Date
25/10/20.

Introduction to JAVA

→ c++ is an object oriented language.

→ James Gosling is a one member of the c++ language.

→ James Gosling, Mike, Ed Frank, Chris Warth, Patrick, these 5 members formed a team in 1986 had developed a new project. The name of the team is Green Team and the project name is green project. They started worked on it. When they going to implement the work from 'c' they taken Syntax & Sementics and from c++ they taken object oriented concepts, and then they developed the programing language JAVA. At first they had decided the name greentalk and they started to save the file as .gt. But later it was renamed as OAK in 1992. OAK is version. The first commercial version of OAK is JAVA in 1995.

→ JAVA is a coffee first produced in Island of Indonesia.

~~→ 20th Jan 2010~~ Initially Java developed from the company Sunmar Sun micro system, But from 20th Jan 2010 Java is handover oracle Coorparation.

26/10/21

Buzz words (features) of JAVA.

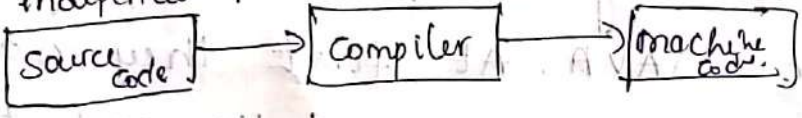
→ Buzz words are the most familiar things.

- Simple
- Secure → Virus programs – Virtually information Resource under Sinc
  Malicious
- Robust (strong) – JAVA is a strictly typed language.
- Portable – Easy to carry. we can run very easily.
- Object oriented. – It's a real time entity / real world compon & have its own component.
- Architectural Neutral –
- Multi threading
- Dynamic
- High performance
- Platform independent

Dynamic: when compiler resolves method call binding during execution of program, such process known as Dynamic or late binding.

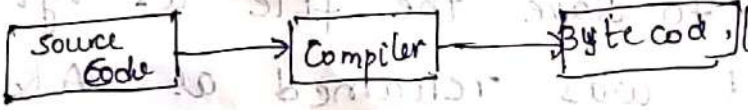
→ Why JAVA is Neutral Architectural? we can run any operating system. & its platform is independent.

In 'C'

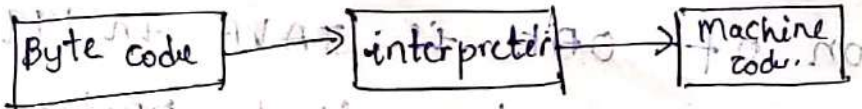


In JAVA.

Step: (1)



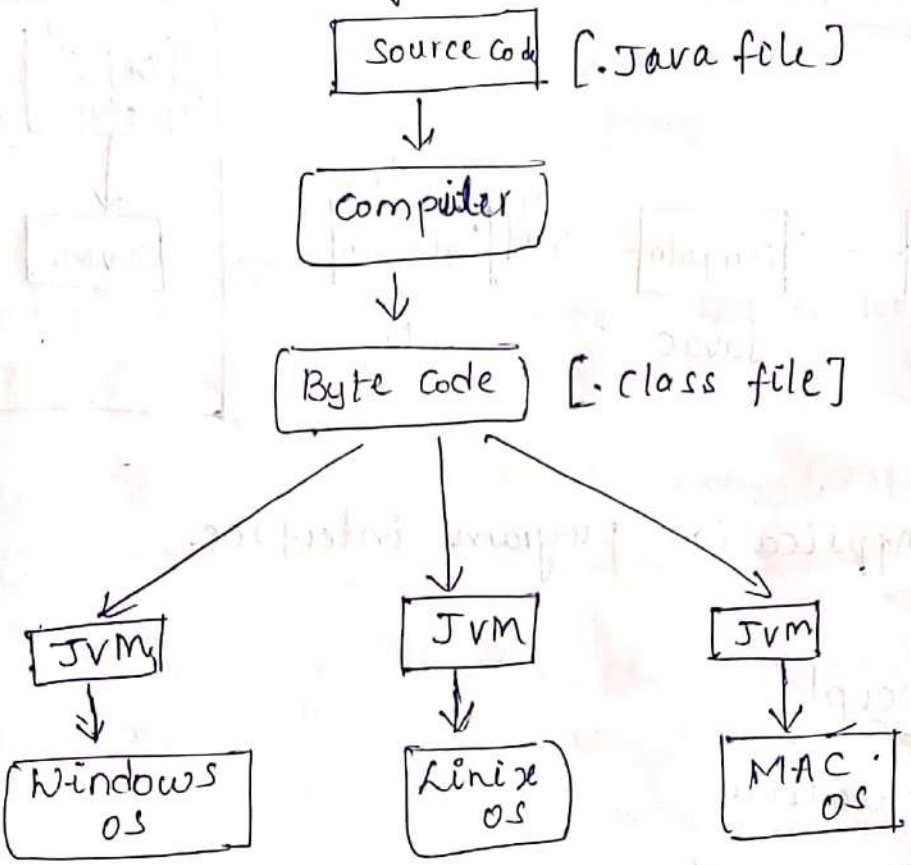
Step: (2)



Robust: Java is robust as it is capable of handling run-time errors, Supports automatic garbage collection & exception handling & avoids explicit pointer concept.

Architectural Neutral.

"write once, run anywhere, anytime forever".



S/w is designed without regard to target platform.

Multithreading

A thread is a light weight process

Executing more than one thread is called multithreading. (at a time)

Dynamic & High performance

Ex: Internet

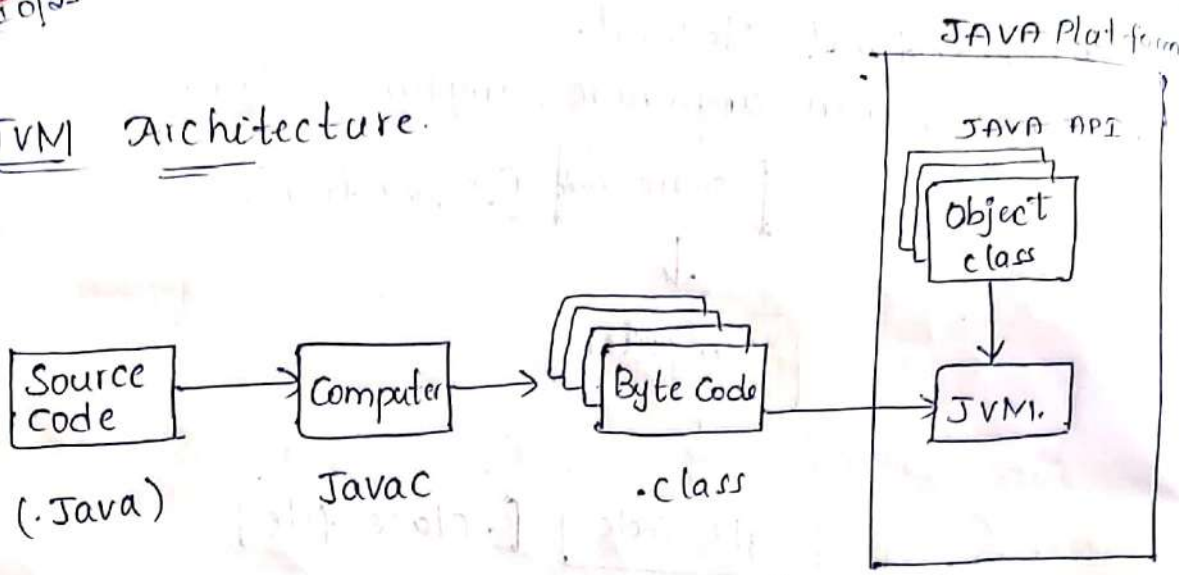
Extensibility & Reusability.

J2ME (code of mobile).

~~Java Architecture~~.

High performance :- Java uses Just-in-time compiler to enable high performance. Just-in-time compiler is a program that turns java bytecode which is a program that contains instructions must be interpreted.

JVM Architecture.



API - Application program interface.

Oops principles

1. Encapsulation
2. Inheritance &
3. Polymorphisam.

Program:- combination of code & data

program = code + data.

1. Encapsulation: It is a process of mechanism that which binds code and data wh with in a protective vapour that which provides Security from outside the vapour.

Ex: capsules, Bike gear system & so on

Note:

Program: Combination of code & data.

2. Inheritance:- Aquiring the properties of one object by another object is called an inheritance.

Ex:- Humans, Mamals etc,.

3. Polymorphisam:

Poly + morphisam are two greek words stands for many + forms.

"One interface, Multiple methods"

One define, multiple implementation

→ Implimenting one stack for any no. of datatypes.

Secure

1. Java prgms run inside a virtual machine which is known as a sandbox.
2. Java does not support explicit pointer.
3. Byte-code verifier checks the code fragments for illegal code that can violate access right to object..

Date
30/10/21

Java Tokens: These are small elements of a program identified by Java compiler.

1. Comments:
2. Identifiers
3. Literals / constants
4. Keywords
5. operators.

1 Comments

The main purpose of using comments of Java is to identify the piece of code in a lengthy program.

→ we can done in this two ways.

(i) Single line comments.
(ii) Multi line comments.

(i). Single line Comments.

Syntax:-

// comment. ...

Ex:- //print stmt
system.out.println("___");

(ii) Multiline Comments

Syntax:- /* comment 1.
comment 2
.....n */.

Note: Compiler ↑ simply ignores the Comments.

2. Identifiers

1. Identifier can be declared with the combination of letters (A a ..... Z, a ..... z), digits (0 to 9) and two special characters underscore (_) and dollar ($).

2. Digit never be first character, violation leads to Compile time error.

Ex:- ~~i=10;~~

int i=10; → valid

int i,j → valid

int ij=10 → valid

int sum=10 → valid

int k7 =10; → valid

int 7k =10; → invalid

int 7_k=10; → invalid.

int _k1 =10; → valid.

int k$ =10; → valid

int $k =10; → valid.

3. There is no restriction for length of an Identifier.

Ex:- int sum=10; length = 3.

int a, b, ... z =10; length = 26.

4. Ex: int A=10, a=20;

Here, it treated as two different declarations. Because Java is a case sensitive language.

5. Reserved words are not allowed to use as an identifier.

Ex: int if=10; not allowed.

int else =5; not allowed

int string =5; allowed

int thread =6; allowed.

6) In-Built classes names can use as as identifier

Ex: int string =5; allowed

int thread =6; allowed.

3. Literals / constants:

our JavaC supports 5 types of literals.

They are:

1. Integral literal
2. floating-point literal
3. Boolean literal
4. char literal.
5. String literal.

Date: 21/01/23.

1. Integral Literal:

Binary

If we want to define the literal as binary the value must start with (0b/0B).

Ex: int t = 0b10;

O/P: 2.

Decimal

int i=10;

O/P: 10.

Octal:- (0/0).

If we want to define the literal as octal the value must start with (0/0).

Ex: int i = 025;

O/P: 21.

int i = 081;

O/P: CT ∈ (compile time Error).

Becuz it is out of range.

Hexadecimal:-

If want want to define the literal as Hexa ducimal the value must start with zero X (0X, 0x).

Ex: int i=0X52;

O/P: 82.

int i = 0XAB;

O/P :- 171.

Floating point literals.

10 ⇒ 1010.

16 ⇒ 10000

33 ⇒ 100001.

5581 ⇒ 1000100 1011.

To write large numbers ~~in~~ to like.

3 8520u26

9 85231ou5 589 10568 2.

we have floating point representation,

There is a notation

$$\pm M \times B^{\pm e}$$

Where, m = mantissa.

B = Base.

e = exponent.

Note:- Integral literals in Java (Binary, octal, hexa) are supported by only 4 primitive datatypes.

Byte, short, int, long.

Boolean literal

It has two literals. They are True and False 'C' does not support boolean whereas Java support boolean.

char Literals

which represents a character. It can be represented with a letter or digit or Special character and a space with single quotes (' ')

Ex: 'k' 'L' ' ' '#' '*'.

String literal:

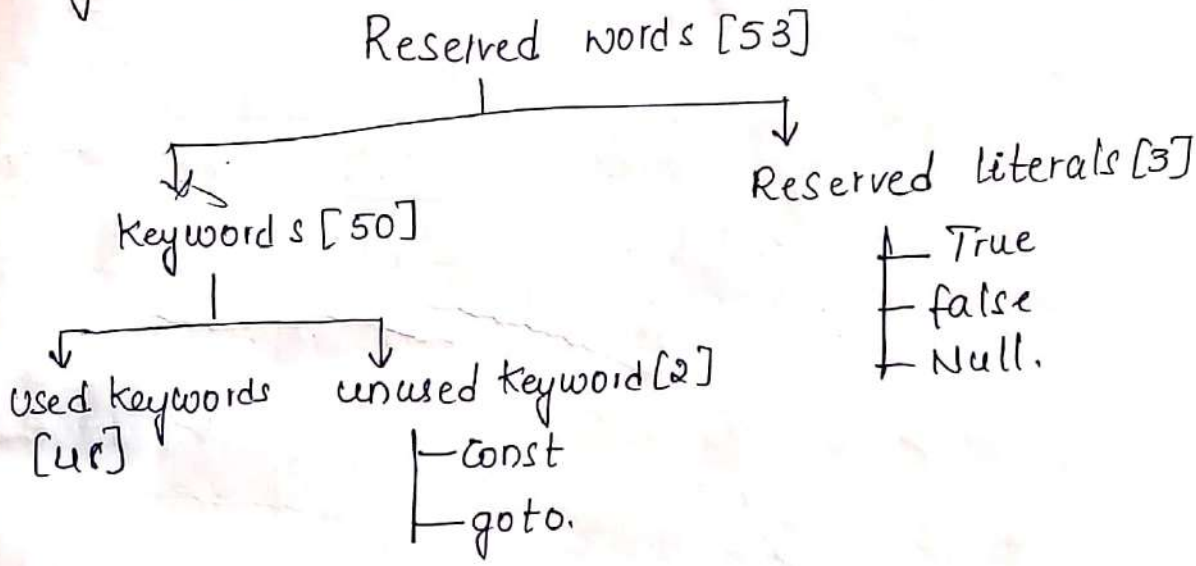
A collection of characters that forms a word or a statement which can be represented with the help of double quotes(" ").

Ex: "STUPID"

"You so cute"

1

4. Keywords



Access Specifier:- Public, Protected, Private, default.

Data types :- Byte, short, int, long, double, float, char, Boolean.

