* Basic Syntax of Java :

```
class Name {
        ↳ First letter should be capital of every word
    public static void main(String[] args) {
        System.out.println("Abc");
                ↳ print statement
    }
}
```

* Identifiers and literals :

Identifiers :

which will identify themselves in the program uniquely (var)

Literals :

Integer (10, 20, 30 ← base 10 ← decimal)

(1001101 ← base 2 ← binary)

hexa, oct, float, char (a, b, c)

Identifiers :

1. byte
2. short
3. int, long
4. float, double
5. char

In simple the type of any data types are known as literals

e.g Datatype Integer is further divided in
- decimal
- binary
- octal
- heradecimal

* Decimal
  - int a = 10;

* Octal
  - prefix is 0 (zero)

  Range should be 0 - 7 [0011 → 3, 0011 → 9]

  [after that binary no. starts from 2-n so on]

* hexa-decimal

  prefix is 0x (zero and alphabet x)

small (x) and capital (X) either of them can be written Range should be 0 - 9 & A - F

Binary →

prefix 0b (zero & alphabet b)
small case alphabet or capital case alphabet is also valid.

float

always 'f' should be written after the number or else it is considered as double

double

not mandatory to write 'd' by default it is double.

* Control Structures

1. Loops (for, while, do-while)

2. Non loops --> decision control structure (if, else, switch)

If opening & closing is not given in If and If-else statement Then only one sentence get executed written after If & else.

* Double & float are not allowed in switch * boolean not allowed

e.g of switch case

```
switch (input = 20) {
    case 10 : ⇒
        break;
    default : break; }
```

Class and Object

Class :→ It's a collection of Object

Object :→ Template of a class

Anything that relates to has a is called attribute

e.g vehicle has a cost
↑ attribute

Class
- Attribute
- Method

behaviour ← Methods

Always keep get & set method in class to access the class member (Attributes)

local variable are on stack
↑ written inside method (function)

* Method & attribute are on (heap)

on decimal, integer, (binary, hexa, oct)

* underscore is not allowed at the starting and at the ending

* Relating class with other class

```
class 1 {
    =
}
cal class 2 {
    ≡
Main class {
```

class1 (reference variable) = new class();
t

To access it t.set_();

If class 1 is in class 2 then in main function

(reference variable of class1).set class2 (reference variable of other 2nd class)

in class 1

class 2 variable void setclass2 (class 2 va
variable = variable

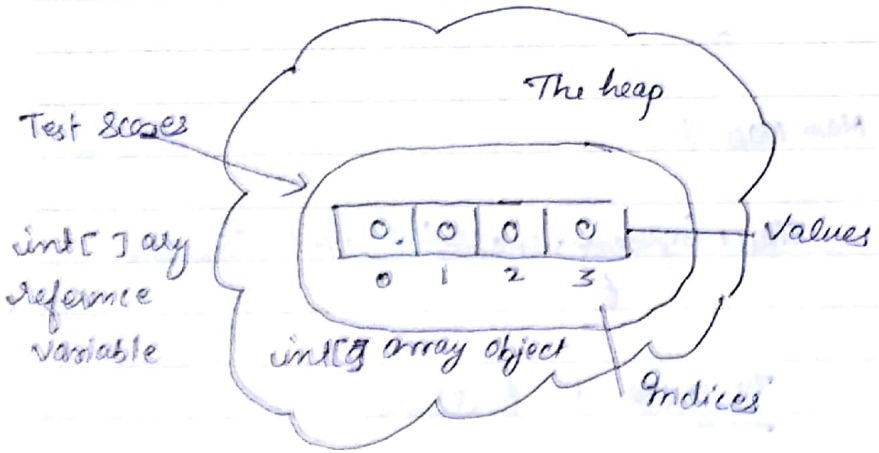
class 2 getclass2 (){
return variable }

* Array

Array are Objects in Java.

reference variable
↕ are on stack & contains value of var in heaps.

Any variable on heap are not garbage they are initally assign value 0



When we try to access the array more than it's (index) size than the error is displayed "Array index out of bound".

length is the attribute of array

To calculate array length use `array name . length`

Size is allowed only after new

construct array on heap with all element as value zero

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

↑ giving size is not allowed

↳ only here it is allowed.

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

```
int a[] = new int [3]{10, 20, 30, 40};
```

(not give size) anonymous ary creation

```
int a[] = new int [4]// less readable by java
```

```
int[] ary = new int [4]
// recomended way
```

Reference value default value is NULL

Datatypes:

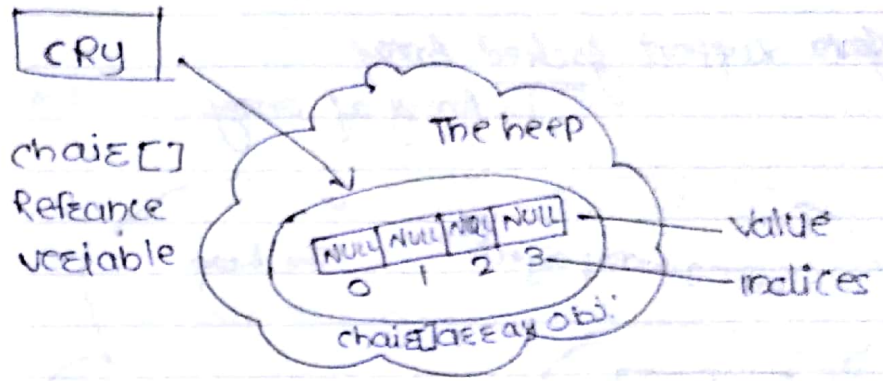
1. Primitive
2. Non-Premitive

1. ex: int, float, double, long, char, byte, short

2. ex: Arrays, string, classes ref(instance)

* Arrays are non-premetive but we can create the array of premitive & non premetive also.

on stack
↓
int[] a = new int[3]
↑ on heap



chaiE C1 = new chaiE();
chaiE[] caEy = neco chaiE[3];
foE (int i=0; i<caEy.length; i++){
caEy[i] = neco chaiE();
// caEy me value

* To create array of Object (Class)

