Introduction to Software

↓

collection of programs

↓

set of instruction

Software

↙ ↘

System software

→ physical hard drives

popular lang

C, C++

Application software

↓

services to end users

↓

1. Desktop application (or)

standalone application

→ Runs on single device

→ Cannot share results

2. Application application (or) Internet based application

→ Runs on single device

→ Share results on multiple device using Internet

→ Ex- Calculator

2. Internet application (.net)

Web appl
→ Common to everyone
→ Ex:- Sign in page of gmail, fb, ig, etc.

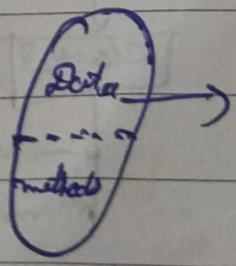
Enterprise appl
→ Limited to some users
→ Ex:-

JAVA is used to develop internet based application

Introduction to OOPS

To hold client info in form of objects

Objects < Data, methods



Data → variables (credentials of user)

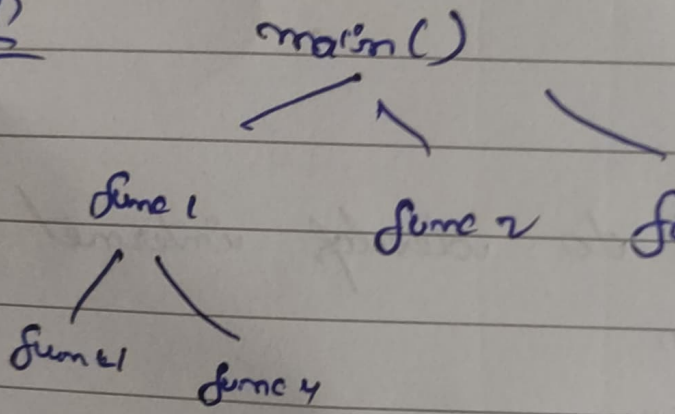
1001 → unique address for security

Advantage

→ Code reusability
→ Easily to identify errors

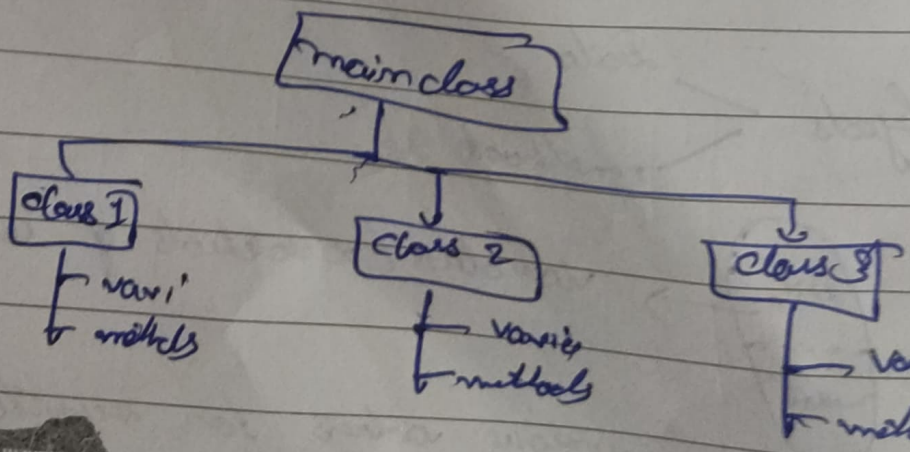
→ POP vs OOPS

POP

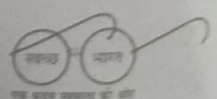


→ Recreation, reimplementation is difficult

OOPS



→ Recreation, reimplementation is easy



the Java programmer cannot overload any operator.

1.4 PROCEDURAL LANGUAGE VS OOP

Table 1.1 highlights some of the major differences between procedural and object-oriented programming languages.

Table 1.1 Procedural Language vs OOP

Procedural Language	OOP
• Separate data from functions that operate on them.	• Encapsulate data and methods in a class.
• Not suitable for defining abstract types.	• Suitable for defining abstract types.
• Debugging is difficult.	• Debugging is easier.
• Difficult to implement change.	• Easier to manage and implement change.
• Not suitable for larger programs and applications.	• Suitable for larger programs and applications.
• Analysis and design not so easy.	• Analysis and design made easier.
• Faster.	• Slower.
• Less flexible.	• Highly flexible.
• Data and procedure based.	• Object oriented.
• Less reusable.	• More reusable.
• Only data and procedures are there.	• Inheritance, encapsulation, and polymorphism are the key features.
• Use top-down approach.	• Use bottom-up approach.
• Only a function call another.	• Object communication is there.
• Example: C, Basic, FORTRAN.	• Example: JAVA, C++, VB.NET, C#.NET.

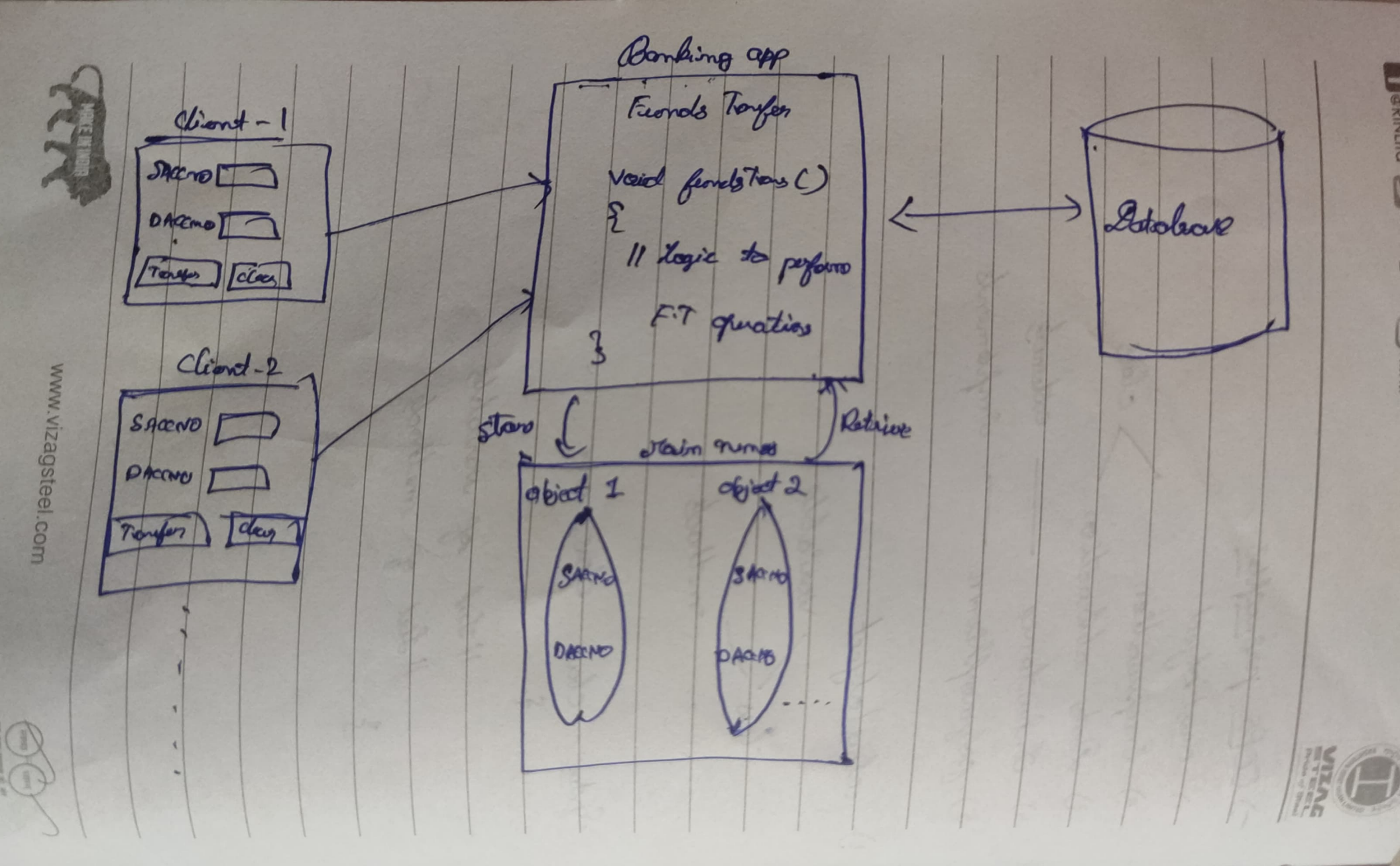
→ OOPS principles

1) class/object
2) Encapsulation
3) Data abstraction } class
4) Inheritance —— extends
5) polymorphism —— implements