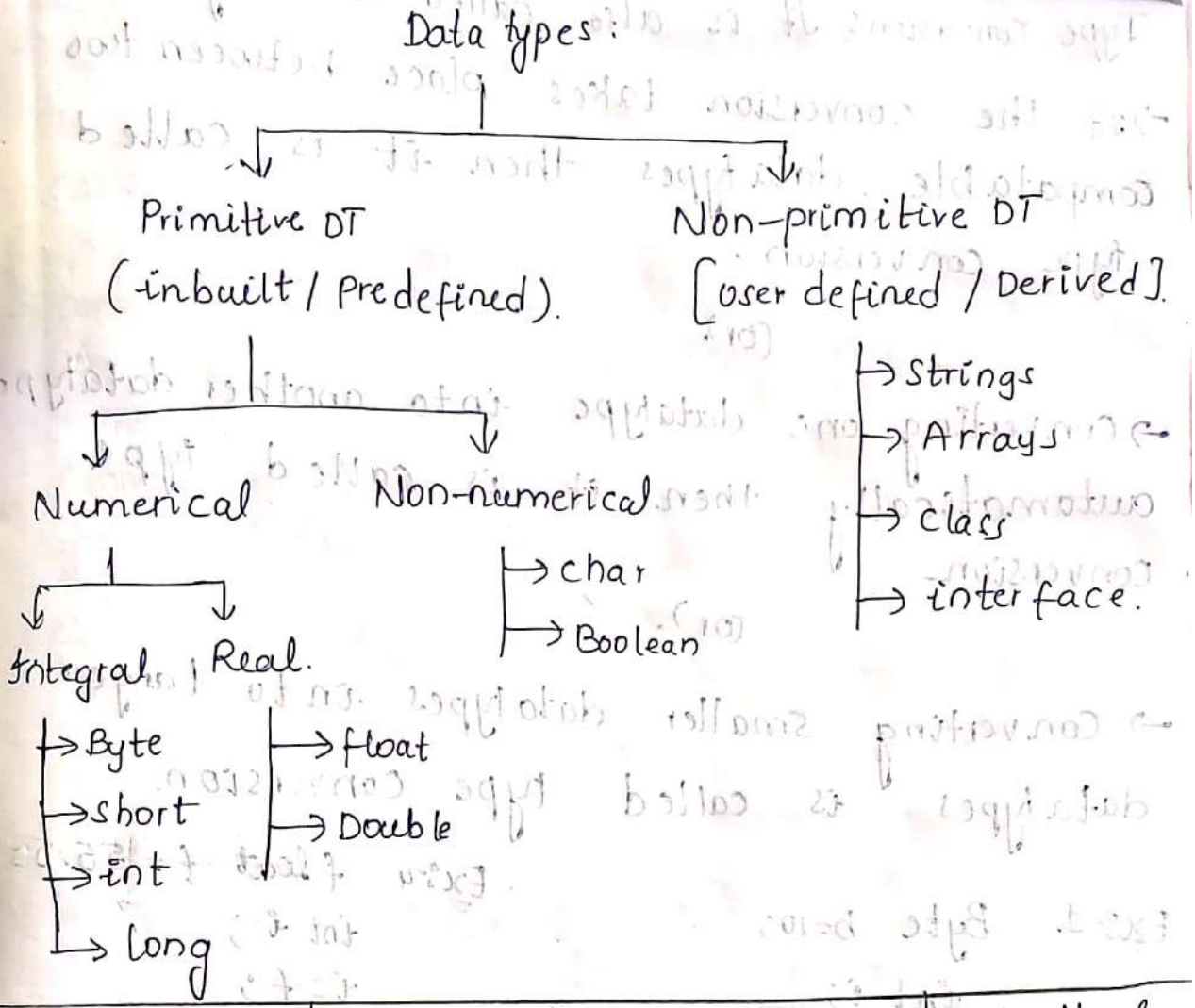
class :- class , import , package , interface.

Exceptions :- Try, finally , throw , Assert , throws , getch.

Control Statement :- if , else, switch , case, default, for, while , dowhile , break , continue, return.

Date
05/11/21

Data types.



S.No	Data type	Size	Range	Default value
1.	Byte	1 Byte	-2^7 to $+2^7-1$	0
2.	Short	2 Bytes	-2^{15} to $+2^{15}-1$	0
3.	int	4 Bytes	-2^{31} to $+2^{31}-1$	0
4.	long	8 Bytes	-2^{63} to $+2^{63}-1$	0
5.	Float	4 Bytes	$-3.4e^{38}$ to $+34e^{38}$	0.0
6.	Double	8 Bytes	$-1.7e^{308}$ to $+1.7e^{308}$	0.0
7.	Char	2 Bytes	0 to 65535	Space '\u0000'
8.	Boolean	N.A	N.A.	False

06/11/21

Type Conversion & Casting.

Type Conversion: It is also called widening conversion

→ If the conversion takes place between two compatable data types then it is called type conversion.

(or)

→ Converting one datatype into another datatype automatically then it is called type conversion

(or).

→ Converting smaller data types into larger data types is called type conversion.

Ex:-1
```
Byte b=10;
int i;
i=b;
```
1 byte → 4 byte.

o/p:- 10.

Ex:-2
```
short S=10;
byte b;
b=S;
```
o/p:- CTE

Ex:-3
```
int i=10;
float f;
f=i;
```
o/p: 10.0

Ex:-4
```
float f=55.5;
int i;
i=f;
```
o/p:- CTE

Ex:-5
```
char ch='A';
int i;
i=ch;
```
o/p:- ASCII value of A will be the output.

Ex:-6:
```
char ch='A';
Short s;
s=ch;
```
o/p: CTE.

Type casting.

It is also called narrowing conversion.

→ If the conversion takes place between two incompatable datatypes then it is called type casting.

(or):

→ converting one datatype to another datatype manually then it is called type casting.

(or)

→ Converting larger datatype into smaller data type is called type casting.

Ex:
```
int i = 20;
byte b;
b = (byte) i;
```

Byte → short.

char → int → long → float → double.

L → R ⇒ widening

R → L ⇒ narrowing.

Note: The Boolean datatype never compatable with any other datatype.

9/11/21

Variable

A variable is a basic unit of storage in a program, that which are independent to the program.

The main purpose of variable to track change the values to solve a given problem.

Declaring a variable.

Syntax: type identifier1 [= value1], identifier2 [= value2], identifier3 [= value3];.....;

Ex: int i;
int i=10;

Note:- In Java we can declare a variable with initialization and without initialization.

Static initialization.

If the value 0 is assigned to a variable at compile time then it is called static initialization of a variable.

Ex: int i=10;
int i=10, j=20;

Dynamic intialization.

If the value is assigned to a variable at run time then it is called dynamic initialization of a variable.

Ex:- k= Math.sqrt(100).0001;

Scope of a variable.

The scope of a variable defines the possibility of accessing a variable, basically the possibility of accessing a variable is with in a block.

Lifetime of a variable.

Lifetime of a variable refers how long that a variable is available in memory.

```
class scope
{
  P.S.Vm (String args[])
  {
      int X = 100;
      SOP(X);
      if (X==100)
        {
          int XY=200;
          SOP(X);
          SOP(Y);
        }
      SOP(Y);
  }
}
```

O/P:-

It is a CTE. Bcz it prints when X=100 & XY=200 but it block in the int XY=

```
Class Scope
{
    P.S.V.M (String args[])
    {
        int x=100;
        SOP(x);
        if(x==100)
        {
            int x=200;
            SOP(x);
            SOP(x);
        }
    SOP(x);
    }
}
```

output: Compile time Errors.

Operators

1. Arthimetic operators. (Binary).
2. Increement or decrement operators.
3. Logical operators.
4. Assignment operators.
5. Conditional operators. (Ternary).
6. Comparison / Relational operators
7. shift operators.
8. short - circuit operators.

Basically the operators are categorized in to three types:

1. Unary
2. Binary
3. Ternary.

1. Arithmetic operator.

Ex: byte a=10, b=20, c;
c = a+b;
S.O.P("c");

output: compile time error.

Reason: For arithmetic operations the java compiler follows

max (int, Type1, type2) to store the resultant value.

2. Increment & Decreement operators.

Increment operator done in two ways pre increment & post-increment.

Increment operator is represented with '++'

Ex: int i=10, j;

1. j = i++; o/p: i=11; j=10.
2. j = ++i; o/p:- i=11; j=11.

In pre-increment or post increment the self variable always increment (+i), but when

we try to assign the incremented value to another variable. then difference comes in to picture.

→ Decrement operator can be carried out in two ways

1. Pre-decrement
2. Post-decrement

→ Decrement operator represented by '--'

Ex: int i=10, j;

j = i--; o/p: i=9, j=10

j = --i; o/p: i=9, j=9

```
int i;
i=5++;
sop(i);
```

Note: Inc/dec can't be applied to a value, we can apply to only a variable.

4. Assignment Operator

=, +=, -=, *=, /=, %=, ==.

Ex: i = i+1.

⇒ i += 1;

Ex: 10==10; -T
10.0==10 -F
97=='a' -T

5. Conditional operator

Syntax: Condition ? Expression 1 : Expression 2;

(F) (T) executed.

executed

Note: Difference b/w if-else and conditional operator is

If we have only one stmt to execute T or F of a given condition then its better to refer conditional operator otherwise if-else stmts.

6. comparsion / Relational operators.

The written type of relational operator is Boolean.

Ex: int a=10, b=20, c=30;
a<b ⇒ True
b>c ⇒ False.

a<b<c
o/p: CTE
1p<20<30 = True<30

```
float f= 5.6;
f = f++;
S.O.P(f);
```

```
float d= 33.44f;
d = d++;
S.O.P(d);
o/p: 33.44
```

7. Shift Operator.

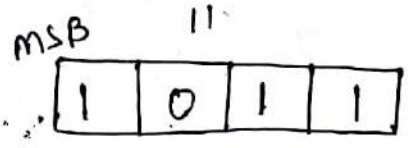
Here we have 2 types of shift operators.

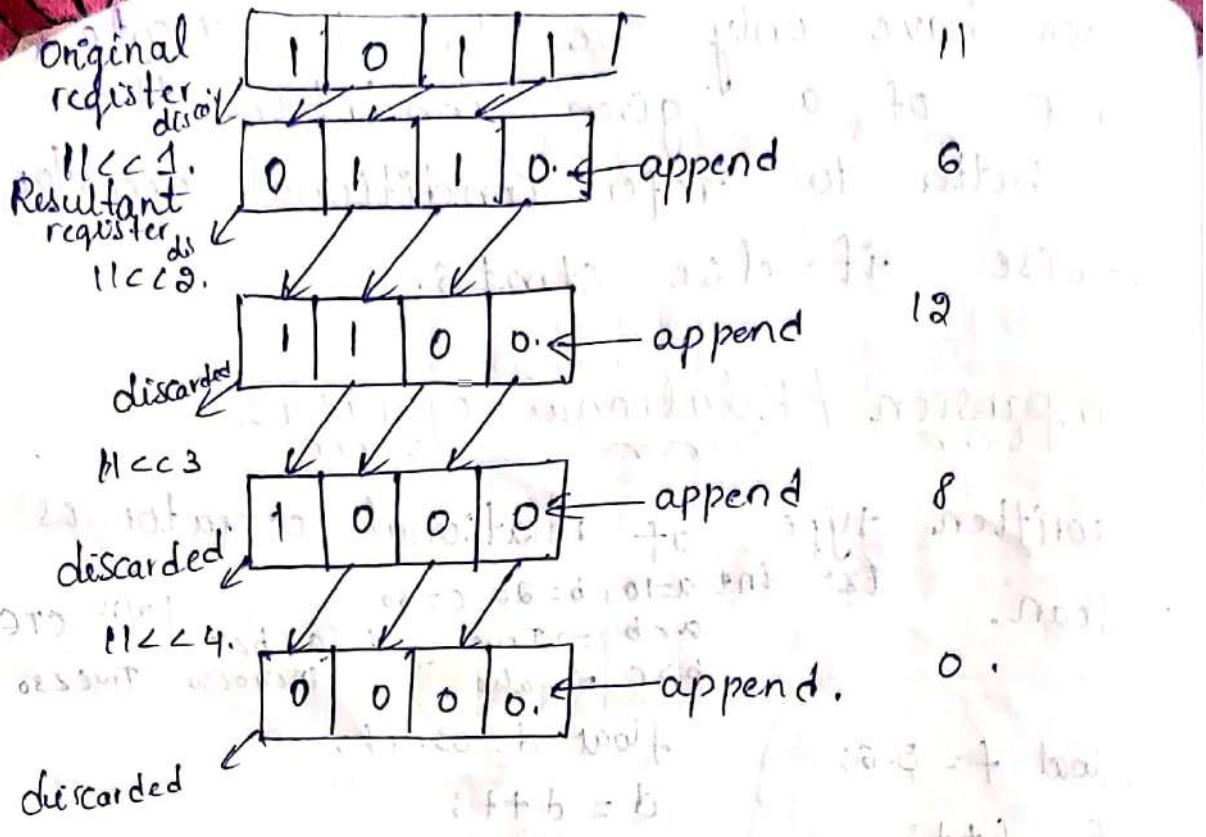
1. left shift.
2. Right shift.

1. Left shift

Left shift operator can be represented as $a<<n$ where a is magnitud
n = no. of times to shift.

Ex:- $11<<1$.





1. In left shift the most significant bit of original register is discarded.

2. (least) LSB of resultant register is appended with '0' (zero).

3. For every shift operation the value may increase or decrease.

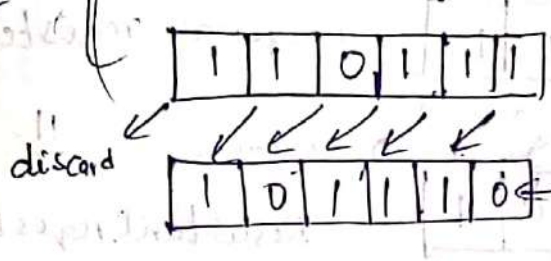
4. The maximum no. of possible shifts is 'n' where n is size of the register.

5. After 'n' no. of shifts the result becomes zero.

Ex:- 55 << 2 :

a << n

Formula: $a \times 2^n$



$a = 55$

$n = 2$.

$\Rightarrow 55 \times 2^2$

55×4

$= 220$.

$a \times 2^n$

Ex: 258 << 3

258×2^3

258×8.