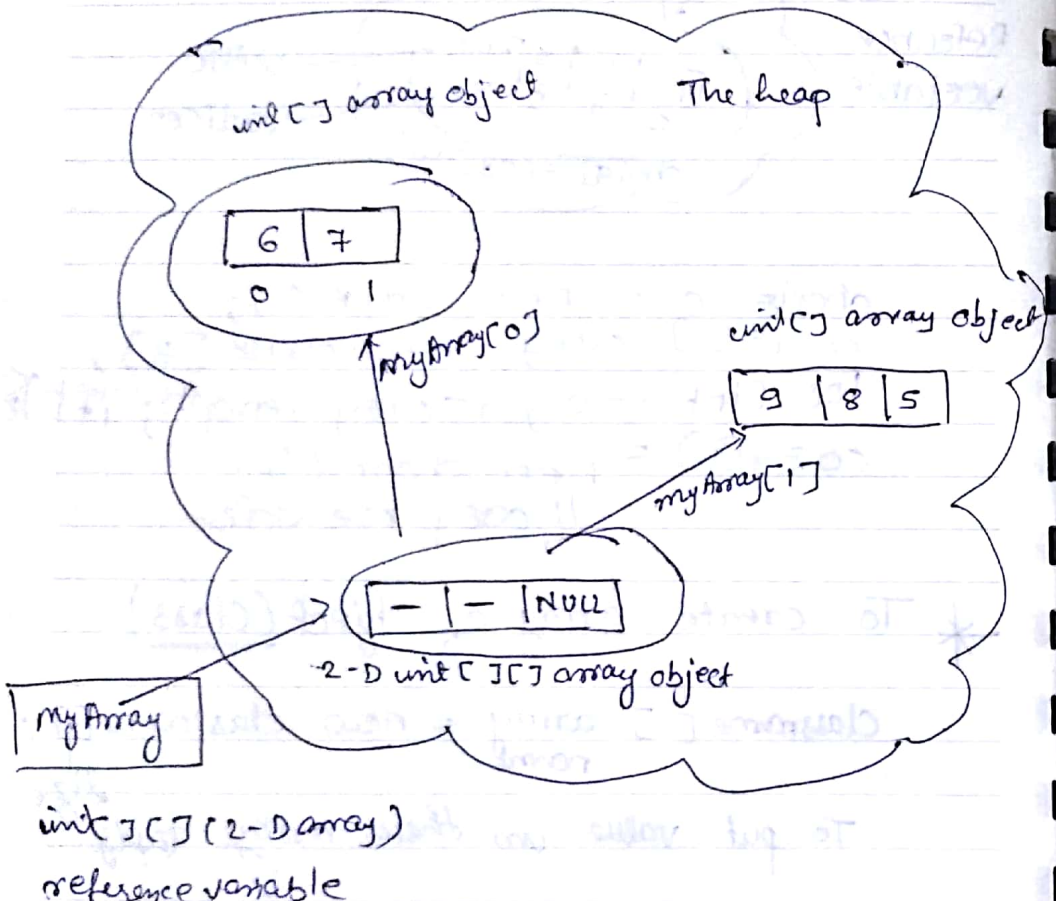
Classname[] array = new Classname[4];
name ↑ size

To put value in there array apply.

array[↑] = new classname(); // initialization
name ↑ index

Java support Jacked Array

↳ Array of array.



int[][] (2-D array)

reference variable

2-D. Array

* Initialization

```
int[][] array = new int[3][];
        name
```

To create new array

```
array[↑] = new datatype[new] of array
name   position
```

e.g. `int[][] arr = new int[3][];`

```
arr[0] = new int[]{2,3};
arr[1] = new int[]{2,4,6};
arr[2] = new int[]{5,4,6,8};

for(int i=0; i<arr.length; i++){

    for(j=0; j<arr[i].length; j++){
    System.out.print(arr[i][j]);
    }

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

* Valid and Invalid Declaration of Array.

```
int[][] ary1 = new int[3][];  // Valid

int[][] ary1 = new int[][3];  // Valid

int[][] ary3 = new int[][] {{10,20}, {20,30}
                            {30,40}}
                            // Valid

int[][] ary5 = ~~new~~ {{10,20}, {30,40}, {50,60}};
```

* giving size ~~be~~ in left side of array declaration is not allowed.

i.e

```
int[][] ary = new int[][] // Invalid
```

↑ Size not allowed

↑ Size must be given here

// Enchance For loop :

```
for(int x : arrayname) {
```

↑ Variable ↑ to print

```
System.out.print(x);   ~~sop(x);~~
}
```

Access Modifier are of two types

1. Access Modifier
2. Non-Access Modifier

1.) Access Modifier

1. public
2. private (Class level access ONLY)
3. protected
4. default (There is no keyword as default, so if the modifier is not applied it means it is in default access.

2.) Non-Access Modifier

1. Abstract
2. Final
3. static
4. Transient (Learn in advance)

* Private :

It cannot be accessed outside of class even if we are having instance of the class in main class.

e.g
```
class Animal {
    // private void eat() {
        System.out.print("Animal");
    }
}
class PrivateDemo {
    public static void main(String[] args)
    Animal a = new Animal();
    a.eat();
} }
```

has access inside class only

* Rule for applying Private

You can apply private modifier to any member of the class but that member once marked private cannot be accessed outside of class.

You can never mark class as private

* Encapsulation :

Hiding the implementation detail from the user, and allows you to use functionality of class

e.g for Driving a car we need the engine to work & we are using accelator & clutch.

Rule : Mark Method as public

Mark Attribute as private

Accelator & clutch are functionality used for running a car engine

* Constructor : 1.> Constructors are used to initialize the members (Attributes) of the class.

2.> Constructors have the same name as that of class name

3.> There should not be any return type to constructor

Constructor syntax : classname() {
}

* Abstraction : Hiding the detail from user

There can be only implicit cast in non-primitive casting and explicit cast will fail at run-time

* Casting :

1. Implicit
2. Explicit

Range

byte (8)
short (16)
int (32)
long (64)
float (32)
double (64)
char (32)

upcast (explicit)

byte
short
char <--> int <--> float
long <--> double

Downcast (Implicit) (auto)

lossy

float -> long not possible

float ——> double.

upcasting → Applied for all in opposite direction from diag.

↳ If we need to convert double to long then

```
double ds = 1111.800;
long le = (long) ds;
```

↑ write data type to convert it

~~char c = 'A';~~

+=, -=, *=, /= ---> Auto cast

```
byte b6 = 45;
b6 = (byte)(b6+5);
b6 += 5;
```

If we need to cast int in byte & short first we need to cast it in byte