1) class/object

class → variables
class → methods

Syntax class class_name
{
// List of variables
// List of methods
}



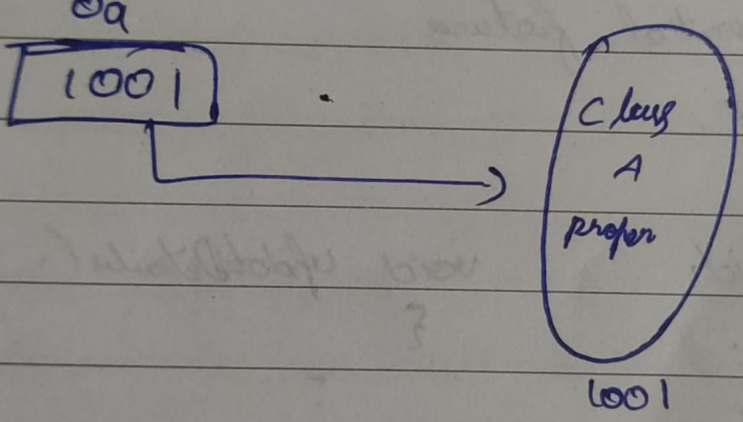
Object creation Syntax

In java we can create objects in 2 ways:

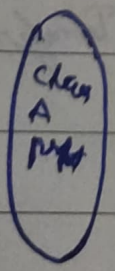
1) Referenced object → classname objref = new classname();
2) Unreferenced object → new classname

Eg:-

1) A aa = new A();



2) new A();



→ JVM — Java Virtual Machine

↓

Garbage collector

2) Encapsulation

Combining data (variables) and behaviour (methods) in a single unit is called encapsulation.

In java class keyword is used to achieve encapsulation

3) Data Abstraction

Hiding non-essential features

Unhiding essential features

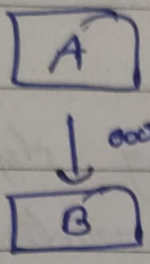
```
class customer
{
    int customer_id;
    String name;
    float amount;
    double mobileno.
    String address;
    int balance

    void updateDetails(...)
    {
        ...
    }

    void fundsTransfer(..)
    {
        ...
    }
```

4) Inheritance

In java we can achieve inheritance through "extends" keyword



Parent/Super/Base

child/derived/sub

→ Memory management

→ Code reusability

6) Polymorphism

1. Static polymorphism → overloading
2. Dynamic polymorphism → overriding

1. Static polymorphism

Whatever the method verified at compile time and the same method is executed at runtime is called static.

In java we are achieving static polymorphism through method overloading

```
class A
{
    void f1(int)
    void f1(float)
}
```

2. Dynamic polymorphism

Verifying the super class method at compile time and executing the derived class same method in runtime is called dynamic polymorphism.

In java we are achieving dynamic polymorphism through method overriding.

```
class A                                    SBI 1.0
{
    void f1();
}
class B extends A                          SBI 1.1
{
    void f1();
}
```

Java features

1. Simple — S
2. Object oriented — O
3. platform independent — P
4. Architectural Neutral — A
5. Robust — R
6. Multithreaded — M
7. High performance — H
8. Dynamics — D
9. Distributed — D
10. Secured — S

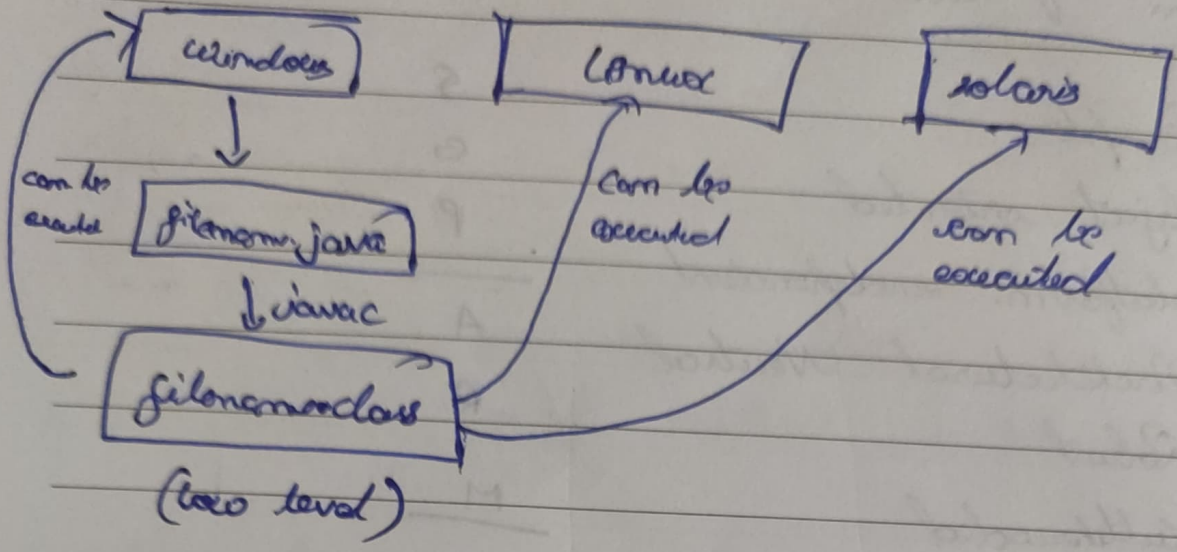
1. Simple

Java is very simple programming language because of : -

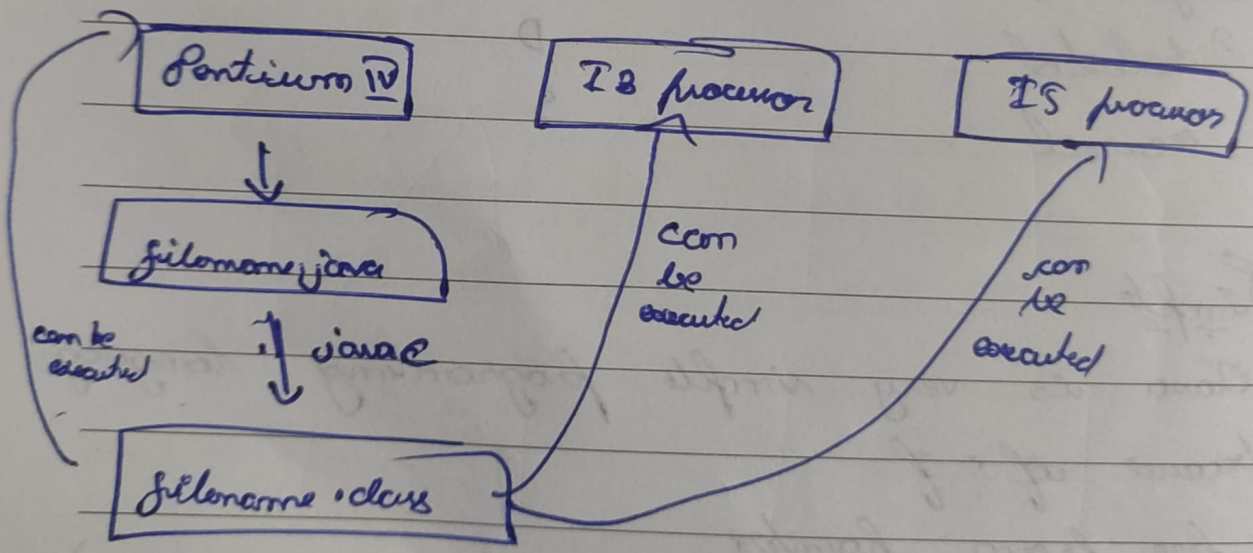
1) free from pointers
2) Huge API (Application program Interface)

Java contains more than 1 lakh predefined classes. All the predefined packages are in API by oracle

3) Platform independent



4) Architectural Neutral



5. Robust (very strong)

In java we are achieving exception handling through "try - catch".

throw is not there, since JVM knows where to throw.