$= 2064$

2. Right shift:

$a >> n$.

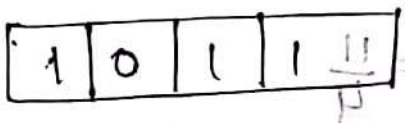
Formula $\Rightarrow$ $\frac{a}{2^n}$

Right shift operator can be represented as $a >> n$; where a = magnitude

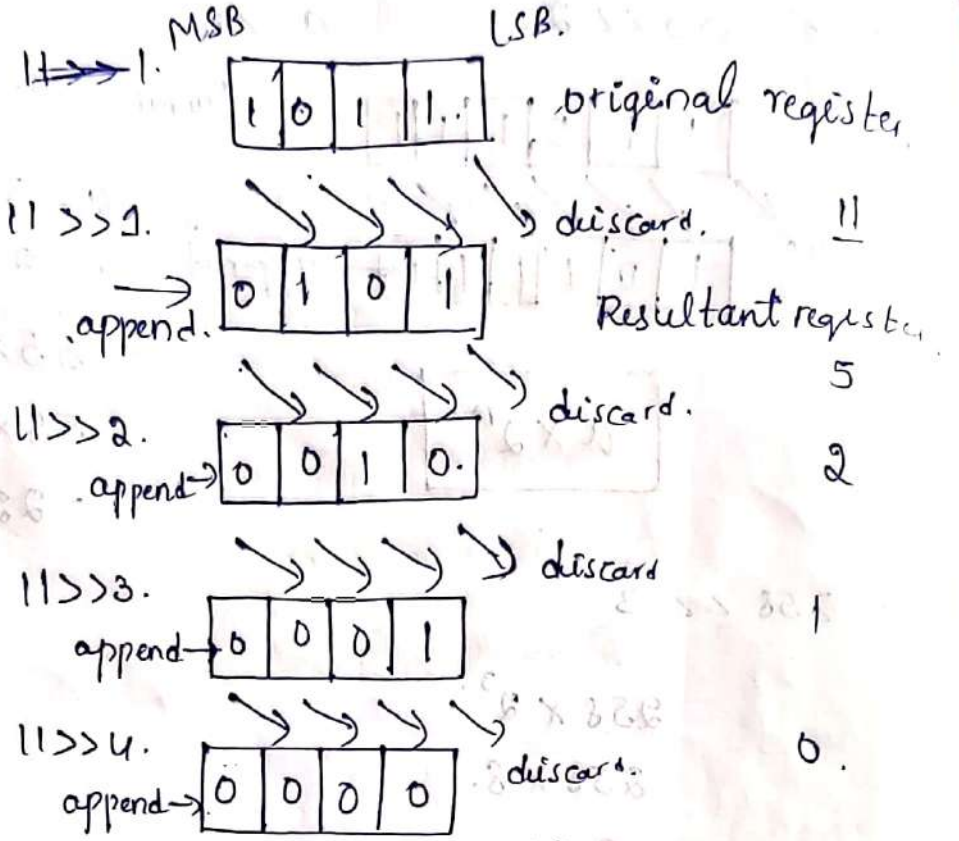
n = no. of times to shift.

$a >> n$ $a/2^n$.

Ex:- 11 >> 1



Ex:-



1. In Right shift the least Significant bit of original register is discarded.

2. MSB of resultant register is appended with zero '0'.

3. For every shift operation the value decreases.

4. The maximum possible no. of shifts is 'n' where 'n' is the size of the register.

5. After 'n' no. of shifts the result becomes zero.

Ex: $11 >> 2$.

$a >> n = \frac{a}{2^n}$.

$= \frac{11}{2^2} = \frac{11}{4}$

$= 2.75$.

10/11/23

8. Short-circuit operators.

1. Short circuit && operator (→ &&).

```
if (Exp1 & exp2)
{
    // code
}
```

```
if (Exp1 && exp2)
{
    // code
}
```

Exp1 & Exp2	
T	T (exuates)
F	T

Exp1 && Exp2.	
T	F not executes

and 2nd stmt also does not executes.

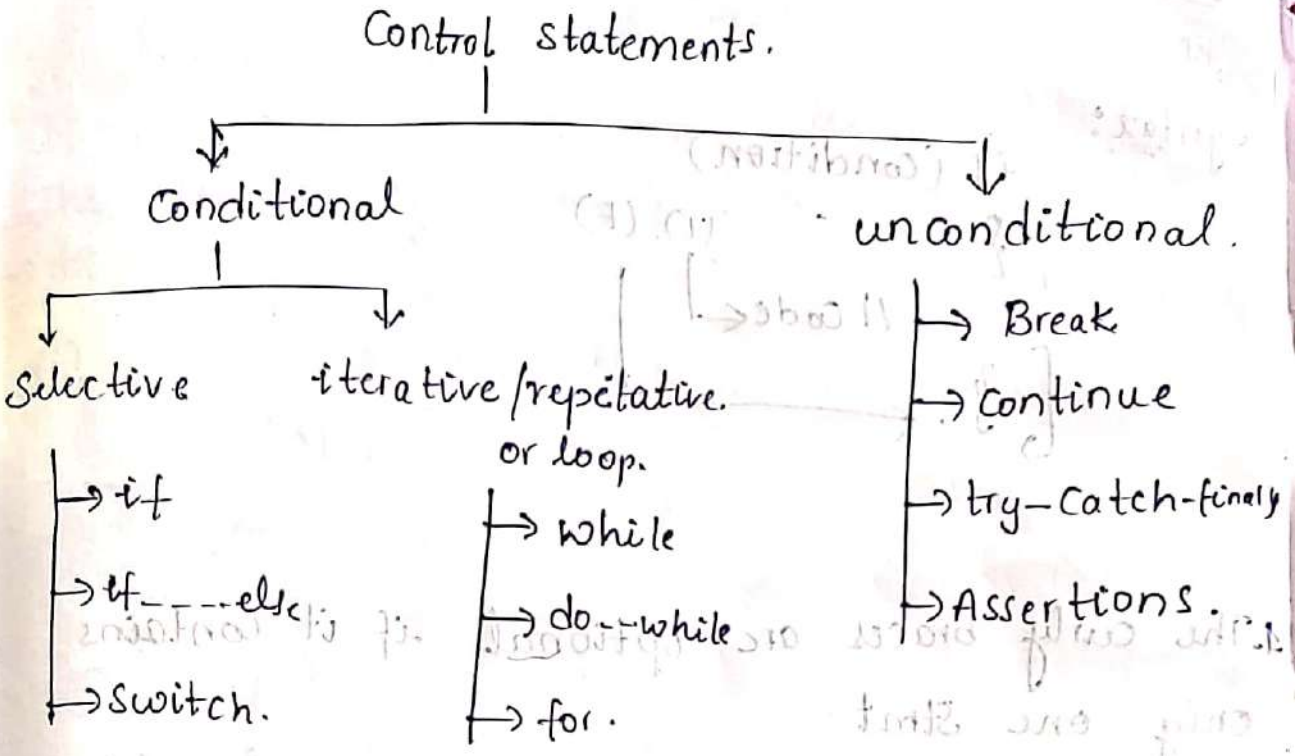
Note: The advantage of short circuit operators to increase the system performance.

Note:

Operator Precedence

Precedence	operator	Type	Associativity.
1	( )	parentheses	Lift to Right.
	[ ]	Array subscript.	
	.	Member selection	
2	++	Unary post-increment	Right to left.
	--	Unary post-decrement	
3.	++	Unary pre-increment	
	--	Unary pre-decrement	Right to left.
	+	Unary plus	
	-	Unary minu	
	!	Unary logical negation	
	~	Unary bitwise complement	
	(type)	Unary type cast	
4	*	multiplication	left to Right.
	/	Division	
	%	Modulus	
5.	+	Addition	left to Right.
	-	Subtraction	
6	<<	Bitwise left shift.	left to right
	>>	Bitwise right shift with sign extension.	
	>>>	Bitwise right shift with zero extension.	

Control Statements

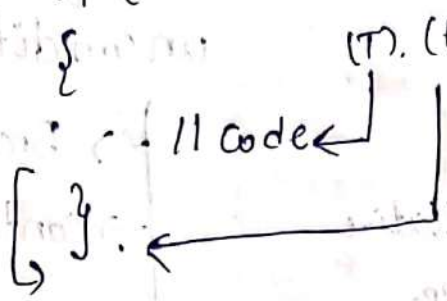


Date
16/11/21

If

Syntax:

```
if (condition)
{
    // code
}
```



1. The curly braces are optional if it contains only one stmt.
2. That stmt never be the declarative Statement.

Ex:-

(1) `if (condition)`
`SOP ("Trust the truth");` ⇒ valid

(2). `if (condition)`
`int i=10;`
invalid.

(3). `if (condition)`
`{`
`int i=10;`
`}`
valid.

④. int x=0;
if(x)
SOP("Bye");

Incompatable data type-
It is of type integer it leads to CTE by saying
The written type of condition is boolean.

⑤.
```
if(1)
SOP (" STUPID");
```

o/p :- CTE.

⑥.
```
if (True)
SOP (" STUPID");
```

o/p : stupid.

⑦.
```
if (false).
sop ("__");
```

⑧.
```
if ( )
SOP("__");
```

if-else

Switch:

```
Switch (option)
{
  case 1: ____;
          Break;
  case 2: ____;
          Break;
   .
   .
   .
  default: ____;
           Break;
}.
```

The switch is also called as multiplay (multiway) control stmts and n (dimensional) control stmts.

→ Curly braces are mandatory.

→ Defining the no. of cases depends on the type of datatype of option.

→ The allowed datatypes of option is byte, short, char, int and long.

→ The real data type and boolean data type never allows for option in switch, control stmt: voilation leads to CTE

Iterative Control Statement.

while

→ curly braces are optional. If it contains only one statement. That stmt never be the declarative statement.

Ex:

1.
```
while (0)
{
    // code
}
```
o/p :- CTE

2.
```
while (1)
{
    // code
}
```
o/p: CTE

③.
```
while (T)
{
    // code
}
```
o/p:- CTE.

(4).
```
while (true)
{
    // code.
}
```

⑤.
```
while (false)
{
    ____
}
```
leads to CTE by saying unreachable statement.

⑥. while ( )
{
// Code.
}.

o/p : CTE.

Due to incompatable datatype.

Note:- For while control stmt the compiler does not provide any default value.

Do while

Syntax:

```
do.
{
  // code
}
  while (condition);
```

Ex:

①: do
while (condition);
invalid.

②. do
int i=10;
while (condition);
invalid.

③. do
{
int i = 10;
}
while (condition);
valid.

④. do
;
while (condition);

Note: In performance way point of view while loop is best.

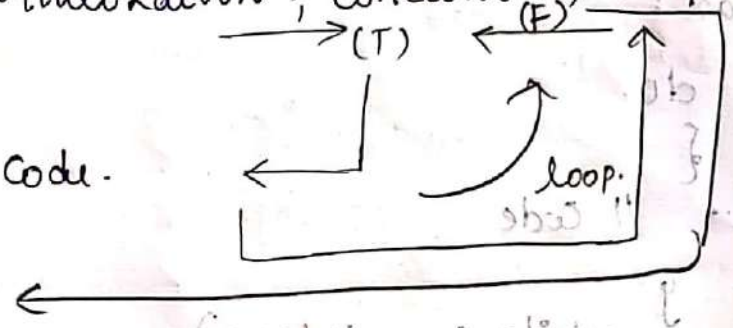
→ Possibility of while is '0' execution in case of do while ~~the possibility~~ '1 time execution possibility.

→ In while loop performance ~~for~~ increases response time decreases.

For control stmt:

Syntax:

```
for (initialization ; condition ; inc/dec)
{
    // code.
}
```




```
for (i=10; true; i++)
{
}
```

infinite

```
for(i=10 ;[   ; i++).
{
}
```

Note: For loop :- the default value is true

Note:

1. If we not mention any thing in condition of for loop, then by default true comes into picture. & goes to infinite loop.

2. If you know no.of times that loop to be executed then it's better to refer for loop.

3. If you don't know the no.of ~~loops~~ times loop to be executed then better to refer while loop.

Ex:

① . for(; ;)

~~invalid~~ valid.

② . for(SOP("Hi") ; condition; i++);

{

};

valid.

④. for (i=0 ; i<10 ; i++)
{
if (i==7)
Break;
SOP(i);
}

O/P:- 0
1
2
3
4
5
6

⑤. for (i=0; i<10; i++)
{
if (i==7)
Continue;
SOP(i);
}

O/P:- 0
1
2
3
4
5
6
8
9

command line Argument.

```
class CLADemo.
{
    P.S.V.m (String args[])
    {
        SOP ("CLA is :" + args[0])
    }
}
```

O/P:- Java CLADemo CSE.

CLA is : CSE.

```
Class CLADEMO
{
    p.s.v.m (string args[])
    {
        for(int i=0 ; i< args.length ; i++)
            SOP ( args[i];
    }
}.
```

O/P:- Java CLADemo All CSE students are stupids

All
CSE
students
are
stupids.

UNIT-2

Object

An object is a Real world entity or it is a real time entity.

(or)

An object is a entity which contains a State, behaviour and identity.

State: State of an object reflects its attributes or properties.

Ex: TV is an object.

State: Size, colour, shape

Behaviour: Behaviour of an object represents the activities or actions.

Ex: TV.

It allows to play, pause, backward and forward.

Identity: The Identity of an object represents its uniqueness.

Ex: TV never becomes a fridge.

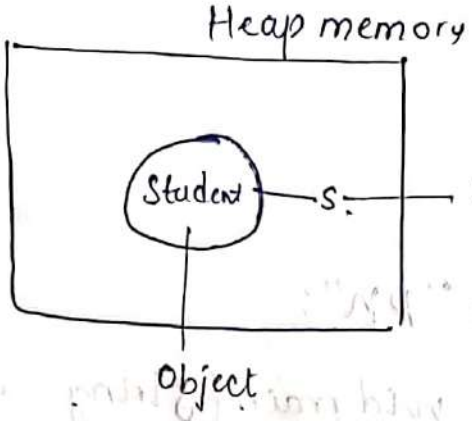
How to Create an object in JAVA.