```
byte b = (int)(31);
```

e.g 131 bin (8 bit) = 1000 0011

131 bin (32 bit) = 0000 0000 0000 0000 0000 0000 1000 0011

~~Byte (8 bit)~~

Byte (8 bit) = only high order bit get removed [(—) removed]

1000 0011 (MSB represents -ve sign (sign bit)

1 ← - sign bit 000 0011 ← - number.

1000 0011 -> 0111 1101 (--) -125.

* Method Overload :

To overload method we must keep the method name same and parameters of method must vary

In method overload return type may or may not be changed

In method overloading it is autocasted

[No argument method is overload]

e.g
```
void add(int a, int b) {
    return(a+b);
}
double add(double d1, double d2) {
    return(d1+d2);
}
main {
    if system.out.print(10,20);
        (12.2,30);
        (10,2,30,40)   error
```

* Error is given only if there is loosly conversion.
  upcasting

We can change the access modifier from the overloaded method

If the method in the superclass has the restrictive access for e.g private. Method can be overloaded in the same class only.

You can keep the overloaded version of method either in the same class or sub class

A sub class can overload a method only if it is inheriting the older version of method from the super class.

e.g

If private written then no access will be given

```
class Vehicle {
    → private void speed() {
        System.out.println("Speed method of the vehicle");
    }
}
```

These is overload version of speed method

```
class Four_wheeler extends Vehicle {
    void speed(double) {
        System.out.println("Overload speed method");
    }
}
```

* Weaker access privilage.

* Method Overiding

If a method with same signature (name & parameter (arguments)) and return types is present in the subclass. Then the method which is in superclass is said to be overridden.

* Always override is done in sub-class
* override method is possible with Inheritance
* Sub- Once method is said to be over-ridden then subclass instance will call sub-class method. & super class instance will call super-class method.
* A sub-class version of the method should not have weaker class privilage

~~Override~~ Superclass & subclass access modifier should be same

Superclass can't be default & subclass can be other.

Return type of the overriding method can be a subtype of the method which is getting overridden

If private access modifier is use then it is neither over-ridden nor ~~overloded~~ over-loaded.

Superclass method is overridden then give same privilage as of superclass method. public is greater & private is smaller

* Polymorphism :

1:) If object (instance) is showing polymorphic (more than one) behaviour at run time or compile time is also Polymorphism.

2:) Objects are constructed at runtime in Java. So there can be run-time polymorphism.

3:) It is the reference of superclass, holding the instance of sub-class

e.g

```
class Clothes {
    void jeans() {
        System.out.print("Mufti jeans:");
    }
}

Class MyClothes extends clothes {
    void jeans() {
        System.out.println("Local jeans");
    }
}

Class polyDemo {
    public static void main (String[] args) {
        Clothes c = new MyClothes();
        c.jeans
    }
}
```

* use instance of other class e.g Clothes c = new MyClothes(); method must be there other class in these class.

* These method will call

* It is used to use functionality of class *

* Abstract Class: * Method with body is kla concrete method

e.g
```
abstract class House {
    void kitchen() {
    System.out.println("Kitchen");   * Method with no body is kla abstract method
    }
    abstract void hall();
    abstract void bedroom();
}
```

Blue print

Implementation of blue print

```
class MyHouse extends House {
    void hall() {
        System.out.println("Inside hall");
    }
    void bedroom() {
        System.out.println("Inside bedroom");
    }
}
class AbstractDemo {
    public static void main(String[] args) {
        MyHouse mh = new MyHouse();
        mh.hall();
        mh.bedroom();
        mh.kitchen();
    }
}
```

Instance of sub class.

coz inherited.

Only Method and class can be marked abstract.

* Rules:

1.) If you keep a method with no body (non-concrete) then method must be marked abstract

2.) If a method is marked abstract then the class in which it is define must be mark abstract.

3.) In abstract class there can be abstract aswell as non-abstract method. (concrete method

4.) Abstract class can never be instantiated.

House is abstract cannot be ~~instiated~~ instantiated

House h = new House();

5.) Abstract class must be ~~do~~ sub-classed. (make sub class) and it should over-ride all the abstract method from abstract super class. (Abstract class)

6.) Once it is extended then the abstract method should be concrete method

7.) Always ~~give~~ make instance of inherited method (extended method).

e.g abstract class GrandFather {

```
    void land() {
        System.out.println("Grandfather");
    }
    abstract void car();
    abstract void house();
}

abstract class Father extends GrandFather {
    void govtJob() {
        System.out.println("Job");
    }
    void House() {
        System.out.println("New house");
    }
    abstract void plot();
}
class Child extends Father {
    void car {
    }
    void plot() {
    }
}
```

The first concrete subclass (Non-abstract subclass) of the abstract class must override the abstract methods from its abstract superclass.

* Override

e.g

```
class A {
    A get() {
        A a1 = new A();
        return a1;
    }
}
class B extends A {
    A get() {        // legal override.
        B b1 = new B();
        return b1;
    }
}
```

(same return type)

* Rule: Return type of the overriding method can be a subtype of the method which is getting overridden.

* once a class mark as abstract we need to give body in next class

* Constructor

this → keyword

WAP to implement queue with enqueue & dequeue functionality. extend same class to implement pick & display display queue functionality.

Rule:

If you are able to write the constructor then java will not provide any Constructor.

In every constructor Java will insert call to super Constructor (↳ and will call default value of Constructor i.e default constructor)

super() —→ call to super class constructor

[Constructor written by JVM]

* we can write the super constructor by super( )
↑ variable declared inside class

~~By default~~

If we extend the class then we need to have default constructor or else need to write super

e.g

```
class Vehicle {
    int cost;
    String brand;
    Vehicle() {                    ← default Constructor
        cost = 200;
        brand = "Honda";
    }
    Vehicle(int c, string b) {
        cost = c;
        brand = b;
    }
class MyCar extends Vehicle {
    String color
        MyCar() {
            color = "Black";
        }
    MyCar(String c) {
      color = c;
    }
    class ConstructorDemo {
        main() {
        MyCar m1 = new MyCar();
    sop("+m1.cost);
    sop(m1.brand);
    sop(m1.color);
}
```

By default always default constructor of super class is called.

for any type of constructor or default parameterized

If default constructor is not present we need to write a super keyword which will initialize the value

depending on these Constructor are called

* Constructor is used to set values (initialize values)

* Constructor don't have return type

* Class name & Constructor name should be same.

* Sub-class constructor calls to super-class constructor (default call to default super-class constructor) to display the values of variable not declared in sub-class

this ⇒ current class instance

super = super class instance

super() = super class constructor

this() → super class constructor

In super class we can use this or super keyword

Object construct at runtime.

* Final :

Variable marked final cannot be changed ~~once~~ once they are initialized.

* Method overide can't be done when method is marked final

* Method can be overloaded

* If a class marked final we cannot extend it.

e.g

```
Class MyFinal
int x=10;
void setX(){
    x=20;   <-- cannot be changed either write
}
void setX(){
    final int x;
    x=10;
}
```

eg