6. Multithreaded :

The same program of same application provide services to multiple clients simultaneously is called as multithreading

In client point of view, thread is a request

7. High_performance

Java is a high performance lang due to multithreading, exception handling and Garbage collection. (G.C)

The G.C. is a predefined method that will run in the background of every java program implicitly and this will deallocates the unused memory.

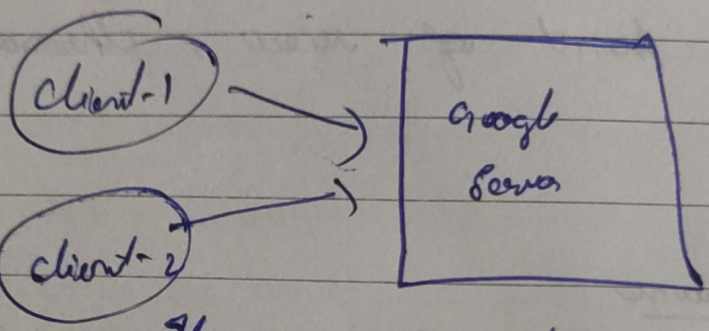
8. Dynamic

Java purely supports dynamic memory allocation

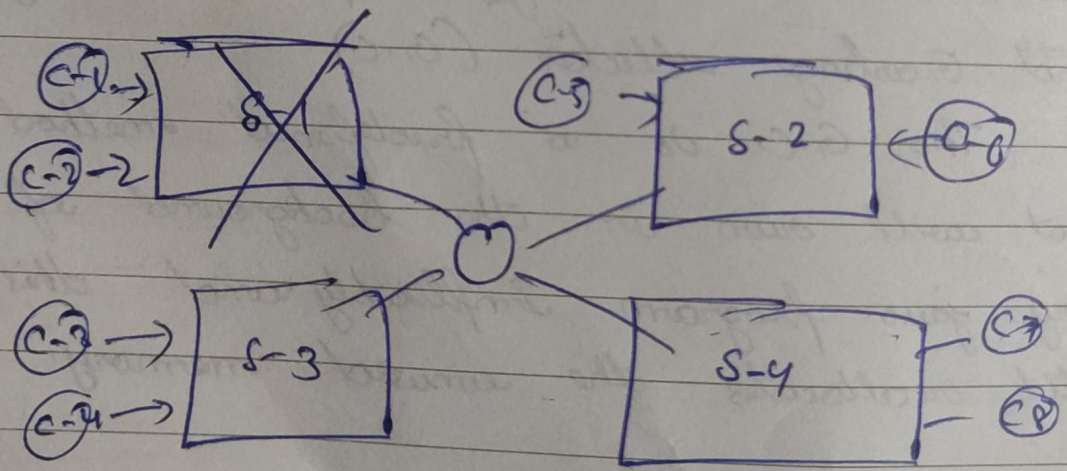
9. Distributer

Java supports 2 types of architecture

1. client - server architecture
2. distributed architecture



If server is down, all clients must wait



If server 1 is down, then it goes to server three

Drawback :- Cost is very high

10. Secured

→ Virus free programming language

→ Suppots both internal & external security

↓

oops-OWN

Encryption

Decryption

using third party apps

Naming Conventions in Java

→ Every ~~letter~~ word of

Variable / Identifier

Except first word, from 2nd word onwards the starting letter must be capital

Ex:- studentName

Method name

Except first word, from 2nd onwards starting letter must be capital

Eg: fundsTransfer();

Class name

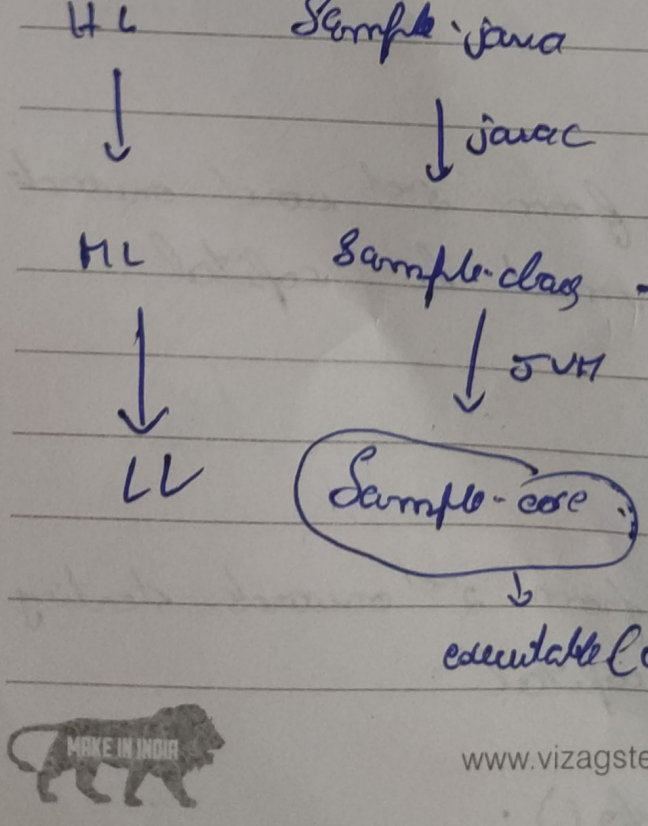
Every word starting letter must be capital