Syntax:

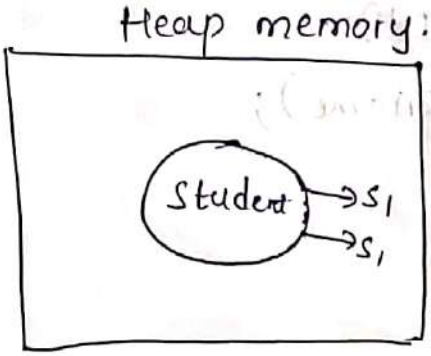
Object name objectReferenceVariable = new objectname();

operator/keyword

Ex: Student S=new Student();



Note: with the support of object reference variable we can access the members.



Note: An object have more than one reference variable.

Object.

An object is an instance of class.

class

A class is a collection of similar objects.

Note:

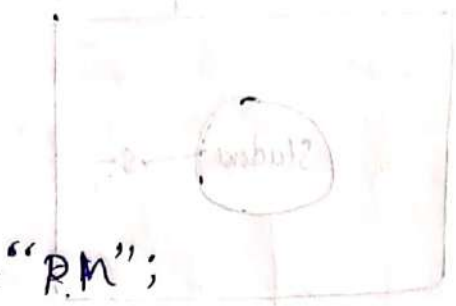
* classes are invisible where as objects are visible things.

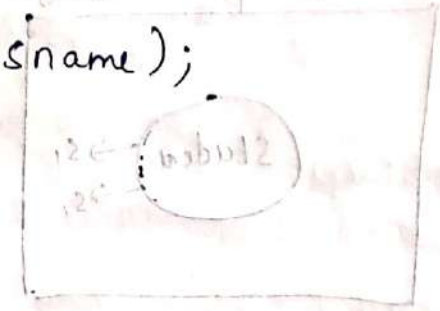
Ex:- Fruits. Mango, orange, Cherry

↓ ↓

class Objects.

```
Class student
{
    int id = 44;
    String Sname = "PM";
    Public static void main (String args[])
    {
        student s=new student();
        SOP(s.Id);
        SOP(s.Sname);
    }
}.
```





output:-

44
pm

Syntax of class.

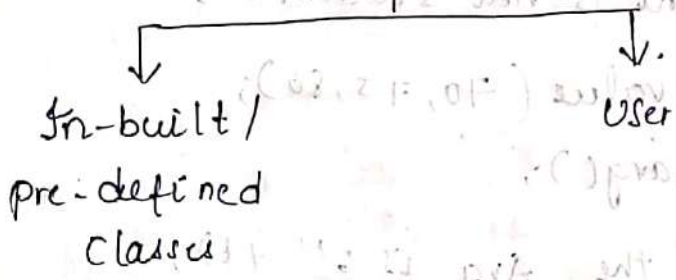
```
Class  classname.
{
    type  instance variable 1 (=value 1);
    type  instanc variable 2 (= value 2);
    :          :          :
    return type  method name 1 (parameter list 1).
      {
        = =
      }.
```

returntype method name a (parameter list a).
{
= =
}.
}.

→ Class can defined with a keyword class.
every class must have a name.

→ class contains instance variables and instance methods.

→ A prgrm contains any no. of classes but one must be main class.



Method.

Syntax: return type method name (parameter list).
{
= =
}.

Basically we can define methods with returntype and without returntype with Arguments & without arguments.

Types of methods

1. with Return type & with Arguments.
2. with Return type & without Arguments
3. without return type & with Arguments
4. without return type & without Arguments.

```
class studentDemo
{
    p.s v m (string args[])
    {
        double k;
        Student s=new student();
        s.setvalue(70,75,80);
        k=s.avg();
        SOP("the Avg is :" +k);
    }
}
Class student
{
    int m1,m2,m3;
    Void setvalue(int n1, int n2, int n3)
    {
        m1=n1;
        m2=n2;
        m3=n3;
    }
    double avg()
    {
        return (m1+m2+m3/3;
```

}

}

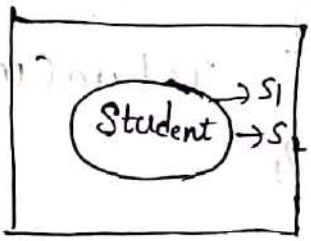
1. A program contains any no. of classes, one must be the main class. and the remaining classes are userdefined classes.
2. Object name and userdefined class name must be same.
3. we can create objects in user defined class also.
4. If the method does not return anything then the written type of that method is void.
5. If the method writtens its results then the written type is depends on data type of the result.
6. Java doesn't allow any spaces for class name.

Assigning object to another object.

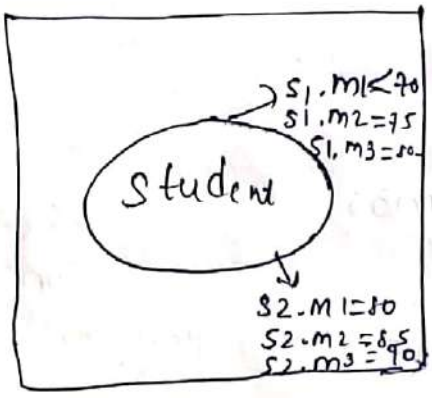
In Java we can assign an object to another object by using object reference variable.

Ex: Student s_1 = new student ();

Student s_2 = s_1;



```
Class StudentDemo
{
    P.S.V.M(String args[]).
    {
        double k1, k2;
        Student s1 = new student();
        Student s2=s1;
        s1.setvalues(70, 75, 80);
        k1=s1.Avg();
        SOP(k1);
        s2.setvalus(80, 85, 90);
        k2=s2.Avg();
        SOP(k2);
    }
}.
class student
{
    int m1,m2,m3;
    void setvalus(int n1; int n2; int n3)
    {
        m1=n1;
        m2=n2;
        m3=n3;
    }.
    double Avg().
    {
        Return(m1+m2+m3)/3;
    }
}.
```



Constructor.

The objects are automatically initialized by constructor, immidiatly whenever object is created.

Syntax: constructor name ()

```
{
    // body.
}.
```

1. Constructors are syntactically similar to methods.
2. Constructor does not have any return type where as methods have return type.
3. Constructors are little stronger than methods. (since it does not have return type).

```
class ConstDemo
{
    Psvm (String args[])
    {
        student s = new student();
        s.arg();
    }
}.
```

```
class student
{
    int m1, m2, m3;
    Student()
    {
        m1=80;
        m2=85;
        m3=90;
    }
    void avg()
    {
        double res;
        res=(m1+m2+m3)/3;
        SOP(res);
    }
}
```

Default Constructer.

→ The object name user defined class name and constructor name must be same.

→ Whenever an object is created the JVM search for a default constructor.

(i) If the default constructor found then it executes.

(ii) If the default Constructor not found then it simply leaves. (It doesn't raise compile time error).

Note: Java allows passing arguments at the time of creating object.

Types of constructors

Basically we have 2 types of Constructors.

1. Default Constructor
2. parameterised Constructor

2. Paramaterised Constructor.

```
class constDemo
{
    psvm(string args[])
    {
        Student s=new Student (70, 75, 80);
        s.avg();
    }
}
class student
{
    int m1, m2, m3;
    Student (int n1, int n2, int n3)
    {
        m1=n1;
        m2=n2;
        m3=n3;
    }
    void avg()
    {
        double res;
        res =(m+m2+m3)/3;
        sop(res);
    }
}
```

Parameterised Constructor.

Note: If you pass the arguments at the time of creating object then its programmer responsibility to develop the parameterised Constructor voilation leads to compile time error

Method Overloading.

Within a class defining two or more methods that which shares same name as long as their parameter are different then the methods in the class are said to be overloaded.

```
class methodoverloadDemo.
{
    psvm(string args[])
    {
        overload ol = new overload()
        ol.Today();
        ol.Today(10);
        ol.Today(20, 30);
        ol.Today(55, 52);
    }
}
class overload
{
    void Today()
    {
```

```
        SOP("No argument");
    }
    void Today(int i);
    {
        SOP("One argument");
    }
    void today(int j, int k)
    {
        SOP("two arguments");
    }
    void today(double d).
    {
        SOP("Double argument");
    }.
}.
```

output:

No argument
one argument
two argument
Double argument.

Constructor overloading.

As method overloading is allowed by java Compiler, constructor overloading is also possible. Since constructors are syntactically similar to methods.

Within a class defining two or more constructors that which shares same name as long as their parameter are different then the constructors in that class is said to be overloaded.

```
class const overloadDemo.
{
    psvm (string args[])
    {
        coverload col1=new coverload(-);
        coverload col2=new coverload(1);
        coverload col3 =new coverload(1,2);
        coverload col4 =new coverload(3.3);
    }.
}
class coverload
{
    coverload()
    {
        sop("No arguments");
    }
```

```
Coverload (int i)
{
    SOP ("one argument");
}
Cover load (int j, int k)
{
    SOP ("Two arguments");
}
  coverload (double d).
  {
      SOP("double argument");
  }.
}.
```

output:

no arguments.
one argument
two arguments
double argument.

28/11/21

This Keyword: This Keyword is used to find current object.

```
class thisDemo
{
    P.S.V.m (string args[])
    {
        Keyword k1 = new Keyword(1,2);
        Keyword k2=new keyword (3,4);
    }
}
Class Keyword
{
    int a,b;
    Keyword (int i, int j)
    {
        this.a=i;
        this.b=j;
        a=a+b
        SOP(a);
    }
}
```

output:

3
7.

Call by value & call by Reference

call by value:

```
class callbyvalueDemo
{
    P.S.vm (String args[])
    {
        int a=10, b=20;
        SOP("befor swapping:"+a+" "+b);
        Swap s= new Swap();
        S. Swaps(a,b);
        SOP("After swapping in main::"
                    +a+" "+b);
    }
}
Class Swap
{
    int a, b;
    void swaps (int i, int j)
    {
        a=j;
        b=i;
        SOP("after swapping in user-defined
            class is::" +a+" :" +b);
    }
}
```

output: 10 20
20 10
10 20.

Note: In call by value there is no impact on original value, where as in call by reference there is impact on original values.

29/11/21

Call by Reference.

```
Class call by reference Demo.
{
    P.S.V.m (String args[])
    {
        test t=new test(10,20);
        SOP ("Before calling:" +t.a+" "+t.b);
        t.call(t);
        SOP ("After calling:" +t1.a+" "+t1.b);
    }
}
Class test
{
    int a,b;
    test(int i, int j)
    {
        a=i;
        b=j;
    }
    void call(test t1)
    {
        t1.a=t1.a*2;
        t1.b=t1.b*2;
    }
}.
```

Access specifier / Access controller / Access modifier.

In any object oriented programming we have 4 types of Access Specifiers.

1. Public
2. protected
3. Default:
4. Private.

In Java, we have 5 areas where we can apply the access specifeers.

1. within the class / same class
2. Sub class of same package.
3. Non-sub class of same package.
4. Sub-class of outside the package:
5. Non-sub class of outside the package.

	Public	Protected	Default	Private.
1. With in the class/ Same class	✓	✓	✓	✓
2. Sub class of Same package	✓	✓	✓	✗
3. Non-sub class of same package	✓	✓	✓	✗
4. Sub-class of outside the package	✓	✓	✗	✗
5. Non-Subclass of outside the package	✓	✗	✗	✗

Public: If you define your member as a Public then it allows you to access from anywhere.

Protected: If you define you member as protected then it allows you to access within the package and sub-class of outside package.

Default: If you define your member as default then it allows you to access within the package.

Private: If you define your member as private then it allows you to access within the class.

Note: We can show the variation between default and public only in the concept of packages.

30/11/21

Working with private Members.

```
class privateDemo
{
    p.s vm (String args[])
    {
        test t=new test();
    }
}.
```

```
class test
{
private test().
 {
      SOP (" it is private constructor "); //CTE
 }.
```

→.

```
class PrivateDemo.
{
   P.S.vm(string arg.[])
   {
       test t=new test();
       SOP(t.a); //CTE
       t.cell(); // CTE
   }
}.

class test
{
  private int i=20;
       test().
   {
       int a=i;
   }.
   private void call().
   {
       SOP("call method");
   }
}.
```

o/p :- CTE.

Note: From the above its very clear private members are not allowed to access, if you try to access the private members lead to compile time error by saying private members are not allowed to access.

→ But we can access the private members by using in-built setter & getter methods.

```
Class privateDemo
{
    P.S.V.m (String args[])
    {
        test t=new test();
        t.setAge();
        t.setName();
        SOP("Age :" + t.getAge());
        SOP("Name :" + t.getName());
    }
}
class test
{
    private int age =24;
    private String name = "Dont know";
        void setAge()
        {
            t.age =age;
        }
        void setName()
        {
            t.name =name;
        }
}
```

Static keyword is used for effectively usage of memory management.

→ static is a keyword allowed to apply to
- variables
  Methods
  Blocks
  Nested classes.

static variables

when we can define a variable is a static, only if that variable have common property of an object (all objects).

```
Class StaticDemo
{
    P.S.V m(String args[])
    {
        Emp E1 = new Emp(101, "sam");
        Emp E2 = new Emp(202, "Ram");
        E1.display();
        E2.display();
    }
}
Class Emp
{
    int id;
    String Ename;
static String Company = "IBM";
    Emp(int i, String nam)
```

```
{
    id = i;
    Ename = name;
}
void display()
{
    SOP("id:" + id + "employe name:"
        + Ename + "company name:"
        + company);
}
}
```

id:101 employe name : Sam company name : IBM
id: 202 employe name : ram company name : IBM.

Static Methods.

```
class staticDemo
{
    P.S.V.M (String.args[])
    {
        Emp E1 = new emp[101, "sam");
        Emp E2 = new emp(202, "Ram");
        E1.display();
        E2.display();
        Emp E3 = new Emp(303, "Raj");
        E3.change();
        E3.display();
    }
}
```

```
class Emp
{
    int id;
    String Ename;
    Static String Company = "IBM";
    Emp(int i, String name)
    {
        id = i;
        Ename = name;
    }
    void display()
    {
        Sop("id:" + id + "employee name:"
            + Ename + "company name:"
            + company);
    }
    Static void chang()
    {
        company name = "WIPRO";
    }
}.
```

Date
02.12.21

Garbage collection / collector:

In c++ the deletion of objects can be done manually by using delete keyword.

→ where as in java the deletion of objects can be done automatically by garbage collector.

→ Garbage collector

Garbage collector collects all unused or unreachable objects that are left in the memory for long time and kills.

→ To make the heap memory free to provide space future objects.

→ If you try to create the objects even the heap memory gets filled, it leads to an error called (out of memory E) "OutofmemoryError".

→ JVM will runs the GC.