```
class Generateloop{
final void generate(){
System.out.println("loop generate
}}
class Generatedatesgen extends
Generateloop{
void generate(){
}
```

cannot be ~~overide~~ override

sudo apt-get update
sudo apt-get install default-jdk

Check ($javac -version)

Constructor:

Use to construct the Object.

* Without Inheritance to call class within class

To call class that is in other class

// Kitchen k;
// k = new kitchen();

class
extend → is a relation
class → attribute → has a

Class Table
↓
Class Kitchen
↓
Class House

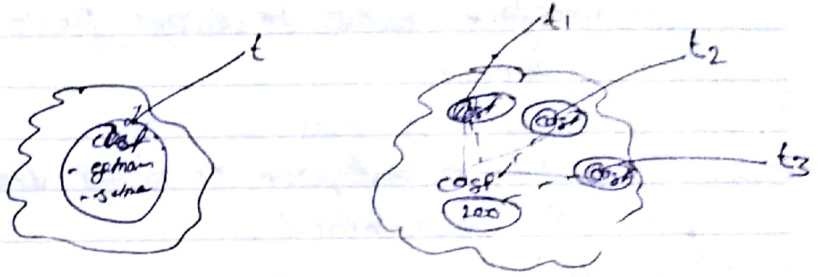
h1.k.t.setCost()

Table t1 = new Table();

Table t1 = new Table();
t1.setCost(1000);

* Static : → methods } can be made static
→ Attributes }

class cannot be made static
Constructor cannot be made static.



Before static

After marked static

attributes & methods

Class can be accessed by multiple reference variable in order to make access by one way mark attribute or method as static once marked static we can access it by class name also. in order to access by class name [MyTable.cost=30;].

* Non-static variable cannot be referenced from a static context.

* Static member have default value '0'

A class must implement the interface for. eg(Class A implements B => B is the interface and A is a class

* Interface :- Rules:
  1:> 100% abstract superclass
  100% abstract superclass

  2:> interface will contain only abstract methods. no method with body is allowed.

  3:) interface methods are always public ~~abstract~~ abstract

  4:) Variables in interface are always static and final (they are constant).

  ~~5:> Only~~

  class ↔ Interface ⇒ Implement

  We can always implement the interface

Interface Follow → Is a relation

class ↔ class is a relation

class ↔ Attribute has a relation

* In interface variable are by default static Final
* Interface instance cannot be created.

To print the variable marked in interface or class we can use either class name or interface name.

* Final class
  1:> Final class can never be extended
  2:> final variable cannot change its value
  3:) you can never override the final method.

* Abstract class

  1. Abstract class must be subclassed
  2. You can never create the instance of a abstract class.
  3 Abstract class may or may not contain the abstract methods.
  4: Abstract class may also contain its own concrete (Non-abstract) methods.

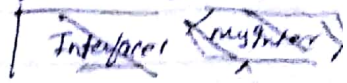
(NOTE: You cannot mark a class as abstract & final both)

You can never mark a method of a class private & abstract both.

* Private · Attribute & Method can mark private in the class

Attribute mark private cannot be used again

Method



A class must implement the interface for eg (Class A implements B ⇒ B is the interface and A is a class

* Interface :- Rules:
  - 100% abstract superclass
  1. 100% abstract superclass
  2. interface will contain only abstract methods. no method with body is allowed
  3. interface methods are always public abstract
  4. Variables in interface are always static and final (they are constant).

5. ~~Only~~

Class ↔ Interface ⇒ Implement

We can always implement the interface

Interface Follow → Is a relation

Class ↔ Class is a relation

Class ↔ Attribute has a relation

* In interface variable are by defaul static Final
* Interface instance cannot be created.

To print the variable marked in interface or class we can use either class name or interface name.

* Final class
  1. Final class can never be extended
  2. final variable cannot change its value
  3. you can never override the final method.

* Abstract class

1. Abstract class must be subclassed
2. You can never create the instance of a abstract class.
3. Abstract class may or may not contain the abstract methods.
4. Abstract class may also contain its own concrete (Non-abstract) methods.

(NOTE: You cannot mark a class as abstract & final both).

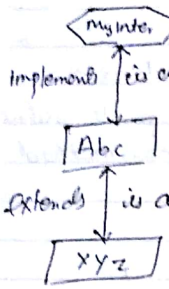
You can never mark a method of a class private & abstract both.

* Private: Attribute & Method can mark private in the class

Attribute mark private cannot be used again

Method → → → →

Interface! myInter



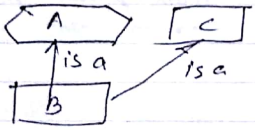
Code

```
interface MyInter {
    void abc();
}
abstract class Abc implements
MyInter {
}
Class Xyz extends Abc {
public void abc() {
System.out.println
("Inside abc");
}
}
```

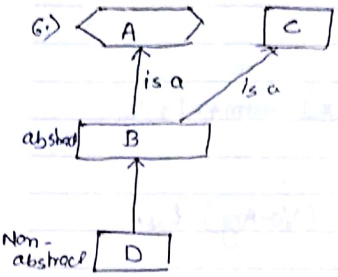
One class cannot have multiple super class

Multiple subclass can have one super class

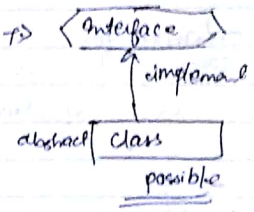
We can have multiple super class of class with help of inheritance.



6.)



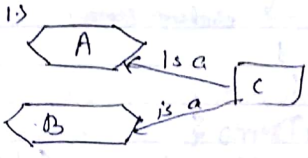
7.)



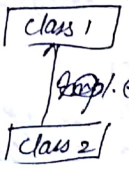
A class can implement multiple interface

One class can extends only one single class.

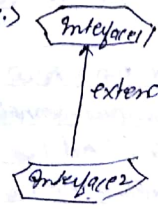
1.) Level 1:



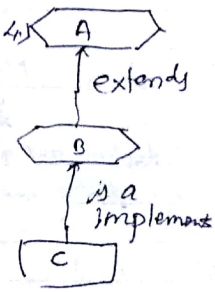
2.)



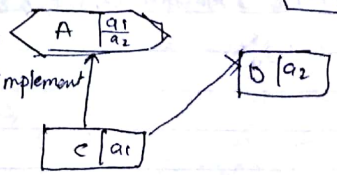
3.)



4.)



5.)



Enhanced for :

```
for(int i : x) sum+=i;
```

* Variable Argument (VarArg) list.