Eg:- Employee Details

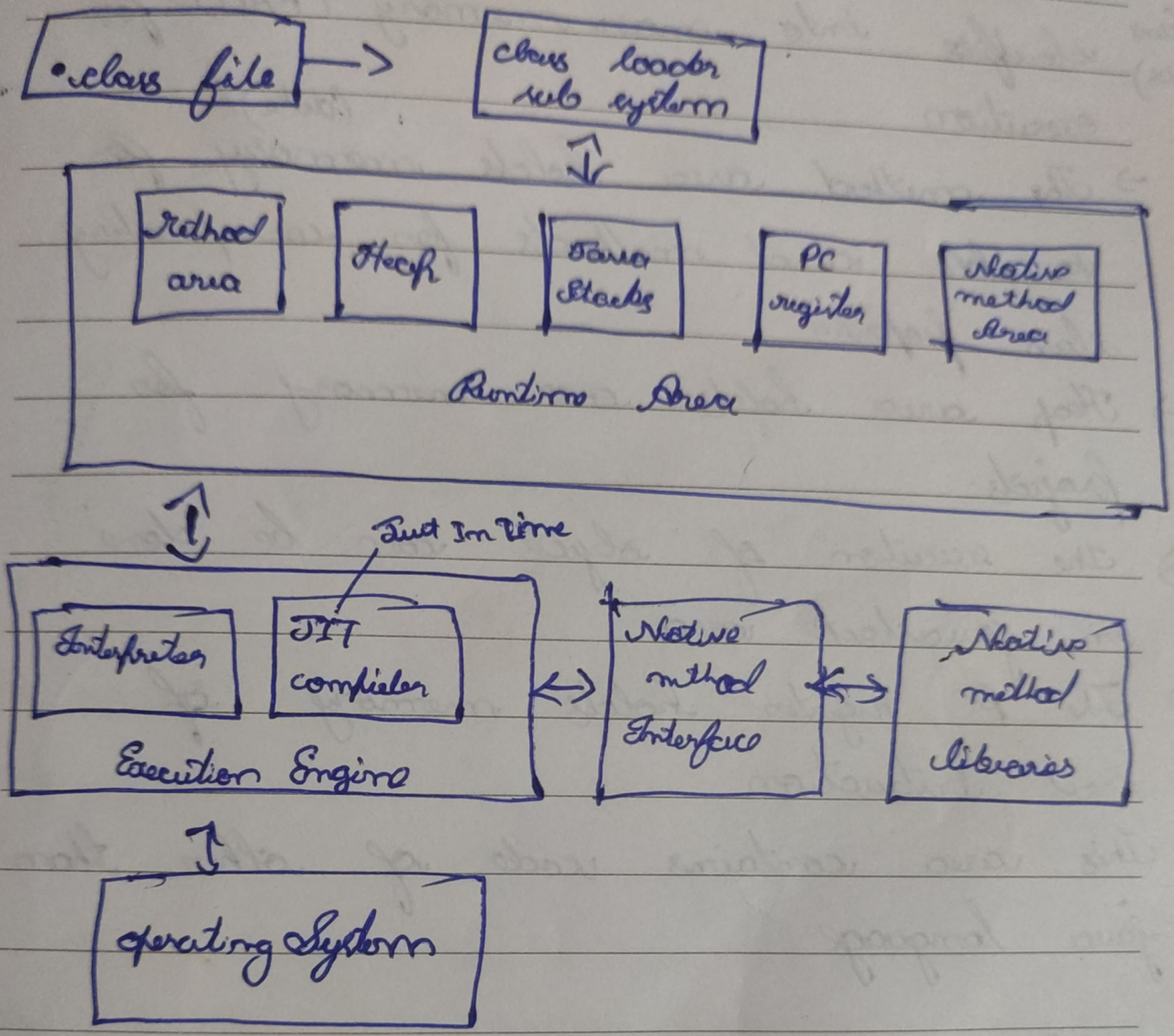
Employee_Details

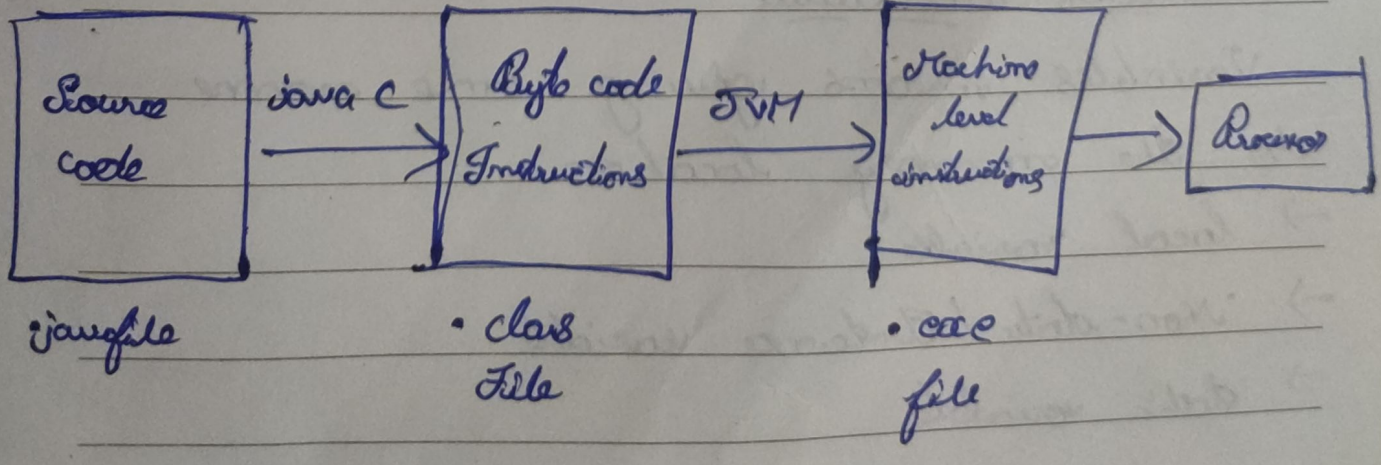
Syntax

```
class MainClass
{
    // list of variable
    // list of methods
}
```

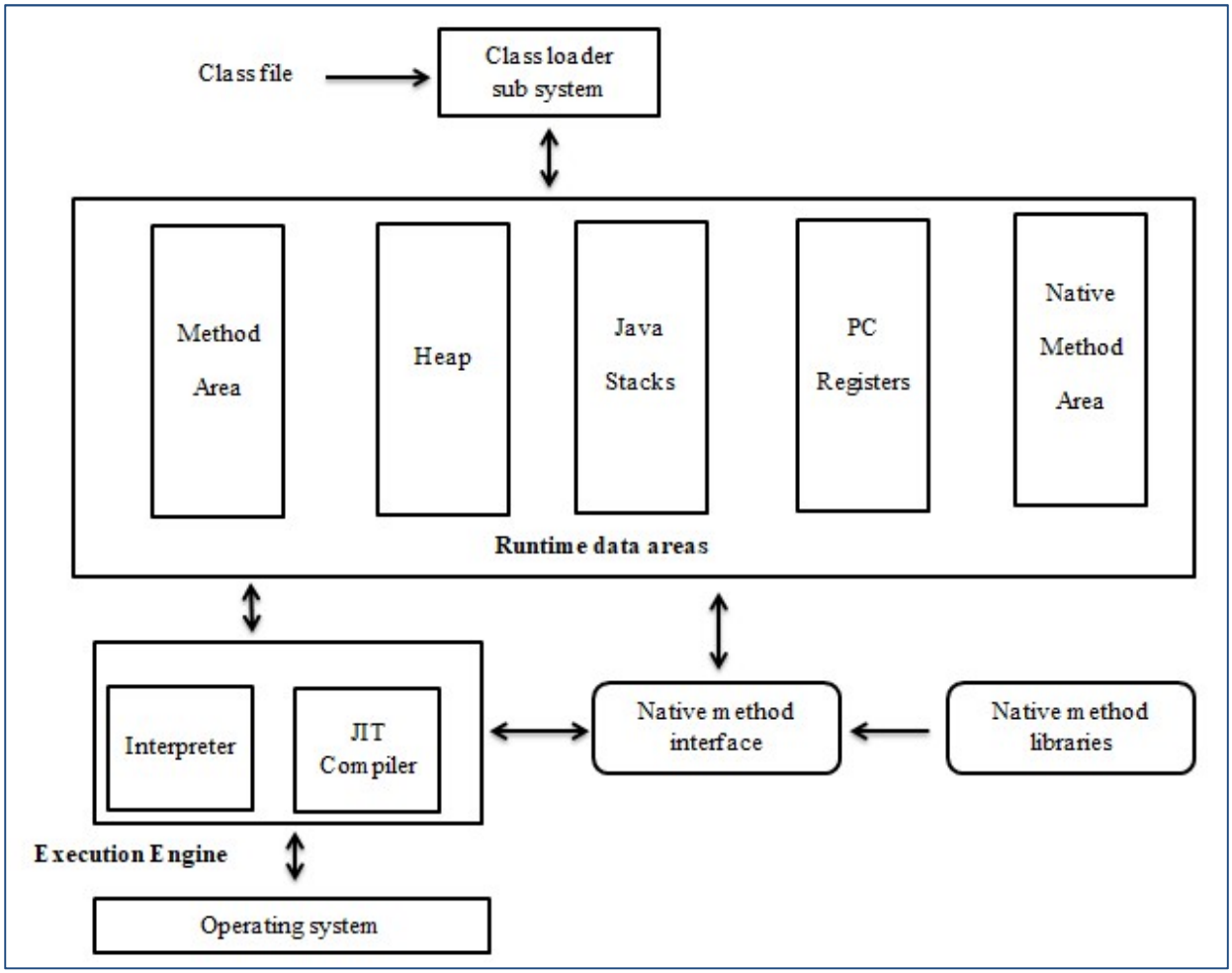


JVM architecture





- First **'javac'** converts the source code instructions in to **'byte code'** and after successful compilation a new **'.class'** file will be created.
- Now the **'.class'** file is given to the JVM as an input and it converts in to machine level instructions.
- The **architecture** of JVM is as follows:



7.1 Class Loader Sub System:

- In JVM, there is a module called "***class loader sub system***", which performs the following functions:
  1) First, it loads the "**.class**" file into main memory for all the properties of the class.
  2) Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
  3) If the byte code instructions are proper, then it allocates necessary memory to execute the program.