→ we are unable to expect when JVM runs GC.

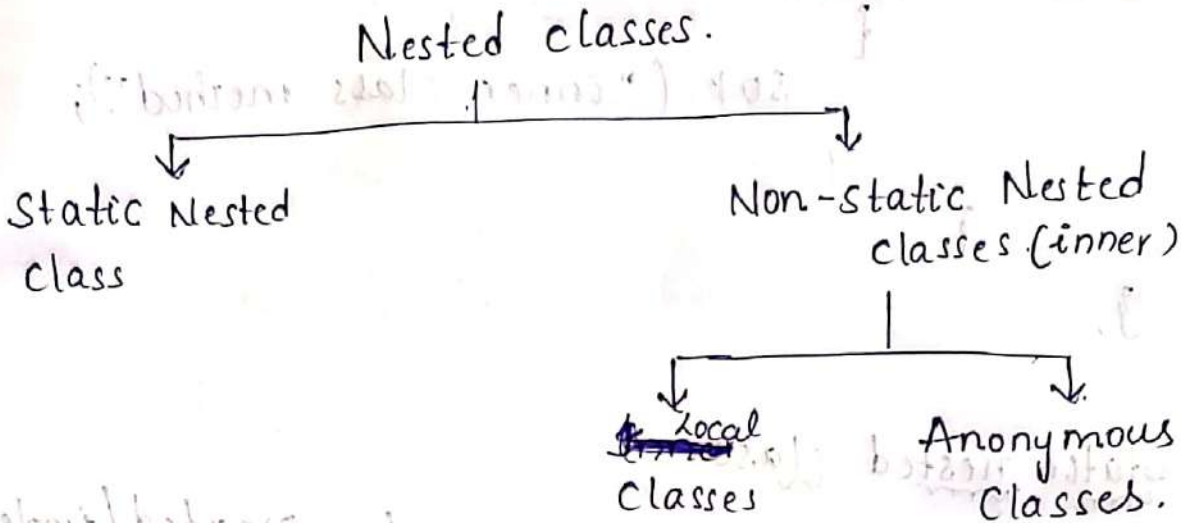
→ JVM allows to run the GC manually by calling System.gc()

→ The java allows the objects that eligible for GC manually, by using a Keyword null.

Syntax:-

```
E = null;
```

Nested class



Nested class

A class which is inside another class is called nested class.

Non-static Nested (or) inner class.

```
Class NestedDemo
{
    P.s.v.m(String args[])
    {
        outer O = new outer();
        O.display();
    }
}
Class outer
{
    void display()
    {
        sop("outer class method");
        Inner i=new inner();
        i.disp()
```

```
}
class inner
{
    void disp()
    {
        sop ("inner class method");
    }
}
}.
```

Static nested class

Static nested classes can be created/implemented by using the following syntax.

Syntax:

```
outerclass.innerclass RV = new outerclass.innerclass();
```

RV → Reference variable.

```
class NestedDemo
{
    p.s.v m (String args[])
    {
        outer.inner oi = new outer.inner();
        oi.display();
    }
}.
class outer
{
```

```
int i=10;
Static class inner
{
        int j=20;
        void display()
        {
            i = i+j;
            SOP(i);
        }
}
};
```

o/p: 30

Recursive Methods

A method that calls itself then that method is called recursive method.

```
Class RecMethodDemo.
{
    P.S.V.m(String args[])
    {
        int res;
        res = sum(20);
        SOP(res);
    }
Static int sum(int k).
{
        if (k>0)
        return k + sum(k-1);
        else
        return 0;
    }
}
```

Note:

Static members are class members not to the object.

Unit-3.

Array:- It is a collection of similar data elements under a single variable name that which allows to store the data in continous memory location

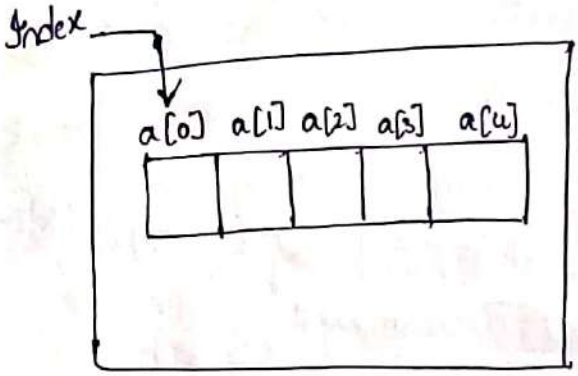
	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

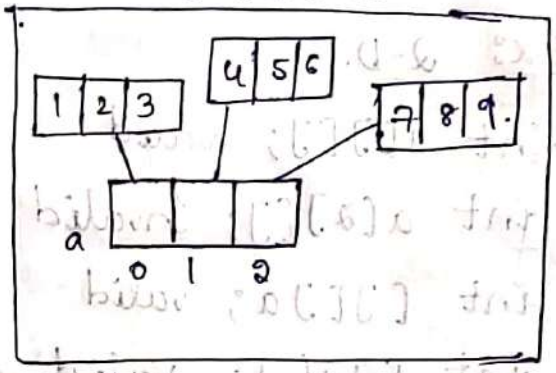
2-D → Base size(rows) → columns.

int a[3][3];

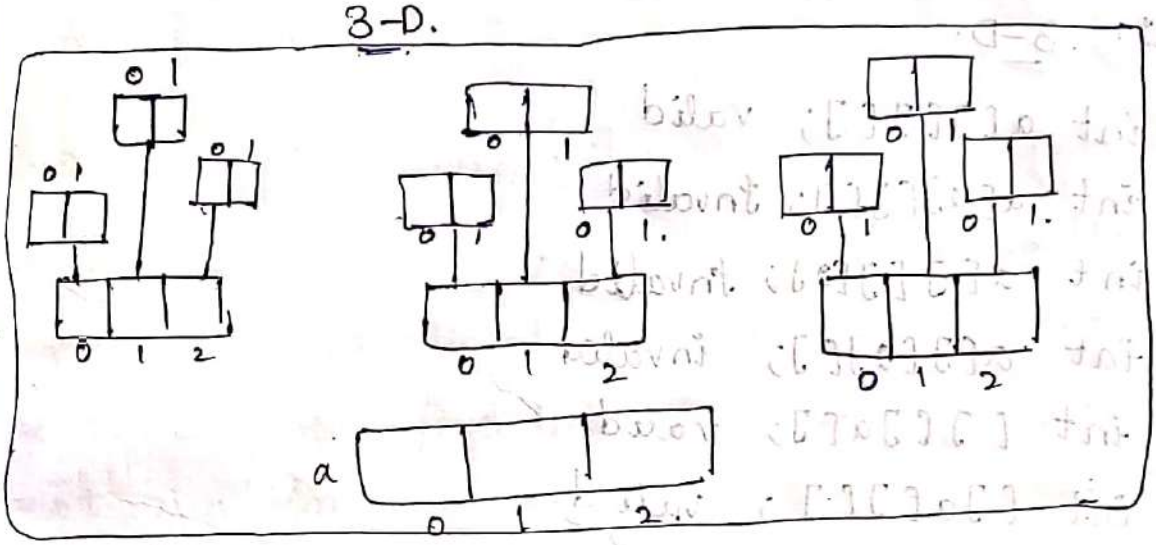
In C- memory diagrams

int a[5]





3-D.



In Java

1. Declaring an array
2. Constructing an array
3. Initializing an array.

1. Declaring an array.

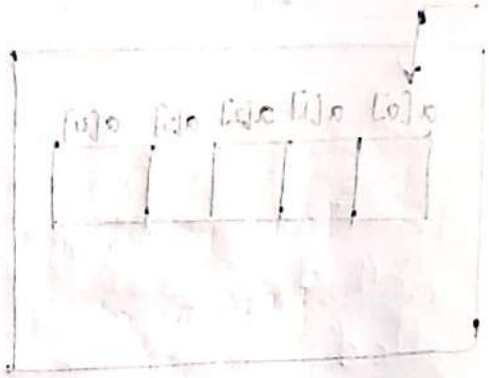
Syntax: Datatype arrayname[];

examples for 1-D array:

```
int a[];    valid
int a[3];   invalid.
int []a;    valid
int [5]a;   invalid.
```

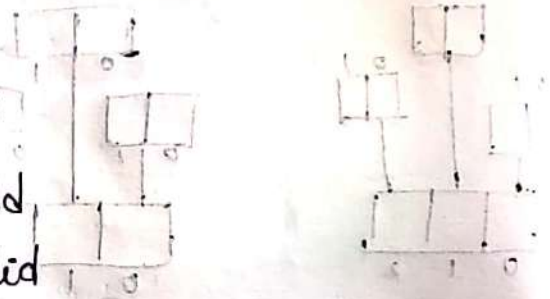
Ex: 2-D.

```
int a[][];    valid
int a[2][];   invalid
int [][]a;    valid
int []a[];    valid.
```



Ex: 3-D.

```
int a[][][];    valid
int a[2][][];   invalid
int a[][][2];   invalid
int a[][2][];   invalid
int [][]a[];    valid.
int []a[][];    valid.
```



Note: If you mention the size of an array at the time of declaration, leads to compile time error.

2. Constructing an array.

syntax: 1-D: Datatype arrayname[] = new datatype [size];

2-D: Datatype arrayname[][] = new data type [Base size][column size];

1. we can construct the array by using a new keyword.
2. At the time of constructing the array the size of an array must represent violation leads to compile time error.

Ex: int a[] = new int [];, invalid.

int a[] = new int [5]; valid

int [] a = new [5] int ; valid

int [5] a = new []int ; Invalid.

→ int a[] = new int [0];

*. In Java zero size arrays are allowed but no space is allocated.

* It compiles fine but it raises an exception at run time by saying zero size array exception

Ex:- int a[] = new int [-7];

In Java negative size arrays are not allowed leads to compile time error.

Ex: int a[][] = new int [3] [2]; valid.

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

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

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

int [][]a = new [3] [] int; valid.

while constructing an array base size must be mentioned the rest are optional.

07/12/21

byte b=10;
int a[] = new int[b];

Note:
we can pass the data types by int, byte and char as a element to an array.

(or)
size

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

o/p: CTE.

3. Initialization of an array.

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

(or).

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

→ int a[] = new int[5] {1, 2, 3, 4, 5};

2-D: int [][]a = {{1,2,3},{4,5,6},{7,8,9}};

→ String s[] = {"CSE","IT","CSD"};

Basic operations on array.

1. Traversal.
2. Search
3. Deletion
4. Insertion
5. Update.

1. Traversal.

```
Class TraversalDemo
{
    P.S.V.M (String args[])
    {
        int n,i;
        int a[] = new int[n];
        SOP ("enter array size:");
        Scanner sc= new Scanner (System.in);
        n = sc.nextInt();
        int a[] = new int[n];
        SOP ("enter elements to array::");
        for (i=0; i<n; i++).
            a[i]=sc.nextInt();
            SOP ("the elements in array is::");
            for (i=0; i<n; i++)
                SOP(a[i]);
    }
```

2. Search :

```
Class : Traversal Demo
{
    p.s.v.m( String args[] )
    {
        int n, i, Key, flag;   correct :
        int a[] = new int[n];
        SOP("enter array size:");
        Scanner sc = new Scanner (System.in);
        n = sc.nextInt();
        int n, i, Key, flag;
        SOP("enter elements to array::");
        for (i=0; i<n; i++)
            a[i] = sc.nextInt();
            SOP("enter key");
            Key = sc.nextInt();
            for (i=0; i<n; i++)
            { if (a[i] == Key)
                {
                    flag = 1;
                    Break;
                }
                else
                { flag = 0;
                }
            }
            if (flag == 1)
                SOP("element found");
```

```
else
    sop("not found");
  }
}
```

3. Deletion

```
Class DeleteDemo.
{
    p.s.v.m (string args[])
    {
        int n, a[i], ind, flag;
        SOP ("enter the element to delete:");
        del = sc.nextInt();
        for (int i=0; i<n; i++)
        {
            if (a[i] == del)
            {
                flag = 1;
                ind = i;
                Break;
            }
            else
                flag = 0;
        }.
        if (flag == 1)
        {
            for (i=ind; i<n; i++)
            a[i] = a[i+1];
        }
        S.O.P("
```

4 Insertion

Array passing to another array.

Passing Array to Another array.

Java supports passing of array to another array

→ for multidimensional applications we can use passing arrays to another array concept.

```
Class Array to array Demo. (no gap).
{
    P.S.V.M (String args[])
    {
        int a[] = {1, 2, 3};
        int b[] = a;
        SOP("the elements in array a is :");
        printarray(a); (no gap)
        SOP("the element in array b is :");
        printarray(b); (no gap)
    }
    static void printarray(int x[])
    {
        for(int i=0; i<x.length; i++)
            SOP(x[i] + " ");
    }
}
```

O/P:- the elements in array a is :

1
2
3

the elements in array b is :

2-D

```
Class arraytoarrayDemo.
{
    P.S.V.m(String args[])
    {
        int a[][]={1,2,3},b[][]={4,5,6};
        int b[][]=a,a[][]=b;
        printar
        int a[][]={{1,23},{4,5,6}};
        int b[][]=a;
        sop("the elements int array a is:");
        printarray(a);
        sop("the elements in array b is:");
        printarray(b);
    }
    static void printarray(int x[], y[])
    {
        for(int i=0; i<x.length; i++)
        sop(x[i]+" ");
        {
            for(int j=0; j<i.length; j++)
            sop(y[j]+" ");
        }
    }
}
```

O/P:- the elements in array a is

4

7

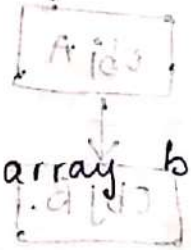
8

2D

```
class arraytoarrayDemo
{
    p.s.v.m(String args[])
    {
        int a[][] = {{1,2,3},{4,5,6},{7,8,9}};
        int b[][] = a;
        S.o.p("the elements in array a is:");
        printarray(a);
        S.o.p("the elements in array b is:");
        printarray(b);
    }
    static void printarray(int x[][])
    {
        for(int i=0; i<x.length; i++)
        {
            for(int j=0; j<i; j++)
                S.o.p(x[i][j]+" ");
        }
    }
}
```

O/P:-

```
the elements in array a is:
4
7
8
9
the elements in array b is:
4
7
8
9
```



13/02/21

Inheritance

Aquiring all properties and behaviour of one object/class (super) by another object/class (sub) is called inheritance.