```
Class MyAdd {

    int add(int... x) {
        int sum = 0;
        for (int i : x) sum+=i;
        return sum;
    }
}

class VarArgDemo {
    public static void main(String[] args){

        Myadd ma = new MyAdd();
        System.out.println("Add is:"+ma.add();
System.out.println("Add is" : +ma.add(1,5,6));
 System.out.println("Add is "+ma.add(1,5,6,7,8,9,0);
    }
}
```

Var-Arg :

1. Var-arg (or) (Variable Argument list)

1) method accepts any number of arguments.

for ex: int add(int .....x)

↑

This method will accept any no. of integers. Min = 1

and the x (is variable arguments) will be treated as array inside the method.

2) Var-arg also accepts array.

3) you can replace the argument inside the main method as ⇒ public static void main(String... arg)

↑

allowed :-) :)

4) Var-Arg must be the last argument in the method if there are many arguments

e.g double add (double d1, double d2, int...x)

5) You can only write VarArg method single if you want it first eg double add (int...x)

e.g double add (double d, double d1, int...x) {

```
    int sum = 0
    for(int i : x) sum+=i;
    return (sum+d+d1);
```

add integers →

```
double add( double d, double d1, int... x) {
    int sum = 0
    for(int i :x) sum+= i;
    return( sum+d+d1);
}
```

5) var-arg cannot be used in place of return type

```
e.g int... getR etC() {
        int[] any= {1,2,3,4};
        return{any};
    }
```

Invalid method declaration ...
Var Arg can not be return type

7) Can be used in Constructor

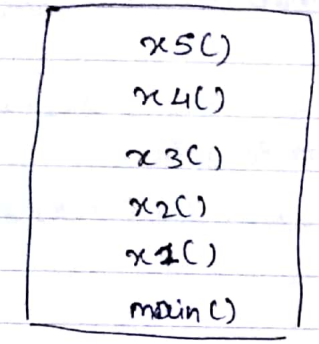
```
Myprog( int...val) {
    x = val[0];
    y = val[1];
```

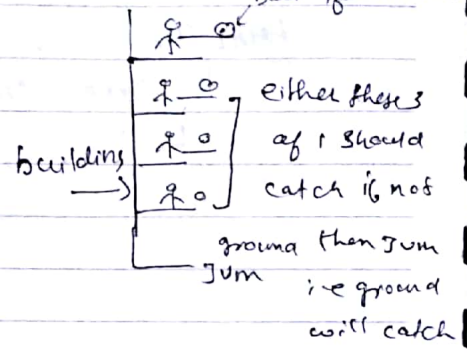
Indexof(n) method
↳ return integer value
↳ It will only print the first occurance of index.

CharAt(4)
↳ return character value
↳ .

* Exception

Eg MyexceptDemo2-java





- Call Stack

(1) AI → example Sophia Robot

Strong AI → makes strong claim

- Weak AI → ~~mates~~ predict some feature that are resembeling to human intelligence and can be incorporated to computer to make useful tool

Machine learning :

Angular

Template → HTML portion

scope → It represent model of app

It contain field that

Controller takes empty scope object as a parameter and add it to function that will be later exposed to the User via view.

Ordinary Year = 1 odd day

Leap year = 2 odd day

Days

0	1	2	3	4	5	6
S	M	T	W	Th	Fr	Sat

Month

[J FM] A M J J A S O N D

(0 3 3)(6 1 4)(6 2 5) 0 3 5

Year

1600 - 1699 → 6

1700 - 1799 → 4

1800 → 1899 → 2

1900 - 1999
↳ 0

~~2000~~ ~~1900 - 1999~~
↳ 0

2000 → 2099
↳ 6

26 Jan 1947 which day

1. Last two digit → 47
2. Divide by 4 → 11 (quotient)
3. Take the date → 26
4. Take no. of Month → 0 (here it's Jan)
5. Take no. of year → 0 (here it's year 1900-199)
6. Add from (1 to 5) 84

→ Divide by 7 and write the remainder

$$7 \overline{)\,84\,(12} \quad \frac{84}{00}$$

0 correspond to day = Sunday

Ans

If leap year then subtract 1 from remainder (29 Feb then only)

Today is Monday After 30 days it will be

$$7 \overline{)\,30\,(4} \quad \frac{28}{2}$$

Monday + 2

Wednesday

Jan 4 2016 falls on Monday

on what Jan 4 2017 falls

→ leap year

Monday + 2

Jan 4 2017 → Wed

100	6
200	4
300	2
400	0

When discount(1) is given & gain(1) gna

and need to find the gain(2)
use the marked % as the m

Profit % = m% − discount − $\frac{m\% \times dis}{100}$

Profit % = marked % − Discount % − $\frac{\text{marked \%} \times d}{100}$

~~C.P x Marked % = S.P~~

Marked Price − Discount = S.P

Discount % is always on the marked price

10/10/18

Simple Interest become n time than

$RT = 100(m-1)$

Rate, Time, how many time

Thrice itself in 3 years in compound interest
how many years will be 9 times

(3) → 3 years

$(3)^2 = 9$ $\quad 3 \times 2 = 6$

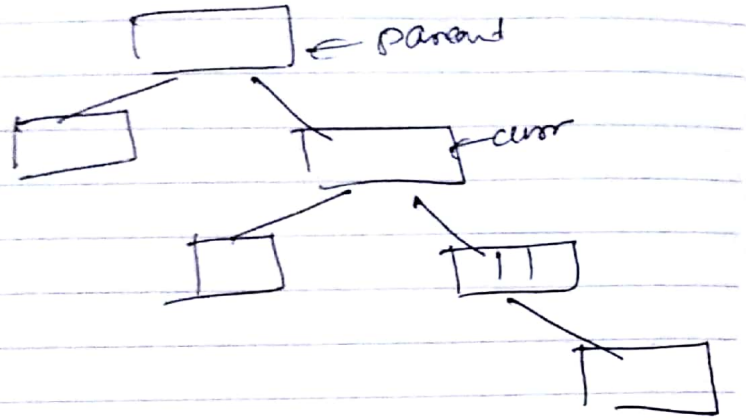
Compound Interest become n time itself

$\left((m)^{1/t} - 1\right) \times 100 = R$

At what Rate at C.I will a sum of money become 9/4 times in 2 years

$\left(\frac{9}{4}\right)^{\frac{1}{2}} \times 100 = R$

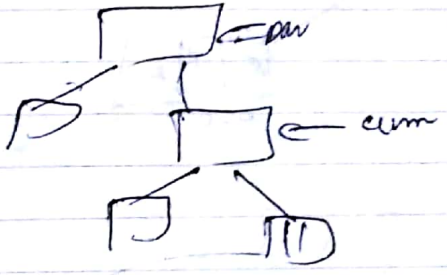
R = 50



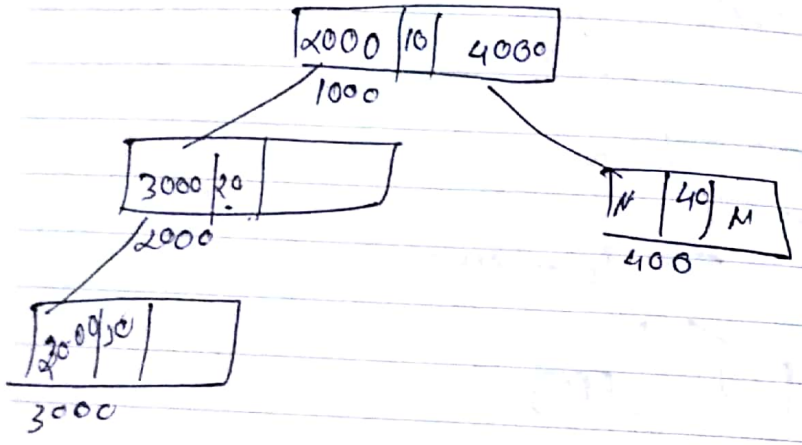
same code above line