7.2 Method area:

- Method area is the method block, in which memory is allocated for the code of the variables and methods in the java program.

7.3 Heap:

- This is the area where objects are created. In this area, memory is allocated for the 'Instance Variables' in the form of objects.

7.4 PC (Program Counter) registers:

- These are the registers, which contains memory address of the instructions of the methods.

7.5 Java Stacks:

- Java stacks are memory area where Java methods are executed.
- It also allocates the memory for 'local variables' and 'object references'.
- JVM uses a separate thread to execute each method.

7.6 Native Method area:

- In which memory is allocated for the native methods.
- These methods contain other than java language code like c, C++, .Net. Etc.
- To execute native method, generally native method libraries are required. These header files are located and connected by the JVM by using 'Native method interface'.

7.7 Execution Engine:

- It contains two components named as **Just in Time (JIT) Compiler** and **Interpreter**.
- These components are responsible for converting the byte instructions into machine code so that the processor will execute them.
- Most of the cases JVM use both JIT compiler and interpreter simultaneously to convert the byte code. This technique is called "**Adaptive Optimizer**".
- **Interpreter** is responsible to convert one by one 'byte code' instructions into 'machine level instructions' and it always comes into the picture while sending the **first request** to java program.
- **JIT Compiler** always appears from the **second request** onwards; it will collect all the 'machine level' instructions that are already converted by the interpreter and gives to processor.
- The main **advantage** with JIT compiler is it will increase the execution speed of the program.

8. Difference between C++ and JAVA

C++	JAVA
1) C++ is platform dependent.	1) JAVA is platform independent.
2) It is mainly used to develop system softwares.	2) It is mainly used to develop application softwares.
3) It is not a purely object oriented programming language, since it is possible to write C++ programs without using class or object.	3) It is a purely object oriented programming language, since it is not possible to write a java program without using class or object.
4) Pointers are available in C++.	4) Java is free of pointers creation.
5) Allocation and deallocating memory is the responsibility of the programmer.	5) Allocation and deallocating memory will be taken care of JVM.
6) C++ has goto statement.	6) C++ doesn't have goto statement.
7) Automatic casting is available in C++.	7) In some cases, implicit casting is available. But it is advisable to the programmer should use casting wherever required.
8) Multiple Inheritance feature is available in C++.	8) Java doesn't support Multiple Inheritance in class level. It can be achieved by interfaces.
9) Operator overloading is available in C++.	9) It is not available in java.
10) #define, typedef and header files are available in C++.	10) These are not available in java.
11) C++ uses compiler only.	11) Java uses compiler and interpreter both.
12) C++ supports constructors and destructors.	12) Java supports constructors only.

⇒ The class loader subsystem loads (class files) classfiles into main memory (RAM) for execution

→ The method area holds memory (data) for variable and methods for corresponding class properties

→ Heap area holds memory for projects.

→ The execution of objects can be done in javastack area.

→ The PC register holds memory of next instruction

→ This area contains code of other than java language

Variables in Java

Variables means giving some name to the memory location.

→ local variable

→ Non-static / Instance variable

→ static variable

→ Local Variable :- Declaring within method

Syntax

```
class A
{
    void f1()
    {
        int x;
        System.out.println(x);
    }
}
```

Not valid

We should declare some value to x

Java does not have garbage value

In java, it is mandatory to initial local variable. JVM doesn't allocate default values to the local variable

```
class A
{
    void f1() // void f1(int x)
    {
        int x=10;
        SOP(x);
    }
}
```

The lifetime of local variable is within method only

-) Non-static / Instance

If any variable is declared outside method and inside class without static keyword is called as instance variable

Syntax:-

```
class A
{                    → Instance variable
    int a;
    void f1()
    {
        S.O.P(a);
    }
}
```

} Valid

→ In java, JVM allocates default value to the instance variable if it is not initialized

-) Static Variable

If any variable is created outside the methods and inside the class with "static" keyword is called as static variable

Syntax

```
class A
{
static int x;
void f1()
{
   S.O.P(x);
}
}
```

```
class B
{
S.O.P(A.x);
```

↳ calling from A

Datatype	Default value
byte, short, int, long	0
float, double	0.0
char	_ (blank space)
String	NULL

→ JVM doesnot allocates default values to the local variable.
Local variables must be initialized

Datatypes in Java

Types of input

- Numerical data
  - Integer — Eg:- -10, 10, 20
  - Floating point — Eg:- 20.3, 30.7
- character type data
  - Character — Eg:- 'c', 'd'
  - String — Eg:- "cat"

(Keyword)
boolean — 1, 0

Datatype

- Primitive
  - Integer
  - Floating
  - Character
- Referenced
  - String

i) Integer :- Default value = 0

D.T	Size	Range
Byte	1B (8 bits)	-2^7 to $+2^7-1$
Short	2B (16 bits)	-2^{15} to $+2^{15}-1$
int	4B (32 bits)	-2^{31} to $+2^{31}-1$
long	8B (64 bits)	-2^{63} to $+2^{63}-1$

(ii) Floating point

D.t	Size	Range
a) float	4B (32 bits)	-2^{31} to $+2^{31}-1$
a) double	8B (64 bits)	-2^{63} to $+2^{63}-1$

float f = 3.14(f); → if not then f, JVM prefers more to double so it creates 8B we will get error.

(iii) Character:- default value is blank ( ) Size is 2B.

2) Referenced DT

There is no fixed memory for ref dt In java we are achieving referenced using string. It takes memory as much as RAM size.

Identifiers

By using identifiers, we can identify variable, method, class name, object reference etc.

Every identifier name can be a combination of alphanumerics

Eg:- for variable

age = 20

-age = 20;

student_age = 20

$age = 20;

123age = 20; → Illegal

age123 = 20;

a# = 20;

Same for object

Same for class name, but starting letter should be capital

Allocating memory

int + int = int

int + float = float

float + float = float

float % float = int

int % int = int

7. always returns int d.T
JVM gives priority to highest d.T

Prog to show default value

```
class A
{
  boolean bl;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  char c;
  String str;
  void f1()
  {
  System.out.println(bl + " " + b + " " + s + " " + i + " " + l + " " + f + " " + d + " " + c + " " + str);
  }
}
class Default
{
  public static void main(String arg)
  {
```

```
    new A().s1();
  }
}
```

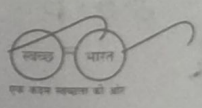
Program to bn diff between referenced obj & unreferenced obj

```
class A
{
int rollno;
String name;
float marks;
void setDetails()
{
  rollno = 123;
  name = "CSE-2";
  marks = 90.5f;
}
void display()
{
  s.o.p(rollno);
  s.o.p(name);
  s.o.p(marks);
}
```


```
class Default
{
    public static void main()
    {
        A aa = new A();
        aa.setDetails();
        aa.display();
        S.O.P("-----");
        aa.display();
        S.O.P("------");
        new A().setDetails();
        new A().display();
        S.O.P("------");
        aa.display();
    }
}
```

Output

```
123
CSE-2
90.5
-------
123
CSE-2
90.5
-------
0
null
0.0
-------
123
CSE-2
90.5
```



Operators in java

In java operator is nothing but a symbol can be used on unary, binary and ternary operators.