→ In Java we can implement the inheritance by using extends keyword.

→ A class that derives the properties is a child class or derived class, A class that derives from is a parent class or base class.

```
class A
{
    obj
}
class B
{

}.
```

All properties & Behaviour extends A.

Types of inheritance

1. Single - level inheritance
2. Multi - level inheritance
3. Hierarchal inheritance.
4. Multiple inheritance
5. Hybrid inheritance.

1. Single - level inheritance

(In case of constructor both objects are mandatory.)

```
class inheritDemo
{
    p.s.v m(String args[])
    {
        A a=new A();
        B b=new B();
    }
}.
class A
{
    int i=10;
    A()
    {
        sop(i);
    }
}
Class B extends A.
{
    B()
    {
        sop(i);
    }
```

O/P:- 10
10
10

Note:-

In the above code creating both the object A and B are mandatory.

```
Class Inherit Demo.
{
    p.s.vm ( string args[] )
    {
        B b=new B();
        b.display1();
        b.display2();
    }
}.
Class A.
{
    void display1()
    {
        SOP ("super class");
    }
}
class B extend A
{
    void display2()
    {
        SOP ("sub class");
    }
}.
```

O/P:- Superclass
Subclass

Note: Here we can take creation of an object A optional

→ The major advantage of inheritance is "code reusability."

• when we go for inheritance?.
whenever more than one class have common properties then we can go with inheritance.

2. Multi-level inheritance

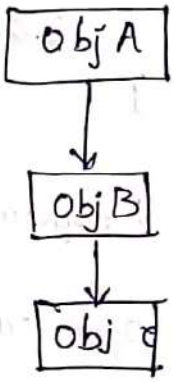
Syntax:

```
Class A.
{

}
Class B extends A
{

}
Class C extends B
{

}
```



inherita is infinite or limit?

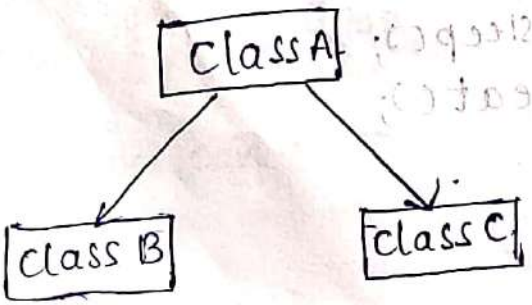
```
class multiinheridemo
{
    p.s.v.m(String args[])
    {
        babydog b=new babydog();
        b.bark();
        b.sleep();
        b.eat();
    }
}
```

```
class Animal
{
    void bark()
    {
        S.O.P ("Barking");
    }
}
Class Dog extends Animal
{
    void sleep()
    {
        S.O.P("sleeping");
    }
}
clas babydog extends Dog
{
    void eat()
    {
        S.O.P("eating");
    }
}.
```

output:-
barking
Sleping
eating.

3. Hierarical Inheritance

A single class get inherited by more than one class is the central concept of hierarical inheritance.



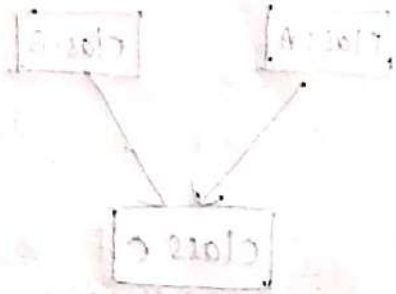
Syntax:

```
class A.
{

}
class B extend A
{

}
class C extend A.
```

```
class HieraricInheritDemo.
{
    p.s.v.m (String args[])
    {
        Dog d=new Dog();
        babydog b= new babydog();
        b.bark();
        d.sleep();
        b.eat();
    }
}.
class Animal.
{
    void bark()
    {
        S.O.P("Barking");
    }
}
class Dog extends Animal
{
    void sleep()
    {   S.O.P("sleeping");
    }
}
```

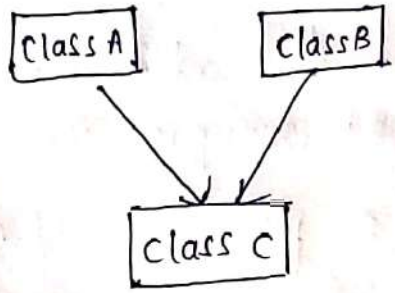


```
Class babydog extends animal
{
    void eat()
    {
        S.O.P("eating");
    }
}.
```

O/P: Barking
Sleeping
eating.

4. Multiple Inheritance-

Eg A class inherits the property from more than one class is called multiple inheritance.



Syntax:

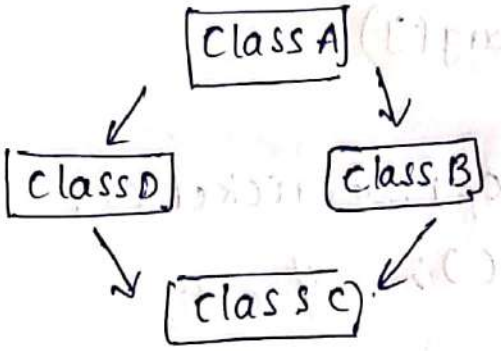
```
class A
{
}
Class B
{
}
class C extends A, B
{
}
```

```
class sportDemo
{
    p.s.vm (String arg[])
    {
        cricket c=new cricket();
        c.score(); // CTE
    }
}
class sports
{
    valid score()
    {
        s o p ("sports.class");
    }
}
class games
{
    void score()
    {
        sop ("Games class");
    }
}
class cricket extends sports, games
{
}
```

Note: In java multiple inheritance is not possible but we can make it possible by using interfaces.

Reason: Because when weka superclass have common members then subclass is unable to call the perfect member of super class. This ambiguity.

5. Hybrid Inheritance



Note: 6-8 levels of inheritance is reasonable
10-12 levels of inheritance is acceptable
Above 12 levels of inheritance is Dangerous.

Method Signature

Type Signature defines the method name followed by parameterised data type's.

→ Here variables and return type of the method doesn't play any role.

→ The order of arguments must be preserved.

(If you want to say 2 (or) more methods have signature are same (or) not.

```
void test (int i, int j)        // test (int, int).
{
}
int test (int i, float f)       test (int, float).
{
}
int test (int j, int k).        test (int, int)   <- Same
{
}
void test (float f, int i).     test (float, int).   <- not same
{
}
```

Method Overriding.

Sub class have same method name and type signature as in the super class method then the method in sub class is said to be overridden by super class method.

```
class BankDemo
{
    public static void main(String args[])
    {
        UCO u=new UCO();
        u.withdraw();
    }
}
class Bank
{
    void withdraw
    {
        S.O.P("interest is more than 5");
    }
}
Class UCO SBI extends Bank.
{
    void withdraw()
    {
        S.O.P("interest is more than 8");
    }
}
Class UCO extends Bank.
{
    void withdraw()
    {
        S.O.P("interest is more than 10");
    }
}
```

→ class BankDemo.

```
{
    p.s.v m (string args[])
    {
        int rate;
        SBI  S= new SBI ();
        UCO  U =new UCO();
        ICICI  I =new ICICI();
        rate = S.ROI();
        Sop(" the SBI ROI is :" + rate);
        r1 =U.ROI();
        Sop(" the UCO ROI is :" +r1);
        r2 = I.ROI();
        Sop(" the ICICI ROI is :" +r2);
    }
}
Class Bank
{
    int ROI()
    {
        return 0;
    }
}
Class SBI extends Bank
{
    int ROI()
    {
        return 1;
    }
}
class UCO extends Bank.
{
    int ROI()
    {
        return 2;
    }
}
```

```
class ICICI extends Bank.
{   int ROI ()
    {
        return 3;
    }
}.
```

o/p: 1 2 3.

Dynamic Method Dispatch / Run time polymorphism

It defines assigning sub class object to super class.

In dynamic

A B

A a = new B();

In general inheritance

A a = new A();

B b = new B();

```
class dynamicDemo
{
    P.S.V.M (String args[])
    {
        Shape s;
        s = new Rectangle();
        s.Draw();
        s = new triangle();
        s.Draw();
    }
}
class shape
{   void draw()
    {   SOP ("shape has no. of dimensions");
    }
}
```

→
```
Class rectangle
{
    void draw()
    {
        SOP ("rectangle has two dimensions");
    }
}
class triangle extends shape
{
    void draw()
    {
        SOP ("triangle has three dimensions");
    }
}
```

O/P: Rectangle has two dimensions
triangle has three dimensions.

Note: When all the subclasses and superclasses have the common members (method, variable) while accessing those members from main class we have some issues, we can achieve this using Runtime polymorphism.

→ Java allows to overide the methods but we can't ovaride the variables

→ If there is no concept of inheritance then there is no concept of method overiding.

→ Method overloading takes place within class where as method place between the classes.

Polymorphism
- Method overloading
- Method overiding.

20/12/21

Super Keyword.

Super is a Keywoid allowed to apply to variables, methods and Constructor.

→ Super Keyword never applied to a class.

Uses of super Keyword

1. we can call super class methods and Constructors.
2. we can call the super class member that which was hidden by sub class member.

Use-1 Example

```
Class SuperDemo
{
   p.s.v.m (string args[])
   {
       Dog d = new Dog ();
       d.eat ();
   }
}
class animal
{
    void bark()
    {
        S.O.P ("Barking");
    }
}
Class Dog extends animal
{
    void eat ()
```

```
    {
        super.bark();
        S.o.p("eating");
    }
}
```

O/P:- Barking.
Eating.

Use1 - Example-2.

```
class SuperDemo
{
    P.S.v.m(String args[])
    {
        Dog d = new Dog();
    }
}
class animal
{
    animal()
    {
        S.o.p("Barking");
    }
}
class Dog extends animal
{
    Dog()
    {
        // super();
        S.o.p("Eating");
    }
}
```

O/P:- Barking
Eating.

→ Super keyword always be the first statement in the block. violation leads to compile time error.

→ For a default constructor in the sub class, super is optional to call super class constructor

order of calling constructor.

```
class SuperDemo
{
  P.S.V.M(String args[])
  {
     babyDog d = new babyDog();
  }
}
Class animal
{
   animal()
   {
     S.O.P("3");
   }
}
class Dog extends animal
{
    Dog()
    {
      S.O.P("2");
    }
}
Class babyDog extends Dog.
{
     babyDog()
     {
        S.O.P("1");
     }
}
```

O/P:- 3
2
1

Use 2 - Example (super).

```
Class SuperDemo()
{
   P.S.V.M (String args[])
   {
      Dog d = new Dog();
      d.today();
   }
}
class animal
{
   int k=10;
}
Class Dog extends animal
{
   int k=20;
   void today()
   {
      SOP("The value of k in SUB:"+k);
      SOP("The value of k in super:"+super.k)
   }
}
```

o/p: The value of k in Sub: 20
The value of k in super: 10.

Final keyword

Final is a Keyword allowed to apply to class, methods and variables, we never applied this Keyword to blocks.

Notes

→ If you apply the final keyword to a method then we can prevent the method overiding.

→ If you apply the final keyword to a class then we can prevent the inheritance.

→ If you apply the final keyword to a variable, if you try to re-initiate the value of that variable leads to compile time error.

For Method.

```
Class finaldemo
{
    P.S.V.m (String args[])
    {
        Bluepen b= new bluepen();
        Blackpen l = new blackpen();
        lead D =new lead();
        b.pencil();
        l.pencil();
        D.pencil();
    }
}
Class Pen
{
    void Pencil()
    {
        S.O.P("Pencils are of different types");
    }
}
```

```
Class Blue pen extends pen
{
    void pencil()
    {
        S.o.p("Blue pens are different");
    }
}
class Blackpen extends pen
{
    final void pencil()
    {
        S.o.p("Black pens are different");
    }
}
class lead extends pen
{
    void pencil
    {
        S.o.p("leads are used in pencils");
    }
}.
```

O/P:- Blue pens are different
Black pens are different
lead are used in pencil

For a class

```
class final Demo.
{
  P.S.V.M (String args[])
  {
     babydog b= new babydog ();
     Dog d= new Dog ();

  }
}
class animal
{
   animal()
   {
     S.O.P("Barking");
   }
}
class Dog extends animal
{
   dog()
   {  S.O.P("Eating");
   }
};
final Class babyDog extends animal
{
   babydog()
   {
     S.O.P("very cute");
   }
}.
```

23/12/21

Abstraction Keyword

Abstract is a keyword allow to apply to class and methods.

→ we never apply to blocks, variables and constructor.

→ By using abstract keyword we can achieve data abstraction.

If we apply the abstract keyword to a method then.

1. It must ends with semicolon.
2. It never contains implementation (body {})
3. It's implementation takes place in its sub classes.
4. If method is abstract then class must be abstract violation leads to CTE.
5. If class is abstract then method may or may not be abstract.

Syntax:

```
abstract returntype methodname();
```

Ex: abstract void test();

```
class abstDemo
{
    P.s.vm (string args[])
    {
        Computer c=new computer();
        c. PC();
    }
}
Abstract class System
{
    abstract void PC();
}
class computer extends System
{
    void PC()
    {
        SOP ("personal computer");
    }
}
```

o/p:- personal computer.

```
class abstDemo
{
    P.S.V.m (String args[])
    {
        computer c=new computer();
        embedded system es=new embeddedsyste
        c.pc();
        es.fridge();

    }
}
abstract class system
{
    abstract void pc();
}
class computer extends system
{
    void pc()
    {
        sop("personal computer");
    }
}
class embedded system extends system
{
    void fridge()
    {
CTE//    sop("embeded system");
    }
}
```

Note:

All the abstract method must implemented in its all sub classes violation leads to CTE.

Note: :

1. Abstract class allows non-abstract methods
2. The sub classes can happily implemented other methods along with super classes abstract method.

Interface

Introduction to interface.

Defining Interface

Implementing Interface

Applying Interface

Extending interface

Variables in interface.

Introduction to interface

"Interface's says what a class must do," "but not how it does". i,e it hides the implementation details but it exploits its services, it's the central concept of interfaces.

→ By using interfaces we can acheive one of the oops principle polymorphism. i,e. "one interface multiple method."

→ The major advantages of interface is Security and enhancement (extensibility).

