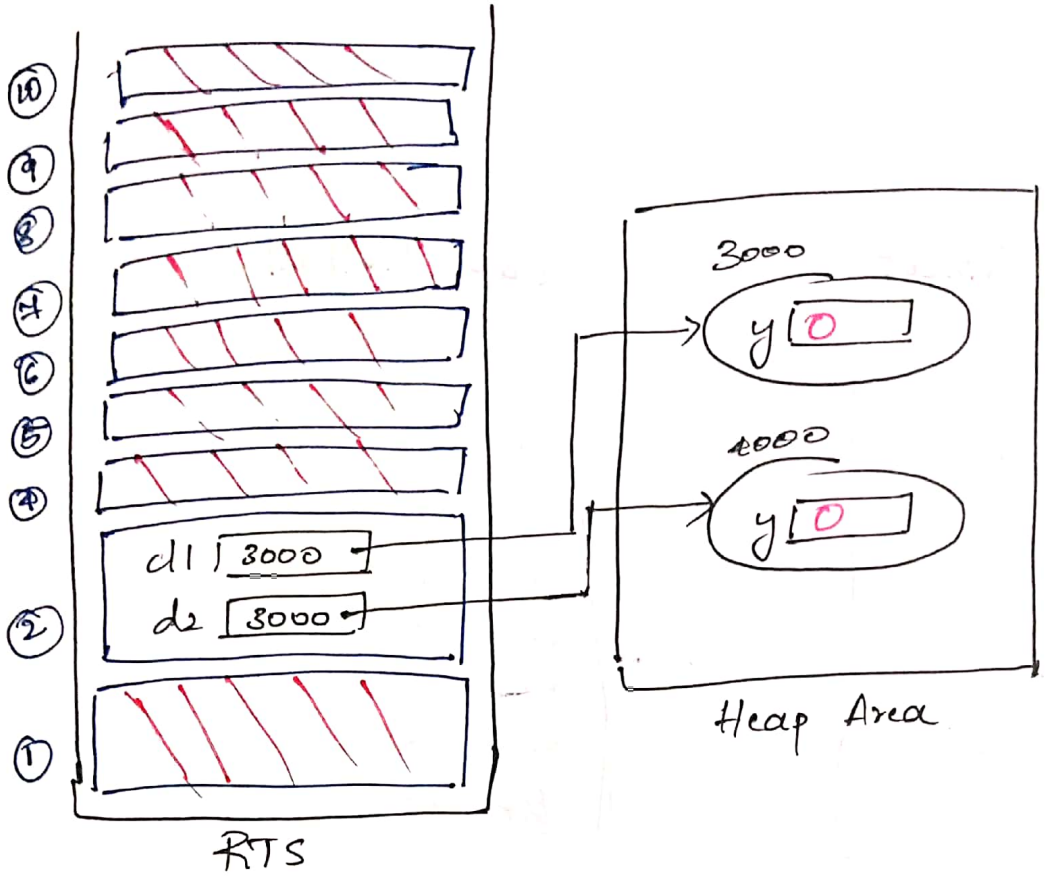
• Static blocks are executed only once.	• These blocks are executed depending upon the objects created.
• Static block is executed during class loading	• Instance block is executed during object creation.
• Multiple static block can be created & the order of execution will be same as the order that has been written.	• Multiple instance blocks can be created & the order of execution will be same as the order that has been written.



```
class Demo {
③  static int x = 5;
    int y;
    static {
④       SOPln("static block : x = "+x);
    }
    {
⑨⑤      SOPln("Instance block : y "+y);
    }
    Demo() {
⑩⑥      SOPln("Constructor");
    }
    static void displayX(){
⑪⑦      SOPln(x);
    }
    void displayY() {
⑫⑧      SOPln(y);
    }
}
class Alpha{
②   psvm(String args[]){
    SOP("main method start");
    Demo d1 = new Demo();
    Demo.displayX();
    d1.displayY();
    Demo d2 = new Demo();
    d2.displayX();
    d2.displayY();
⑬   SOPln("main method stop");
    }
    {SOPln("instance block from Alpha");} // will not get executed
    static {
①       SOPln("static block from Alpha");
    }
}
```



Method Area



Output:

Static block from Alpha.
main method start
static block x = 5
instance block y = 0
constructor
5
0
instance block y=0
constructor
5
0
main method stop

- Static method is used when only static variables are in use & not accessing any instance variable.
- Instance method is used when atleast one instance variable is in use or any number of instance variables being accessed.