The types of operators can be supported by java are as follows

1. Unary operator
2. Arthimatic operator
3. Relational operator
4. Shift operator
5. logical operator
6. Bitwise operator
7. Assignment operator
8. Ternary operator

1. Unary operator

```
class Default
{
    public static void main (String arg [])
    {
        boolean bl = false;
```

int a=10;
int b=20;
S.O.P (a+b);
S.O.P (!b);
S.O.P (a++ + ++a);
S.O.P (b++ + --b);
S.O.P (a++ - ++b);
S.O.P (-a);
S.O.P (-b);
}
}

output
1020

S.O.P (a+" "+b);

2. Prog on Arithmetic operator

class Default
{
public static void main (String arg[])
{
int a=10;
int b=20;
S.O.P ("a+b=" + (a+b));
S.O.P ("a-b=" + (a-b));
S.O.P ("a*b=" + (a*b));
S.O.P ("a/b=" + (a/b));
S.O.P ("a%b=" + (a%b));

3. Relational operator

```
class Defaut
{
public static void main(String arg[])
{
  int a=10, b=20;
  S.O.P("a<b=" + (a<b));
  S.O.P("a<=b" + (a-b));
  S.O.P("a>b" + (a>b));
  S.O.P("a>=b" + (a>=b));
  S.O.P("a!=b" + (a!=b));
}
}
```

4. Shift operator

Left shift (<<) Right shift (>>)

Eg: 10 << 2

```
8 4 2 1
1 0 1 0   } Internal conversions
32 16 8 4 2 1
1 0 1 0 0 0 = 40
```

Formula :- $10 \times 2^2 = 40$

value × 2^(no. of shift bits)

```
class Default
{
    public static void main(String arg[])
    {
        S.O.P(10<<2)
        S.O.P(20<<3)
    }
}
```

Output
40
160

Right shift operator

10 >>> 2

Formula

8 4 2 1

1 0 | 1 0 → X

0 0 + 0 = 2

⇒ value / $2^{\text{no of bits}}$

⇒ $\frac{10}{2^2} = 2$

Logical & Bitwise Operator

Logical AND :- &&

Bitwise AND :- &

Output type is boolean

If 1st input is wrong → this is not considered

a<b && a>b

(a<b) & (a>b)

Even though 1st one is wrong, 2nd is considered Hence both are taken

Logical OR :- ||

~~Logical~~

Bitwise OR : |

For logical AND or OR, there should be expression or relational operator to compare which means boolean input. Variables & Constant cannot be compared

For Bitwise it is not like that, O.S will take care.

9. Ternary operator

```
int a=10;
int b=20;
int max = (a>b) ? a:b;
S.O.P (max);
```

Typecasting in Java

Java supports 2 types of typecasting

1. Implicit type casting (or) widening
2. Explicit type conversion (or) Narrowing

1. Implicit Type casting (JVM will T.C.)

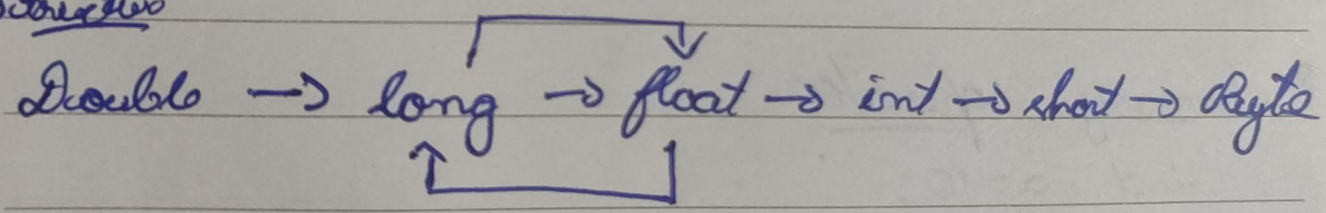
Structure

Byte → short → int → long → float → Double

low to high

2. Explicit Type casting

Structure



Control structure in java

- Conditional structure
  - if
  - if else
  - nested if
  - switch
- Looping Study
  - Range-based loop

for-each

Syntax

datatype variablename : array name

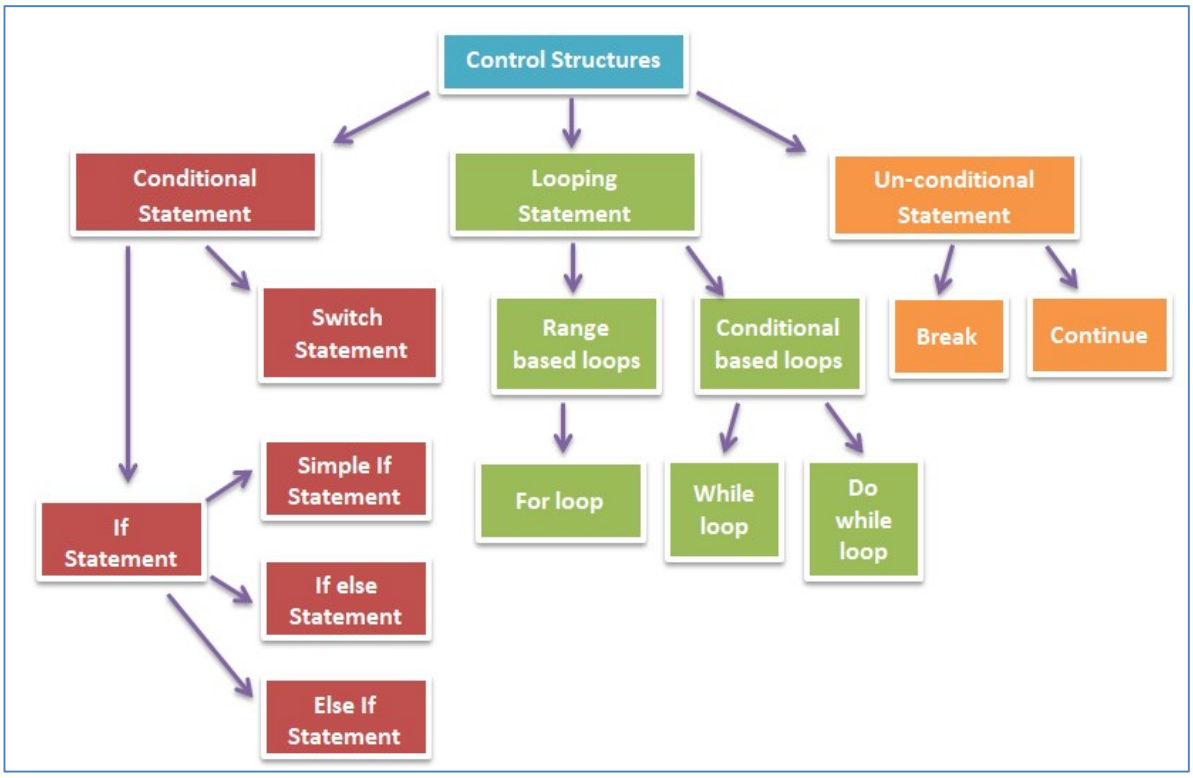
```
class Sample
{
  public static void main(String arg[])
  {
    int a[] = {10, 20, 30, 40, 50, 60}
    for (int x : a)
    {
      S.O.P(x);
    }
  }
}
```

```
for (int x : a)
{
  if (x == 50)   // we have to use x only
  {
    break;
  }
  S.O.P(x);
}
```

OUTPUT:
Double value 100.04
Long value 100
Int value 100

18. Control Structures in Java

- Control Structures are used to **control the execution flow of logic**, in other words it can be used to achieve random execution.
- These are categorized into following:



18.1 Conditional statements:

- These are used to execute one or more logical instructions based on the condition, these statements are executed only once.