2. Defining interface

Syntax:

```
Access interface Interfacename
{
    returntype methodname1 (Parameterlist 1);
    returntype methodname2 (Parameterlist 2);
    :
    returntype methodnamen (Parameterlistn);

    Datatype variablename1 = value1;
    Datatype variablename2 = value2;
    :
    Datatype variablenamen = valuen;
}
```

1. interfaces are syntactically similar to class
2. The methods in interface does not contain body and ends with semicolon (;). where the methods in classes have body. (implemented)
3. The variables in interface must assigned with a value at the time declaring variable whe as in classes it is optional.
4. The methods in interface are by default abstract.

28/12/21

3. Implementing Interface

~~In an implement~~ we can implement an interface in a class by using "implements" keyword or classe

Syntax

```
access class classname implements interfacename
{
  - - - -
  - - - -
}
```

```
access class classname extends superclass implements interface name
{
  - - -
  - - -
}
```

A class can extend the properties of another class as well as implements an interface at a time.

```
access class classname implements interface1, interface2,...
{
  - - -
  - - -
}
```

Note:

A class can implement any no. of interfaces at a time.

→ Multiple Inheritance

```
class year
{
    Public static void main (String args[])
    {
        Sem S = new sem();
        S.Assignment1();
        S.Descriptive1();
        S.Assignment2();
        S.Descriptive2();
    }
}
interface MID1
{
    void Assignment1();
    void Descriptive();
}
interface MID2
{
    void Assignment2();
    void Descriptive2();
}
class sem implements MID1, MID2
{
    public void assignment1()
    {
        S.O.P("for 10 marks");
    }
    public void Descriptive()
    {
        SOP("for 20 marks");
    }
    public vaid Assignment2()
    {
```

```
        sop("for 10 marks");
    }
    public void Descriptive2()
    {
        sop("for 20 marks");
    }
}
```

O/P:- for 10 marks
for 20 marks
for 10 marks
for 20 marks.

Note:
Abstract class (contains) allows abstract methods as well as non-abstract methods. where as interfaces doesn't allow non-abstract methods.

→ How classes are there in the prgm that many no.of class files will be created in the current working directory.

Functional Interface (interface with abstract

```
Interface A
{
    void meth();
}
class today implements A.
{
    public void meth()
    {
        system.out.println("Interfaces");
    }
```

```
}
class interdemo
{
    public static void main(String args[])
    {
        today t = new today();
        t.meth();
    }
}.
```

o/p:- Interfaces.

Interface never allows non-abstract methods;

→ if we want to implement the non-abstract method in an interface we go with static keyword (which is independent of object)

Static Interface.

```
interface jio
{
    void custcare();
    public static void techsupport()
    {
        System.out.println("its nice");
    }
}
class sim implements jio
{
    public void custcare()
    {
        System.out.println("99999999");
    }
```

```
}
class mobile
{
    public static void main(String args[])
    {
        Sim s=new Sim();
        s.custcare();
        jio.techsupport();
    }
}
```

→ In an,

Nested Interface

```
class animal
{
interface ani
{
   void barking();
}
}
class dog implements animal.ani
{
   public void barking()
   {
      System.out.println("Dog always barks like humans");
   }
}
class nestedinterfacedemo
{
    public static void main(String args[])
```

```
{
    dog d = new dog();
    d.barking();
  }
}
```

output:

Dog always barks like humans.

Note:

→ Nested interfaces can be done in two ways 1. by declaring interface within a class

2. By declaring an interface in another interface

```
→ interface A
  {
     interface B
     {
        void barking();
     }
  }
  public class nestedinterfacedemo implements A.B
  {
     public void barking()
     {
        System.out.println("Hi Java dont forget me");
     }
     public static void main(String args[])
     {
        A.B d = new nestedinterfacedemo(); //upcasting
        d.barking();
```

```
    }
}
```

o/p:- Hi java don't forget me.

Multiple ~~extending~~ Interface. (Achieving multiple inheritance).

```
interface A
{
    void time1();
}
interface B
{
    void time2();
}
class sam implements A, B
{
    public void time1()
    {
        System.out.println("If you follow the
                            time");
    }
    public void time2()
    {
        System.out.println("It makes you Queen");
    }
}
class timedemo
{
    public static void main(String args[])
    {
        Sam s=new Sam();
        s.time1();
```

```
        s.time2();
    }
}
```

Extending Interface.

```
interface A
{
    void time1();
}
interface B extends A
{
    void time2();
}
class sam implements B
{
    public void time1()
    {
        system.out.println("If you follow the
                            time ");
    }
    public void time2()
    {
        system.out.println("It makes you quen
    }
}
class timedemo
{
    public static void main(String args[])
    {
        sam s = new sam();
        s.time1();
```

s.time2();

}

}

→ The variables in interface are by default final.

Date:
08/01/22

UNIT-4

Packages

Packages

The major advantage of using packages are to overcome the naming conflicts of .class files. (dot class files)

Syntax: Package Packagename;

Ex: Package K1;

Types of packages

Basically we have two types of packages inbuilt packages, userd-defined

1. In built packages
2. User-defined packages.

1. In-Built packages

Java provides several inbuilt packages lang, util, io, Applet, Swing, AWT (abstract window tool), Servelt that which contains several objects and methods.

2. User-defined packages

Java. Supports user-defined packages By using these user-defined packages java allows to share the content between

ways to import a package.

we have three possible ways of importing a package.

(i). Using package name
(ii) Using class name
(iii) Using fully qualified name.

(i) Using package name.

```
Package k2;

Public class A
{
Public void display()
{
    sop("using package
        name");
}
}
```

Save as A.Java.

A.class

```
Package k1;
import k2.*;
Public class B
{
    P.S vm(String arg:[])
    {
        A a=new A();
        a.display();
    }
}
```

B.Java.

B.class

How to compile a package program.

javac -d : filename.java.

Here -d stands for current working directory

→ After compilation JVM creates a new folder with the current working directory with a name of package name.

→ And it places all the class files that belongs to that package in that folder.

How to Run a package.

Java packagename.filename.

Ex: Java K2.B.

Note: Package statement always the first statement in the program violation leads to CTE.

(ii) Using class name

```
Package K2;
Public class A
{
    void display()
    {
      SOP("using class name");
    }
}
```

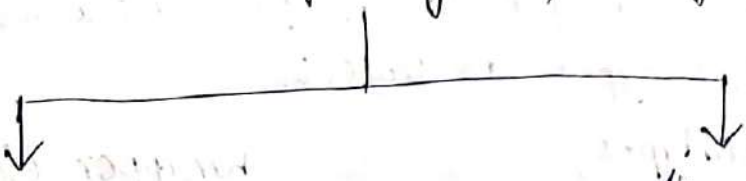
```
Package K1;
import K2.A;
class B
{
  P.S.V.m(String args[])
  {
     A a=new A();
    a.display();
  }
}
```

(iii) Fully Qualified Name

```
Package k2;
Public class A
{
    void display()
    {
      SOP("using class
          name");
    }
}
```

```
Package k1;
class B
{
  P.S.V.m(String args[])
  {
    K2.A a=new k2.A()
    a.display();
  }
}
```

Importing a package.



Explicit
(fully qualified name)

Ex: Java.util.array,
Java.util.arraylist

Implicit.

* Import is mandatory

Ex: import package.*;
name;

Note:
Explicit importing is best because it takes less time to compile but in the real time we always going with implicit importing a package. Because a programer is unable to remember all class, objects and method names.

order of statements in a program.

1. Package stmts
2. Import stmts
3. Class stmts.

Note:
The no: of package stmts is more then its compile time is also more. But there is no difference in run time.

→ And it places all the class files that belongs to that package in that folder.

How to Run a package.

Java packagename.filename.

Ex: Java K2.B.

Note: Package statement always the first statement in the program violation leads to CTE.

(ii) Using class name

```
Package K2;
Public class A
{
    void display()
    {
        sop("using class name");
    }
}
```

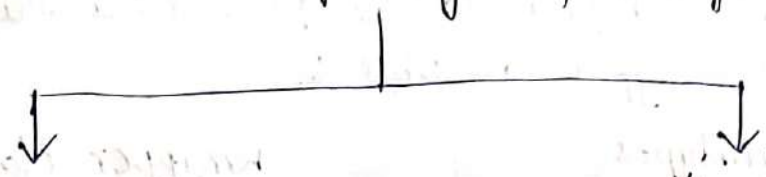
```
Package K1;
import K2.A;
class B
{
    P.S.V.m(String args[])
    {
        A a=new A();
        a.display();
    }
}
```

(iii) Fully Qualified Name.

```
Package k2;
Public class A
{
    void display()
    {
        sop("using class
            name");
    }
}
```

```
Package k1;
class B
{
    P.S.V.m(String args[])
    {
        K2.A a=new k2.A();
        a.display();
    }
}
```

Importing a package



Explicit (fully qualified name)

Ex: Java.util.array,
Java.util.arraylist

Implicit

* Import is mandatory

Ex: import package.*;
name;

Note:
Explicit importing is best because it takes less time to compile but in the real time we always going with implicit importing a package. Because a programer is unable to remember all class, objects and method names.

order of statements in a program.

1. Package stmts
2. Import stmts
3. Class stmts.

Note:
The no: of package stmts is more then its compile time is also more. But there is no difference in run time.

Wrapper class : wrapper classes provides a way to use Primitive datatypes as object.

Primitive datatypes		wrapper class
byte	———	Byte
Short	———	Short
int	———	Integer
long	———	Long
float	———	Float
double	———	Double
char		
boolean	———	Boolean

AutoBoxing

Converting a primitive datatype into an object is called is called an autoboxing.

UnBoxing.

Converting a wrapper class object to a primitive datatype is called UnBoxing.

Note:

we can referer the wrapper classes when we want to impact the primitive datatypes as objects.

Program for autoboxing.

```
class Auto
{ p.s.v.m (String args[])
  {
     int i=10;
     Integer j=1; //implicit
     Integer k= Integer.valueof(i); //explicit
     S.O.P("j");
     S.O.P("k");

  }}
```

Program for Onboxing

```
class Auto
{  p.s.v.m (string args[])
   {
      Integer i=new Integer(7);
      int j= i;  // Implicit
      int k=i.intvalue() // explicit
      SoP(j);
      SOP(k);
   }
}
```

Date
08/01/22

Exception Handling.

An exception is an abnormal condition that raises at runtime.

→ Exceptions are recoverable things (it gives alternative ways) where as errors non-recoverable things.

→ Exception raises at runtime. where as errors raise at compile time.

→ If a program raises an error we will get zero output. whereas if a program raises an exception atleast you can get partial output.

Ex:

```
class sample
{
    p.s.v.m (String args[])
    {
        int i,d=0;
        i=52/d;
        sop(i);
    }
}
```

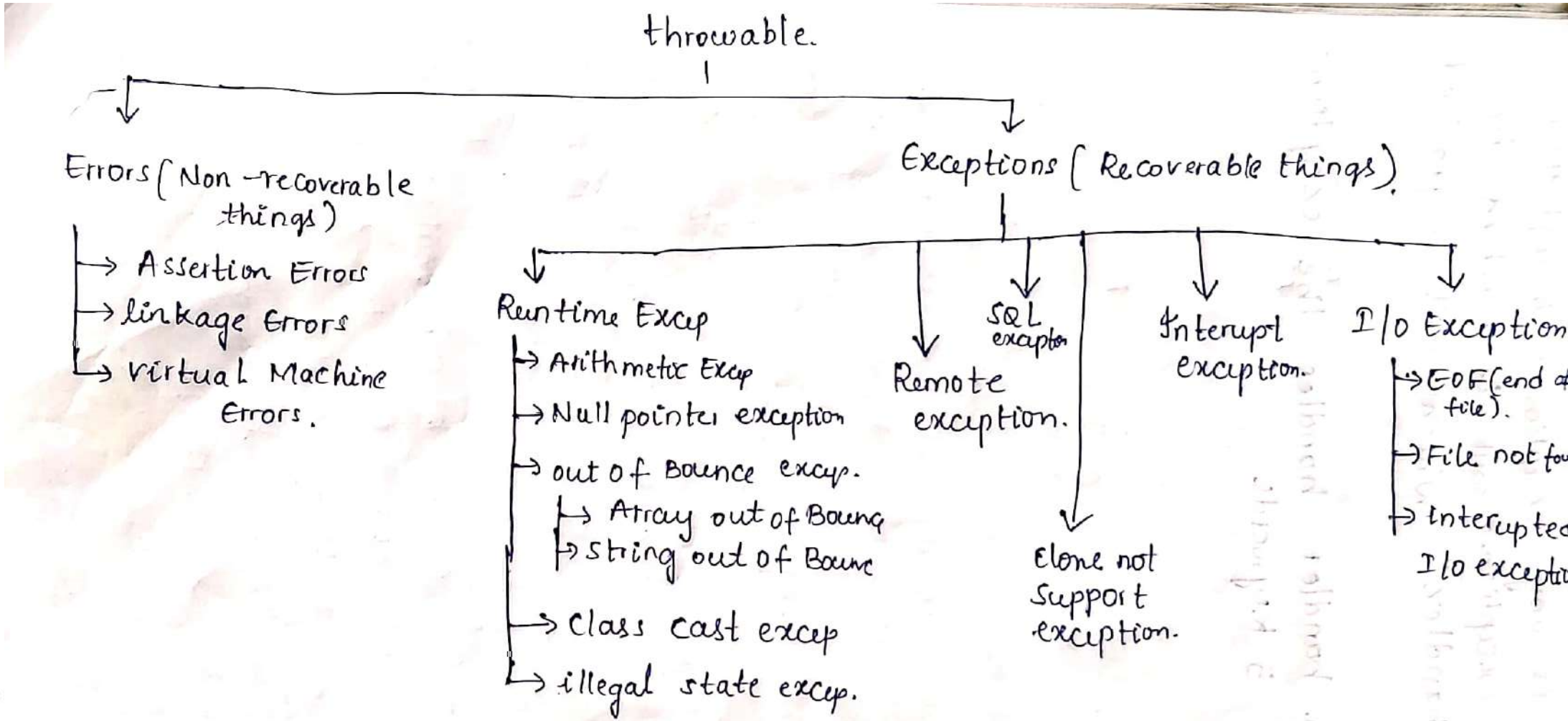
The above program compiles fine. Since there are no syntax errors But it raises runtime exceptions, by saying divide by zero

/ by zero : Arithmetic Exception.

→ If there is any exceptions in a program then that exceptions will be handled by "exception handler.", which will be provided by JVM.

→ Exception handler handles the exceptions by using 5 keywords.

try
catch
throw
throws
finally.