```
if (t1 -> right != NULL && t1->left == NULL)
{
    curr -> data = t1 -> data
    curr -> right = t1 -> right
    t1 -> right = NULL
    free (t1);
}
}
```

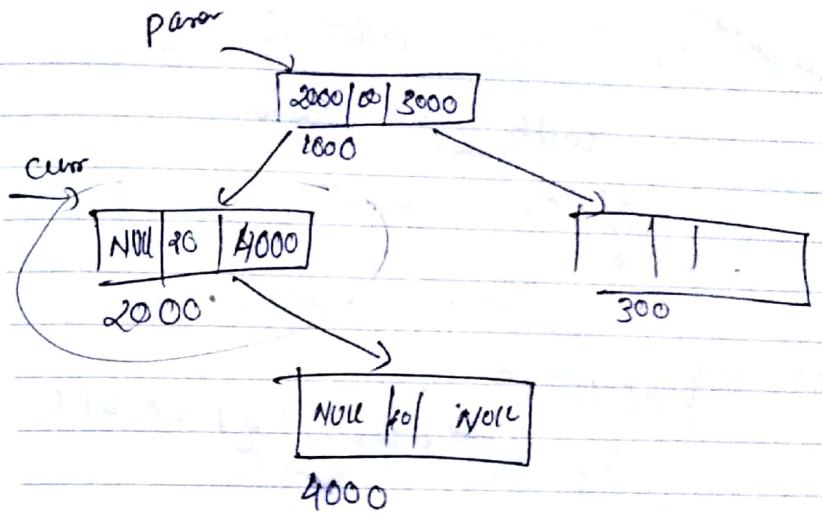
Deletion of node in Binary Search tree (having 2 children)




```
if (curr -> left != NULL && curr -> right
    != NULL) {

    struct node *t1, *t2;
    t1 = curr -> right;

    if (t1 -> left == NULL && t1 -> right ==
    NULL) {
        curr -> data = t1 -> data
        curr -> right = NULL
        free (t1);
    }
}
```




```
if (curr->left != NULL) {
    if (curr == parent->left) {
        parent->left = curr->left;
    }
}
curr->left = NULL;
free(curr);
```




```
if (curr->right != NULL) {
    if (curr == parent->left) {
        parent->left = curr->right;
    }
}
curr->right = NULL;
free(curr);
```

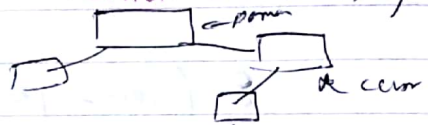
early & late bindings

Delet node from BST

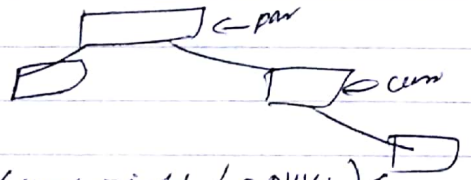
with no children

```
if (curr == parent -> right ){
        parent -> right = null
} else {
        parent -> left = NULL
}
```

Delete having 1 child. connected to parent to right




```
if ( curr->left != NULL ) {
    if ( curr == parent ->right ) {
        parent->right = curr -> left ;
    } }
    curr -> left = NULL
    free(curr);
```




```
if (curr->right != NULL) {
    if (curr == parent->right) {
        parent -> right = curr -> right ;
} }
    curr->right = NULL;
    free(curr);
```

Insertion in BST

```
Void Insert (int d)
    struct node *t, *p
    t = (struct node *) malloc ( sizeof( struct node) )
    t -> data = d;
    t -> left = NULL , t -> right = NULL;
    p = root;
if (root == NULL ){
    root = t;
} else {
    struct node * current;
    curr = root;
    while ( curr ) { parent = curr;
        if ( t -> data > curr -> data ) {
            curr = curr -> right
        } else {
            curr = curr -> left
    }
    if ( t -> data > parent -> data ) {
        parent -> right = t;
    } else {
        parent -> left = t;
    }
}
```

Adding after specified node

```
void After() {
    struct node *temp,*p;
    int loc, len, i=1;
    printf("Enter location");
    scanf("%d", &loc);
    len = length();
    if(loc > len){
        printf("Invalid location");
        printf("List contain only %d node", len);
    }else{
    temp = (struct node*) malloc(sizeof(struct node));
    printf("Enter node data");
    scanf("%d", &temp->data);
    temp->left = NULL;
    temp->right = NULL;
    while(i<loc) {
        p = root;
        p = p->right;
        i++;
    }
    temp->right = p->right;
    p->right->left = temp;
    temp->left = p
    p->right = temp;
}
```

Find the length of Double list

```
int length(){
    struct node *temp = root;
    int count = 0;
    while(temp != NULL) {
        count++;
        temp = temp->right
    }
    return count;
}
```

Display all Element in Singly list

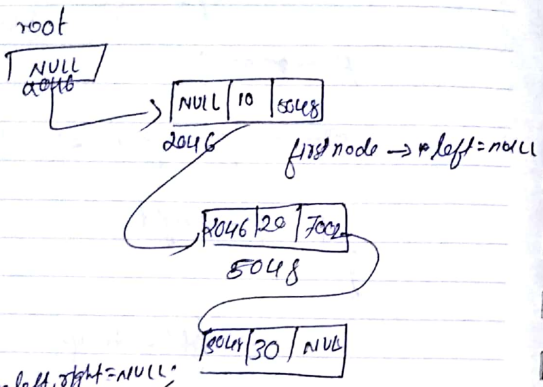
```
void display(){ if(temp==NULL) printf("no node");
    struct node *temp = root;
    while(temp != NULL){
        printf("%d", temp->data);
        temp = temp->right
    }
}
```

Adding node at Beginning in Circular