18.1.1 If Statement:-

- The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in java.
  - if statement
  - if-else statement
  - else-if ladder
  - nested if statement

Simple If statement:

- It can be used by performing any operation by checking each and every condition. It executes the logic if the condition is true.

Syntax:

```
if(condition)
{
//code to be executed
}
```

Example Program:

```
class IfExample
{
public static void main(String[] args)
{
   int age=20;
   if(age>18)
  {
      System.out.print("Age is greater than 18");
   }
}
}
```

OUTPUT:

Age is greater than 18

If else statement:

- The Java if-else statement also tests the condition. It executes the code if condition is true otherwise else block is executed.
- In real time it can be used whenever we want to execute one logic among two logics.

Syntax:

```
if(condition)
{
//code if condition is true
}
else
{
//code if condition is false
}
```

Example Program:

```
class IfElseExample
{
public static void main(String[] args)
{
   int number=13;
   if(number%2==0)
{
      System.out.println("even number");
   }
else
  {
      System.out.println("odd number");
   }
}
}
```

OUTPUT:

odd number

else-if statement:

- In real time it can be used to execute only one logic among multiple logics.

Syntax:

```
if(condition1)
{
//code to be executed if condition1 is true
}
else if(condition2)
{
//code to be executed if condition2 is true
}
else if(condition3)
{
//code to be executed if condition3 is true
}
...
else
{
//code to be executed if all the conditions are false
}
```

Example Program:

```
class IfElseIfExample
{
public static void main(String[] args)
{
   int marks=65;
if(marks<50){
      System.out.println("fail");
   }
   else if(marks>=50 && marks<60)
{
      System.out.println("D grade");
   }
   else if(marks>=60 && marks<70)
{
      System.out.println("C grade");
   }
   else if(marks>=70 && marks<80)
{
      System.out.println("B grade");
}
   else if(marks>=80 && marks<90)
{
      System.out.println("A grade");
   }
else if(marks>=90 && marks<100)
```

```
{
        System.out.println("A+ grade");
}
   else
{
        System.out.println("Invalid!");
}
}
}
```

OUTPUT:

c grade

Nested-if statement:

- In real time, if one 'if-statement' is existing inside another 'if-statement' is known as Nested-if statement.
- It can be used to improve the efficiency of simple if statement, if-else statement and else-if statement.

Syntax:

```
if(condtion1)
{
  // executes when the condtion1 is true
  ifcondtion2)
   {
    // executes when the condtion2 is true
   }
}
```

Example Program:

```
public class Test
{
 public static void main(String args[])
{
     int x = 30;
     int y = 10;
     if( x == 30 ) {
       if( y == 10 ) {
          System.out.print("X = 30 and Y = 10");
       }
     }
  }
}
```

OUTPUT:

X = 30 and Y = 10

18.1.2 Switch Statement:

- It is same as 'else-if statement' used to executes only one logic among multiple logics.
- The only difference between switch and 'else-if' is in performance. Switch is better than else-if statement.
- In 'else-if' statement every condition should be verified even to execute last block of statements, it leads to increase the time consuming process and this problem can overcome using 'switch' statement.

Syntax:

```
switch(expression)
{
        case value1:
         //code to be executed;
         break;  //optional
        case value2:
         //code to be executed;
         break;  //optional
        .....

        default:
        //code to be executed if all cases are not matched;
}
```

- 'Java compiler' will memorize all the case label name values and it will help to JVM to identify the case blocks at runtime.
- If no label name is matching with input choice then default block will be executed, default bloack can exist anywhere in the switch statement.

Example Program 1:

```
public class SwitchExample
{
public static void main(String[] args)
{
   int number=20;
   switch(number){
   case 10: System.out.println("10");break;
   case 20: System.out.println("20");break;
   case 30: System.out.println("30");break;
   default:System.out.println("Not in 10, 20 or 30");
   }
}
}
```

OUTPUT:

20

Example Program 2:

```
public class SwitchExample2
{
public static void main(String[] args)
{
   int number=20;
```

```
    switch(number){
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

OUTPUT:

20
30
Not in 10, 20 or 30

Note: All conditional statements are executed only once.

18.2 Looping statements

- In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.
    - for loop
    - while loop
    - do-while loop

18.2.1for loop:

- These are used to execute repeatedly based on given range, it can be achieved using for loop.
- The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed then it is recommended to use for loop.
- There are three types of for loops in java.
    - Simple For Loop
    - Nested For Loop
    - For-each or Enhanced For Loop
    - Labeled For Loop

Simple for Loop:

- The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

```
for(initialization; condition; incr/decr)
{
//code to be executed
}
```

Example Program on for loop:

```
public class ForExample
{
public static void main(String[] args)
{
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
} }
```

OUTPUT:

```
1
2
3
4
5
6
7
8
9
10
```

Nested loop:

- If the 'for loop' is existed within another '***for***' loop are called as Nested for-loops.

Syntax:

```
for(initialization;condition;incr/decr)
{
        for(initialization;condition;incr/decr)
        {
        //code to be inner for loop
        }
//code to be outer for loop
}
```

Example Program on nested for loop:

```
class NestedForLoop
 {
  public static void main(String[] args)
  {
    for (int i = 1; i <= 5; ++i)
   {
       System.out.println("Outer loop iteration " + i);
       for (int j = 1; j <=2; ++j)
          {
          System.out.println("i = " + i + "; j = " + j);
       }
    }
  }
}
```

OUTPUT:

```
Outer loop iteration 1
i = 1; j = 1
i = 1; j = 2
Outer loop iteration 2
i = 2; j = 1
i = 2; j = 2
Outer loop iteration 3
```

```
i = 3; j = 1
i = 3; j = 2
Outer loop iteration 4
i = 4; j = 1
i = 4; j = 2
Outer loop iteration 5
i = 5; j = 1
i = 5; j = 2
```

for-each loop:

- The for-each loop is used to traverse array or group of elements in java.
- It is easier to use than simple for loop because we don't need to increment value and use subscript notation.
- It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```
for(Datatype var:array)
{
//code to be executed
}
```

Example Program on 'for-each' loop:

```
public class ForEachExample
{
public static void main(String[] args) {
   int arr[]={12,23,44,56,78};
   for(int i:arr){
      System.out.println(i);
   }
}
}
```

OUTPUT:

```
12
23
44
56
78
```

Java Labeled For Loop:

- We can have name of each for loop. To do, we use label before the for-loop.
- It is useful if we have nested for loop so that we can break/continue specific for loop.
- Normally, break and continue keywords breaks/continues the inner most for loop only.