Syntax:

```
try
{
    // body
}
catch (exceptiontype e)
{
    // body
}
```

Note: If you feel an instruction or a piece of code may raise an exception then place that piece of code in try block.

Ex:

```
class sample
{
    P.S.V.m(String args[])
    {
        int d=0;
        try
        {
            i = 52/d;
        }
        catch (ArithemeticException ae)
        {
            SOP("It is divided by zero Problem");
        }
    }
}
```

O/P: It is divided by zero problem

Ruff code

```
SOP("stmt 1");
try
{
    SOP("stmt 2");
    SOP("stmt 3");
    SOP("stmt 4");
}
catch (Exception e)
{
    SOP("stmt 5");
    SOP("stmt 6");
}
SOP("stmt 7");
```

(i) No exceptions

o/p: 1, 2, 3, 4, 7.

Comment: If there is no exceptions in try block then compiler never goes execute the catch block.

(ii). If Exception at stmt 3 & corrcisponding catch is found.

o/p: 1, 2, 5, 6, 7.

comment

→ If exception raised in try block and corresponding catch is found then the stmts under the exceptions, stmt never compile and never executed.

(iii) If exception at stmt 3 & corresponding catch is not found

o/p: 1, 2, Default exception handler.

comment: If exception raised in try block and corresponding catch not found then it gives partial output (up to exception stmt) then default exception handler comes in to the picture.

(iv) If exception raised at stmt 6.

o/p:- 1, 2, 3, 4, 7.

comment: If there is no exceptions in try block then compiler never goes to catch block.

(v) Exception at stmt 3 & corresponding catch is found & exception at stmt 6.

o/p:- catch found → 7.
catch not found - 1, 2, 5 Default exception handler

comment:- If a try block raises an exception and corresponding catch is found again if there is an exception in catch block.

(a). corresponding catch found

(b) corresponded catch not found.

11/6/12

Multiple catches.

Every try block may contain more than one catch block. This is the central concept of multiple catch.

→ Each try block can raise only one exception even it may contains many exceptions and it executes its corresponding catch.

Note: At a time one try and one catch will be executed.

Ex:

```
Class Sample
{
    Public static void main(String args[])
    {
        int n,i ;
        {
          try
          {
            i = 52/n;
          }
          catch (ArithmeticException ae)
          {
              S.O.P("it is divided by n");
          }
          catch (ArithmeticException ae)
          {
              S.O.P("It is not divided by n nullpointer");
          }
    }
}
```

It is divided by n

Throw Keyword

→ In the try block JVM rises an exception and handled by user (catch block).

→ If user rises an exception then it has to handled by JVM. This can be done by using ~~pro-k~~ Throw Keywoid. It is a user defined Exception.

Syntax:

```
throw ThrowableInstance;
```

Ex:

```
throw new IOException;
```

→ An user defined exception can raised by using new keyword

```
static
{
  sop("stmt 1");
  throw new IOException;
  S.O.P("stmt 2"); // CTE.
}
sop("stmt 3");
catch (IOException e)
{
  S.o.P ("stmt 4");
}
```

→ This leads to CTE by saying unreachable stmts.

Reason: Here we are manually raising an exception, and we already know the stmts

under the exception stmt never compile and never execute, so if you give the stmts under the throw keyword leads to CTE.

Throws Keyword.

As method is also a part of program, method also may raise an exception.

→ If method raises an exception then we can handle by using throws Keyword.

Eg,
~~Finally Keyword~~.

Syntax:

Access Returntype methodname (Parameter list) throws Exceptionlist.

{

}

Ex:

```
Public void Today() throws throwable
{
}.
```

Note:

We can give more than one exception type to a method by using comma operator. If we

→ If you are unable to predict the type of exception that is raised by a

method then its better to give its hiraicy throwable.

Finally

Syntax:

```
try
{  //code

}
catch (Exception e)
{  //code

}
finally
{
   //code

}
```

Exi

```
sop("stmt 1");
try
{
   sop("2");
   sop("3");
   sop("4");
}
catch (Exception e)
{
   sop("5");
   sop("6");
}
finally
{
   sop("7");
}
sop("8");
```

(i) No Exception.

o/p: 1, 2, 3, 4, 7, 8.

Comment: If there is no exception in try block then compiler never executes the catch block

(ii). If exception at at 3 & corresponding catch found.

o/p:- 1, 2, 5, 6, 7, 8

Comment: If (corres) exception raised in try block and corresponding catch is found then stmt under exception stmt never compile and never executed.

valid syntax (or) formats of try catch

Java allows three valid formats of try, catch, finally.

1.
```
try
{
}
catch(Exceptione)
{
}
```

2.
```
try
{
}
finally
{
}
```

3.
```
try
{
}
catch (Exception e)
{
}
finally
{
}
```

Note:

Try, catch, finally is a single stmt. i,e. if you try to place any stmt b/w try and catch, catch and finally, try & finally, leads to CTE. By saying try, catch, finally is a single stmt.

Rethrowing an exception.

Sometimes we need to rethrow an exception in java.

→ If catch block cannot handle the particular exception that it has caught we can rethrow the exception.

Note:
The rethrow expression causes only when original thrown object can be rethrown

```
Public class allDemo.
{
  public static void main (String args[]) throws Throwable
  {
      try
      {
        test2();
      }
      catch (Exception e)
      {
        SO.P ("This is catch of test 2");
      }
  }
public static void test2() throws Throwable
{
  try
  {
    test1();
    throw new Exception();
  }
  catch(Exception e)
  {
```

```
        SOP("This the catch of test 1");
        throw e;
    }
}
public static void test1() throws Throwable
{   SOP("This is test 1");
}
}
```

op:
This is catch of test 2
This the catch of test 1.
This is test 1.

checked Exception.

If an exception is checked at compile time then it is called checked exception

→ Remote exception, SQL, clone not support, I/O and its subclasses, interupte exceptions are all checked exceptions.

Unchecked Exception.

If an exception is not checked at compile time then it is called unchecked exception.

→ Run time and its sub classes are unchecked exceptions.

→ checked exceptions are categorized in to two types.

1. Fully checked exceptions
2. Partially checked exceptions.

1. Fully checked Exceptions:
If the exception class and its subclasses are checked exceptions then that exception is called fully checked exceptions.

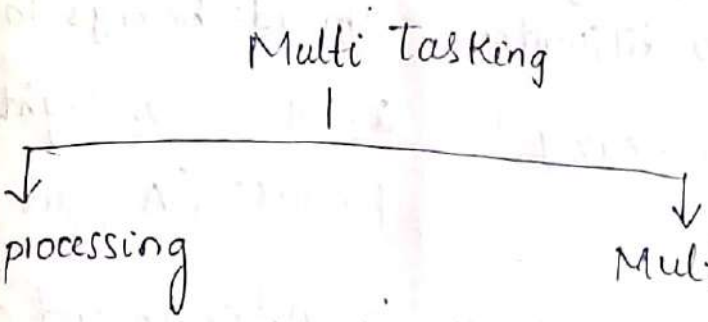
→ As I/O and its subclasses exceptions are checked exceptions then we can say that I/O exception is a fully checked exception.

2. Partially checked Exceptions.
If the exception class is checked and its subclass are unchecked then that exception is called partially chicked exception.

Date 20/01/22

UNIT-5.

Multi-threading



Multi Tasking.

Executing more than one task at a time is called multi tasking.

→ Multi-tasking done in two ways.

1. Multi processing.
2. Multi Threading.

Multi processing.

Executing more than one process simultaneously is called multi-processing.

Ex: performing - 1. Typing a document, printing a document, 3. playing songs, 4. copying data - - - - ..

Multi-Threading.

Executing more than one thread concurrently is called multi-threading.

→ The world's best example for multi threading is Animation.

Multiprocessing

1. Executing more than 1 process simultaneously, all process belongs to different prgms.
2. Process is a heavy weight
3. Inter process communication is difficult.
4. The context switching between the process is costly.
5. Ex:- Typing document, printing document, playing songs ----

Multithreading

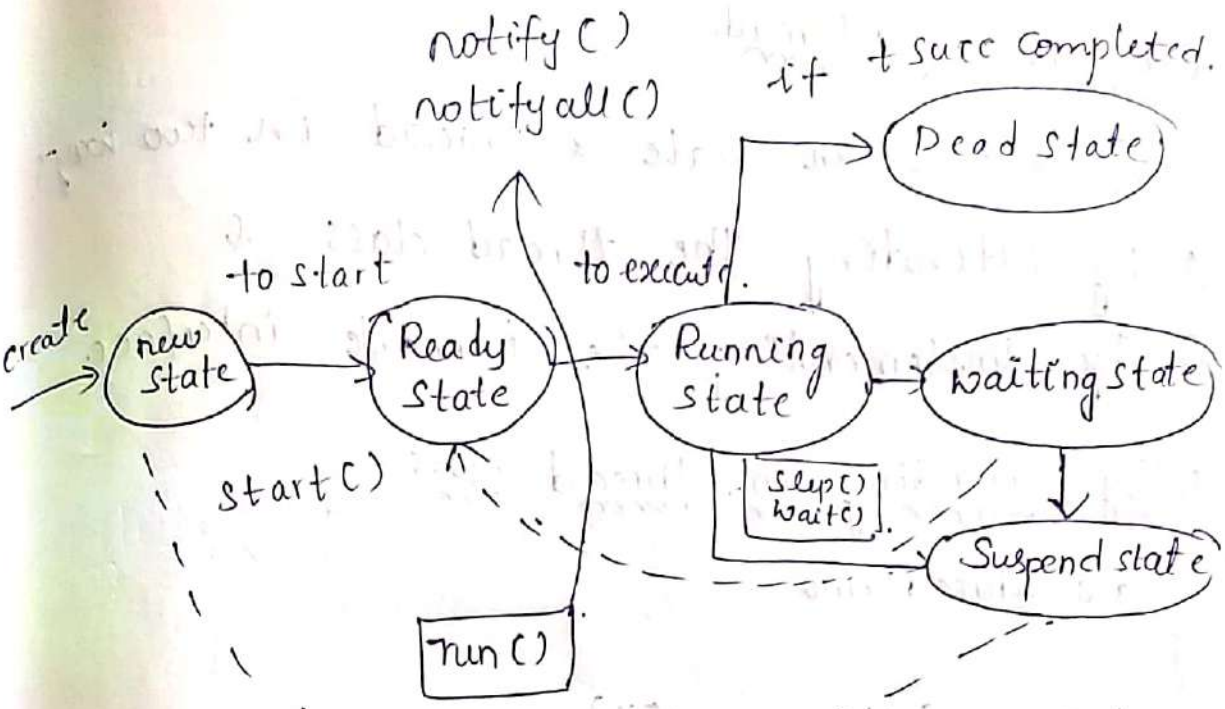
1. Executing more than one thread concurrently, all threads belongs to same prgm
2. It is a light weight process. (A part of process)
3. Inter thread communication is easier.
4. The context switching between the threads is cheaper.

Ex:- Animation.

Note:- The major advantage of using multi-threading is performance.

Life cycle of a thread:

Life cycle of a thread



to restart the thread.

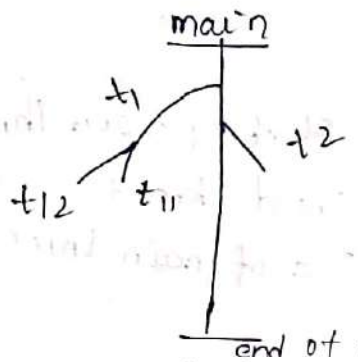
Note: Whenever public static void main execute by default JVM creates a thread called "main thread".

→ Main thread is capable of creating any number of child threads.

→ A child thread can again creates sub-child threads.

→ After completion of all child threads then only the main thread comes to end.

→ Default thread is main.



Date
27/01/22

Creation of a thread.

In java we can create a thread in two ways:

1. By extending the thread class &
2. By implementing the Runnable interface.

1. By extending the thread class

```
Class threadDemo
{
   P.S.V.m(String args[])
   {
      Sop("start of main thread");
      My thread t1 = new my thread();
      t1.start();
      Sop("End of main thread");
   }
}.
class my thread extends Thread
{
   public void run()
   {
      Sop("child Thread");
   }
}
```

o/p:- start of main thread
child thread
End of main thread.

(or). start of main thread
End of main thread
child thread.

Comment:- If main thread schedule first

Thread schedule

whenever a class is extended by in-built thread class then the thread schedular come to know that a new thread is going to be registered.

→ when a thr new thread get registered in a thread schedular it's completely the thread schedular's responsibility to schedule the thread.

→ The thread schedular schedules the thread based on the availability of the resources

→ The thread schedular is under control of the JVM.

Note: In multithreading there is no guarantee for the "order of output."

Note:
out of two ways after creating a thread the most preferable one implementing runnable interface.

29/01/22

2. By using Runnable Interface.

```
Class threadDemo
{
   P.S.V.m(String args[])
   {
      S.O.P(" Start of My thread ");
      My runnable r= new myrunnable();
      My thread t=new my thread(r);
      t.Start();
      S.O.P(" End of M.T ");
   }
}
Class mythread implements Runnable
{
   public void run()
   {
      S.O.P(" Child thread");
   }
}
```

Thread priority.

Java allow to give our own priority to Schedule the threads. Basically we have three types of thread priorities.

MIN_PRIORITY - 1
NORM_PRIORITY - 5
MAX_PRIORITY - 10

→ To set the priority of threads allows the) a range of 1 to 10. violation leads to CTE by saying illegal argument.

Note:
The default priority is normal priority.

Getter & Setter Methods

Java allows to set the name and priority to a thread instead of default. This can be done by using setter methods.

→ Public void setName (String name);
Ex: t1. setName ("Today");

→ Public void setPriority (int value);
Ex: t1. setPriority (7);

→ Public String getName ();
Ex: String S1 = t1. getName ();

Public int getPriority ();
Ex: int k = t1. getPriority ();

```
Class threadPriorityDemo
{
    P.S.V.m(String args[])
    {
        Mythread t1=new mythread();
        Mythread t2=new mythread();
        Mythread t3=new mythread();
        t1.setName("Thread 1");
        t1.setPriority(1);
        t2.setName("Thread 2");
        t2.setPriority(10);
        t3.setName("Thread 3");
        t3.setPriority(7);
        t1.start();
        t2.start();
        t3.start();
    }
}
class mythread extends Thread
{
    Public void run()
    {
        S.O.P(This getName());
        S.O.P(This.getPriority());
    }
}
```