```
addBegin(){
    struct node *temp
    temp = (struct node*)malloc(sizeof(struct node));
    pf("Enter Node Data");
    sf("%d", &temp->data);
    temp->left = NULL;
    temp->right = NULL;
    if(root == NULL){
        root = temp;
    }else{
    // struct node *p;
    temp->right = root;
    root->left = temp;
    root = temp;
}
}
```

```
else{
    struct node *p;
    p = root;
    while(p->right != NULL){
        p = p->right;
    }
    p->right = temp;
    temp->left = p;
}
}
```

Double Linked list




```
struct node {
    int data;
    struct node *left, right=NULL;
} struct node *root = NULL
```

* Append node in Double linked list
↳ add node at last

```
void append() {                    root = NULL
    struct node *temp;
temp = (struct node*) malloc (sizeof (struct node));
    temp→ left = NULL;
    temp→ right = NULL;
if (root == NULL) {
    root = temp;
} else {
```

Sort a Singly Linked List.

// Bubble Sort :

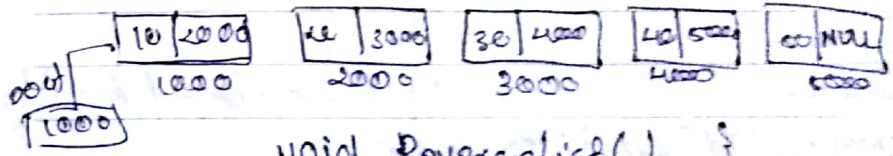
```
For(i=0; i<n-1; i++) {
For(j=0; i<n-1; j++) {
    temp in ab:
if (a[j] > a[j+1]) {
    temp = a[j]
    a[j] = a[j+1]
    a[j+1] = temp;
}
}
```

```
temp = p -> data
p -> data = q -> data
q -> data = temp;

i++;
j--;

p = p -> link;
q = root;

}
```

Reverse all the elements of singly linked list

```
i = 0 - for(i=0;i<n;i++)
j = n-1
while(i<j) {
    temp=a[i];
    a[i] = a[j];
    a[j] = temp;
    i++;
    j--;
}
```

Reverse all element of array




```
void ReverseList() {
    int i, j, len;
len = length();
    i = 0; struct node *p, *q;
    j = len-1;  p = root; q = root;
    while(i<j) {
        k=0;
    while(k<j) {
q = q -> link;
        k++;
    }
```

Queue

Insert use rear variable

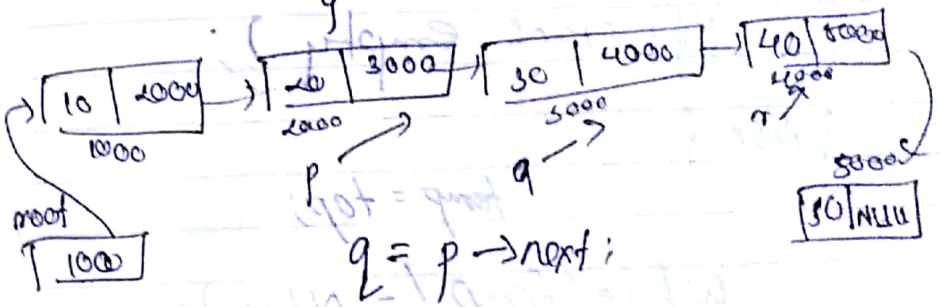
Delete use Front

```
Int Front=0, Rear=0;
void Insert() {
    if (rear==CAPACITY) {
        pf("Queue Full");
    }else{
        int ele
        sf("%d",&ele);
        queue[rear] = ele;
        rear++;
    }
}
```

```
void traverse() {
    if(Front==Rear){
        pf("Queue is full");
    }else{
        pf("Queue element are")
        for(i=Front; i<Rear; i++)
            pf("%d", queue[i]);
    }
}
```

Adjacent Node Using Singly Linked list

```
int i=1;
p = root;
while(i<loc-1) {
    p = p->next
    i++;
}
```




```
q = p->next;
r = q->next;
q->next = r->next;
r->next = q;
p->next = r;
```

Traverse/Display all element

```
void traverse() {
    struct node *temp;
    if (top == NULL)
        Pf(stack empty);
    } else {
        temp = top;
        while(temp != NULL) {
            Pf("%d", temp->data);
            temp = temp->link;
        }
    }
}
```

Pop/deletion

```
void pop() {
    struct node *temp;
    temp = top;
    if (top == NULL) {
        Pf("No node present");
    } else {
        temp=top;
        Pf("Element %d", temp->data);
        top = temp->link
        temp->link= NULL
        free(temp);
    }
}
```

* Stack Implementation using Singly Linked List

(1) Push/ Insertion

```
struct node {
    int data;
    struct node *link;
};

struct node *top = NULL;

void push() {
    struct node *temp;
    temp = (struct node*) malloc (sizeof(struct node));
    pf("Enter node data");
    sf("%d", &temp->data);
    temp->link = top;
    top = temp;
}
```

// Deleting from middle/ last

```
else {
    struct node *p = root;
    struct node *q;
    int i = 1;
    while (i < loc-1) {
        p = p->link
        i++;
    }
    q = p->link
    p->link = q->link;
    q->link = NULL;
    free(q);
```

Stack Implementation Using Singly Linked list

Delete node from Singly Linked list

```
void delete() {
    struct node *temp;
    int loc;
    Pf("Enter loc to delete");
    Sf("%d", &loc);
    if (loc > length()) {
        Pf("Invalid location");
    // Delete from Start
    else if (loc == 1) {
        temp = root;
        root = temp->link;
        temp->link = NULL;
        free(temp);
```

POP

```
if (top == -1)
    printf("Stack Empty");
else
    top = top - 1;
```

show

```
For(i=top; i=>0; i--)
    Pf(stack[i]);
If (top == -1)
    Pf(Stack Empty)
```

stack

↳ works on LIFO

```
stack = (int *) calloc (capacity, sizeof(int));
```

Creation of stack

```
void push(int ele) {
    if (isfull()) {
        top> top == CAPACITY - 1
        pf("stack is full");
        return 1;
    else
        return 0;
    }
else { top++;
    stack[top] = ele;
    pf("Element Pushed");
```

Sort List

```
void bsortlist (node *head) {
    struct node *i, *j, temp;
    for(i=head; i!=NULL; i=i->next) {
        i->next!=NULL
        for(j=i->next; j!=null; j=j->next
        if (i->data > j->data) {
            temp = i->data;
            i->data = j->data;
            j->data = temp;
        }
    }
}
```

Add at last