Syntax:

```
labelname:
for(initialization;condition;incr/decr)
{
//code to be executed
}
```

Example Program 1 on labeled for loop:

```
public class LabeledForExample
 {
public static void main(String[] args)
 {
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break aa;
               }
               System.out.println(i+" "+j);
            }
      }
}
}
```

OUTPUT:

```
1 1
1 2
1 3
2 1
```

Example Program 2 on labeled for loop:

```
public class LabeledForExample2
{
public static void main(String[] args) {
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break bb;
               }
               System.out.println(i+" "+j);
            }
      }
}
}
```

OUTPUT:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

Example Program 3 on infinite for loop:

- If you use two semicolons ";;" in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;)
{
//code to be executed
}
```

Program:

```
public class ForExample
{
public static void main(String[] args)
 {
   for(;;)
   {
      System.out.println("infinitive loop");
   }
}
}
```

OUTPUT:

```
infinitive loop
infinitive loop
infinitive loop
infinitive loop
infinitive loop
ctrl+c (To exit out of the program)
```

18.2.2 While loop:

- These are used to execute one or more logical instructions based on the condition if the range is not known, these can be achieved by **while** and **do-while** loop.
- While loop is an entry controlled loop that means condition is verified before performing operation only.

Syntax:

```
while(condition)
{
//code to be executed
}
```

Example Program 1 on while loop:

```
public class WhileExample
 {
public static void main(String[] args)
 {
   int i=1;
   while(i<=5){
      System.out.println(i);
   i++;
   }
}
}
```

OUTPUT:

```
1
2
3
4
5
```

Example Program 2 on infinite while loop:

- If you pass true in the while loop, it will be infinitive while loop.

Syntax:

```
while(true)
{
//code to be executed
}
```

Program:

```
public class WhileExample2
 {
public static void main(String[] args)
 {
   while(true){
      System.out.println("infinitive while loop");
   }
  }
}
```

OUTPUT:

```
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
infinitive while loop
ctrl+c (To exit out of the program)
```

18.2.3 do-while loop:

- The Java do-while loop is used to iterate a part of the program several times.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.
- The Java do-while loop is executed at least once because condition is checked after loop body.

Syntax:

```
do
{
//code to be executed
}while(condition);
```

Example Program 1 on do-while loop:

```
public class DoWhileExample
{
public static void main(String[ ] args)
 {
   int i=1;
```

```
        do{
           System.out.println(i);
        i++;
        }while(i<=5);
    }
    }
```

OUTPUT:

```
1
2
3
4
5
```

Example Program 2 on infinite do-while loop:

- If you pass true in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
do{
//code to be executed
}while(true);
```

Program:

```
public class DoWhileExample2
 {
public static void main(String[] args)
 {
   do{
      System.out.println("infinitive do while loop");
   }while(true);
 }
}
```

OUTPUT:

```
infinitive do while loop
infinitive do while loop
infinitive do while loop
ctrl+c (to exit out of the program)
```

18.3 Un-Conditional statements

18.3.1 Break Statement:

- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- The Java break is used to break loop or switch statement.
- It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;
break;
```

Example program 1 on break statement:

```
public class BreakExample
 {
public static void main(String[] args)
 {
   for(int i=1;i<=10;i++){
      if(i==5){
         break;
      }
      System.out.println(i);
   }
}
}
```

OUTPUT:

```
1
2
3
4
```

Example program 2 on break statement:

- It breaks inner loop only if you use break statement inside the inner loop.

Program:

```
public class BreakExample2
     {
     public static void main(String[] args)
         {
             for(int i=1;i<=3;i++)
                 {
                  for(int j=1;j<=3;j++)
                   {
                     if(i==2&&j==2)
                     {
                        break;
                     }
                     System.out.println(i+" "+j);
                   }
                }
         }
     }
```

OUTPUT:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

18.3.2 Continue Statement:

- The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.
- The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

```
jump-statement;
continue;
```

Example program on continue statement:

```
public class ContinueExample
{
public static void main(String[] args)
{
   for(int i=1;i<=10;i++)
    {
      if(i==5)
      {
         continue;
      }
      System.out.println(i);
   }
}
}
```

OUTPUT:

```
1
2
3
4
6
7
8
9
10
```

Switch

Without break

```
int n=10;
switch(n)
{
case 10: S.O.P("In case 10");
case 20: S.O.P("In case 20");
case 30: S.O.P("In case 30");
}
```

Output

In case 10
In case 20
In case 30

With break

```
int n=10;
switch(n)
{
case 10: S.O.P("In case 10");
break;
case 20: S.O.P("In case 20");
case 30: S.O.P("In case 30");
}
```

Output

In case 10

Typecasting

1) Implicit typecasting

```
short s=30;
float f=s;
double d=f;
S.O.P(s);
S.O.P(f);
S.O.P(d);
```

Output

80
30.0
30.0

2) Explicit typecasting

```
double d=84.345345453
float f=(float)d;
int i=(int)f;
S.O.P(d);
S.O.P(f);
S.O.P(i);
```

19. JAVA Expressions:

- Expressions are made up of variables, operators, literals and method calls that evaluates to a single value according to the syntax of the language.
- Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.
- **Example 1:**

```
int abc;
abc = 100;
```

Here, abc = 100, is an expression that returns an int because the assignment operator returns a value of the same data type as its LHS operand.

- **Example 2:**

```
int a=10, b=30;
int c;
c = a + b;
```

Here, c = a + b, is an expression that returns an int.

- **Example 3:**

```
If (number1 = = number2)
{
Sytem.out.println("Hello java world");
}
```

Here, number1 = = number2 is an expression that returns Boolean value and "Hello java world" is an string expression.

19.1 Compound expressions:

- The Java programming language allows you to construct **compound expressions** from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Example of a compound expression is: 1 * 2 * 3;
- Here the above example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications.
- However, this is not true of all expressions. For example, the following example 1 expression gives different results, depending on whether you perform the addition or the division operation first:
- **Example 1:**

```
x + y / 100;    // ambiguous
```

- To overcome this, an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression **unambiguous**, you could write the following example 2:
- **Example 2:**

```
(x + y) / 100;  // unambiguous, recommended
```

19.2 Java Blocks:

- A block is a group of statements (zero or more) that is enclosed in curly braces { }.
- **Example:**

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

```
                String band = "Beatles";
                if (band == "Beatles")
                { // start of block
                System.out.print("Hey ");
                System.out.print("Jude!");
                } // end of block
            }
        }
```

- There are two statements System.out.print("Hey "); and System.out.print("Jude!"); inside the mentioned block above.

20. Java operator precedence and associativity rules

- In Java, when an expression is evaluated, there may be more than one operators involved in an expression.
- When more than one operator has to be evaluated in an expression, **Java interpreter** has to decide which operator should be evaluated first.
- Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators.
- **For example:** multiplication and division have a higher precedence than addition and subtraction.
- Java makes this decision on the basis of the **precedence** and the **associativity** of the operators.

20.1 Precedence order:

- Precedence is the **priority order** of an operator, if there are two or more operators in an expression then the operator of highest priority will be executed first.
- For example: In expression 1 + 2 * 5, multiplication (*) operator will be processed first and then addition. It's because multiplication has higher priority or precedence than addition.

20.2 Associativity:

- When an expression has two operators with the same precedence, the expression is evaluated according to its associativity.
- Associativity tells the direction of execution of operators that can be either left to right or right to left.
- For example, in expression a = b = c = 8 the assignment operator is executed from right to left that means c will be assigned by 8, then b will be assigned by c, and finally a will be assigned by b. You can parenthesize this expression as (a = (b = (c = 8))).
- Note that, you can change the priority of a Java operator by enclosing the lower order priority operator in parentheses but not the associativity.
- For example, in expression (1 + 2) * 3 addition will be done first because parentheses has higher priority than multiplication operator.

20.3 Precedence and associativity of Java operators:

The table below shows all Java operators from highest to lowest precedence, along with their associativity.

Precedence	Operator	Description	Associativity
1	[] ( ) .	array index method call member access	Left -> Right
2	++ -- + - ~ !	pre or postfix increment pre or postfix decrement unary plus, minus bitwise NOT logical NOT	Right -> Left
3	new	Object creation	Right -> Left
4	* / %	multiplication division modulus (remainder)	Left -> Right
5	+ , - +	Addition, Subtraction Concatenation	Left -> Right
6	< <= > >=	less than less than or equal to greater than greater than or equal to	Left -> Right
7	= = !=	Equal to Not equal to	Left -> Right
8	& ^ | && ||	bitwise AND bitwise XOR bitwise OR Logical AND Logical OR	Left -> Right
9	?:	Ternary (conditional)	Right -> Left
10	= += -= *= /= %=	Assignment operators	Right -> Left