```
struct node{
    int data;
    struct node *link;
}; struct node * root = NULL;

void append(){
    struct node *temp;
    temp = (struct node*) malloc(sizeof(struct node));
    pf("Enter node data")
    sf("%d", &temp->data);
    temp->link = NULL;
    if (root == NULL){
        root = temp;
    } else {
        struct node *p;
        p = root;
        while(p->link != NULL) {
            p = p->link;
        }
        p->link = temp;
    }
```

Display list

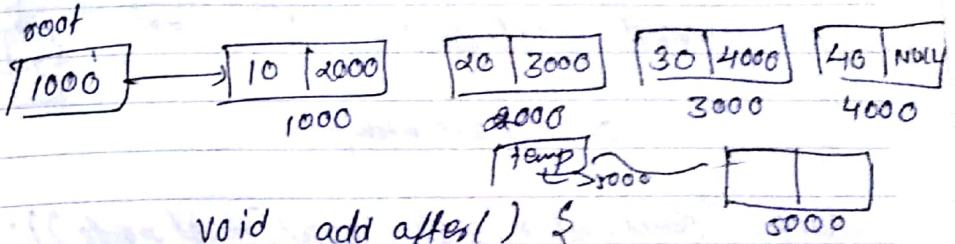
```
void display() {
    struct node * temp;
    temp = root;
    if(temp == NULL){
        pf(No node);
    } else {
        while(temp != NULL)
            printf("%d -> ", temp->data)
        temp = temp->link;
```

loc [3] i [1] p [1000]

```
else {
    int i = 1;
    struct node *p;

    p = root;

    while(i<loc) {

        p = p->link;
        i++;
    }

// creating node
temp = (struct node*)malloc(sizeof(struct node));

    read node data & store null

    temp->link = p->link

    p->link = temp;
}
```

Add After

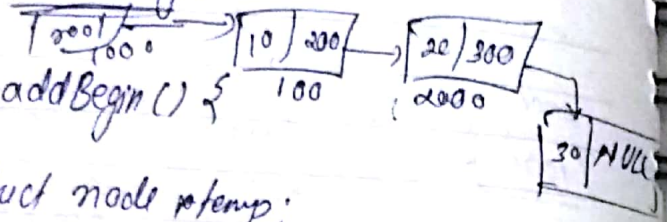
~~Display All element of linked list~~



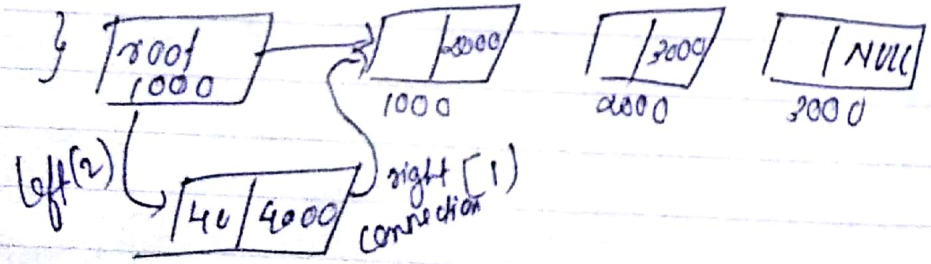

```
void add after() {

    struct node *temp;

    printf("Enter Location");

    scanf("%d", &loc);

    len = len();  → Discuss Previously

    if(loc > len) {

        printf("Invalid Location");
        printf("Currently node is having %d" node, len);
```

loc

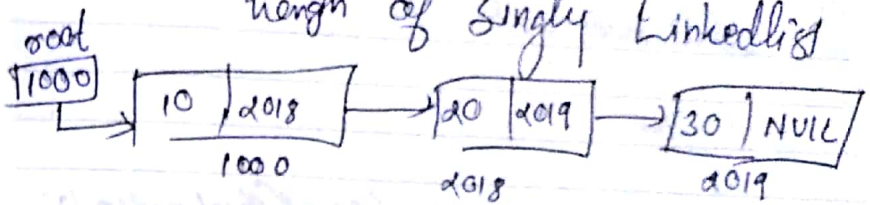
☆ Add At Begin



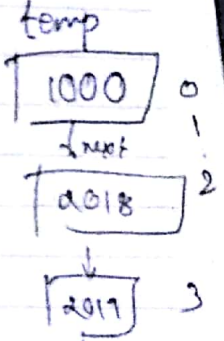

```
void addBegin() {
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    printf("Enter node data");
    scanf("%d", &temp->data);
    temp->link = NULL;
    if (root == NULL) {
                    // if its there is no node
        root = temp;
    } else {
        temp->link = root.
        root = temp;
    }
}
```



length of Singly Linkedlist




```
int length() {
    int count = 0;
    struct node *temp;
    temp = root
    while (temp != null)
        temp = count++;
    temp = temp->link;
    return count;
}
```



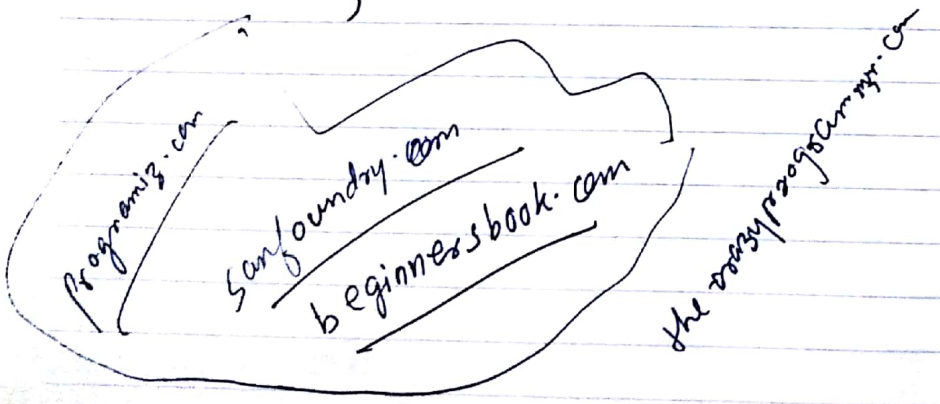
```
struct node *ptr
int item;
ptr = (struct node*) malloc (sizeof(struct node*));
if (ptr==NULL)
```

cbasicprogram.blogspot.com

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

```
For (i=1; i<=5; i++) {
    For (j=5; j>i; j--) {
        printf (" ")
    }
    For (j=1; j<=i; j++) {
        printf ("%d", j)
    }
    printf ("\n")
}
```

Javaconceptoftheday.com



Select ~~*~~ from match, venue, team

where m.venue = v.id ;
m.tname = t.

Select *.