Objected Oriented Programming with JAVA

Important Points

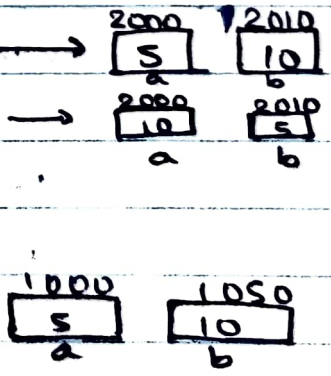
→ Compilation always begin from 1st line of code
→ Execution will begin from main().
→ Example:

```
# include          → Compilation start
# define
int add {
    return sum;
}
void main {          → Execution starts
    add ( )
}
```

Swaping two numbers

M-1 : Call by Value

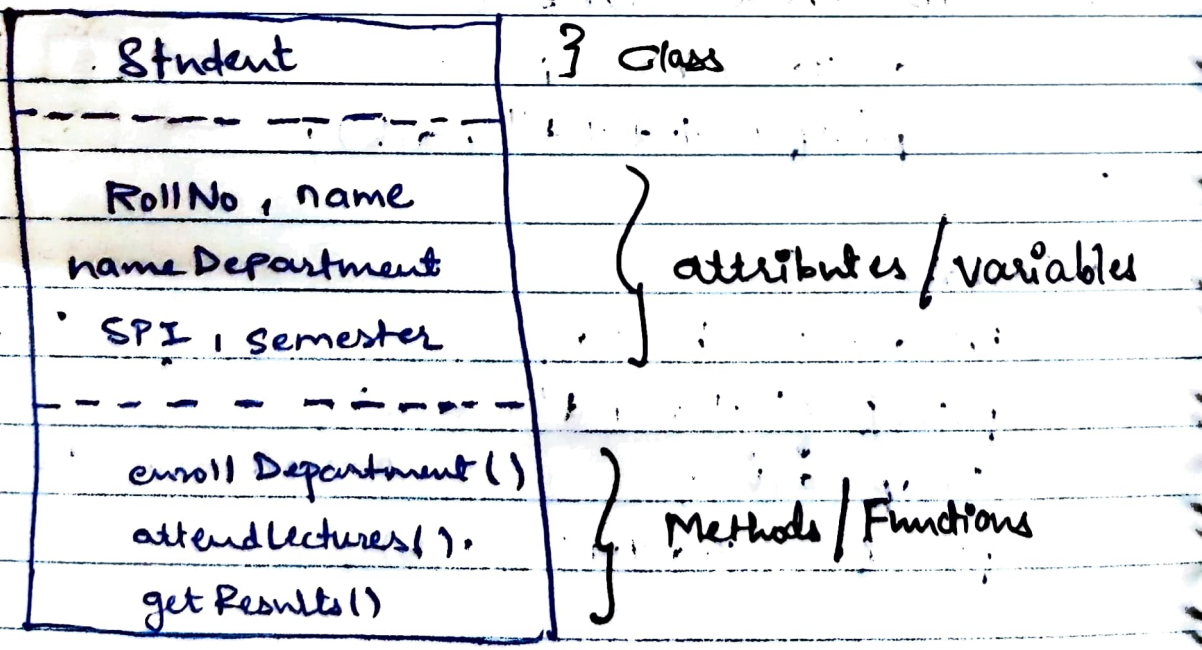
```
void swap (int a , int b){          ← Formal (local) parameters
    a=10 , b=5 ;
    printf ("%d %d", a,b);
}
void main ( ) {
    int a = 5 , b = 10 ;
    printf ("%d %d", a,b);
    swap (a,b);                     ← Actual parameters
    printf ("%d %d", a,b) ;
```



M-2: Call by Address (Pointers)

```
void swap (int *a, int* b) {
    *a = 10;
    *b = 5;
    printf("%d %d", *a, *b);        (10,5)
}
void main() {
    int* a = 5;
    int* b = 10;
    printf("%d %d", *a, *b);        (5,10)
    swap(&a, &b);
    printf("%d %d", *a, *b);        (10,5)
}
```

Objected oriented approach



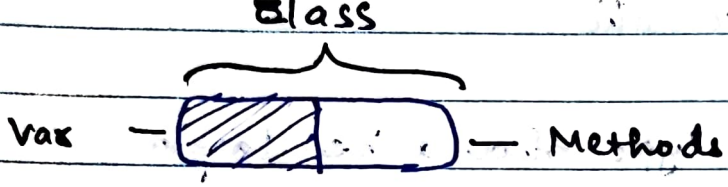
Default Decimal is double datatype

Important Points

→ If same precedence of operators is there (+, -), the entity is calculated from left → right.

Eg x = a + b * c - d ^ e .
③ ② ④ ①

* Incapsulation



* Polymorphism
Same name, many forms

Java Compiler
- Creates Intermediate representation (Bytecode (.class file)).
- Platform independance
- Needs Interpreter (JVM)

• Java is platform independant ; Java Virtual Machine (JVM) is platform dependent

Data Types
→ byte, short, int, long
→ char
→ float, double
→ boolean

} Primitive

Byte	$-2^7 \rightarrow 2^7 - 1$
Short	$-2^{15} \rightarrow 2^{15} - 1$
Int	$-2^{31} \rightarrow 2^{31} - 1$
long	$-2^{63} \rightarrow 2^{63} - 1$

Type Conversion

① Implicit – Automatic
- → Compatibility
- → Destination type should be larger

② Explicit – Casting
- → Incompatible

Type Promotion

Java automatically promotes byte, short, char operand to int.

Main Method

```
public static void main (String [] args)
```

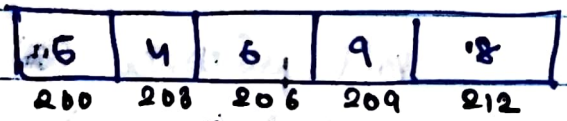
- public → access modifier
- static → In JVM, no object of class is present
- void → Doesn't return anything
- String [] args → Command line arguments

Print Method

1. public void print (String s)
2. public void println (String s)
3. public PrintStream printf (Str format, Object)

Array

- Like typed variable
- Can have one or more dimensions
- Continous memory allocation



Declaration → Datatype [ ] name;
→ Datatype name [ ];
→ Datatype [] name;

Instantiation: int a [ ] = new int [5];

Declaration → int a [10];
Initialisation → int a[10] = { 1, 2, 3, 4 };
Assignation → int a [10];
a [0] = 1;
a [1] = 2;

Note :- Initialized array with zero.

After declaration:

```
Int [] numbers;
numbers = new Int [] { 22, 33, 44, 55, 66 }
```

String:

- Not primitive datatype
- It's an object
- Not an array of character

Practise Example

```
class student {
    String name;
    float marks;
    int no;
    void setData ( float m, int n ) {
        marks = m;
        no = n;
    }
}
void printData (s) {
    Sout (s.marks);
    Sout (s1.name);
    sout (s1.no)
}
class mainProg {
    public static void main ( String [] args ) {
        Student s1;
        s1.name = "abc";
        s1. marks = 100;
        s1. no. =
    }
```

Default arguements must be written on right side.

```
        Student S2;
        S2.name = "xyz";
        S2.marks = 96;  ,  S2.no = 2;
      S1.printData();
      S2.printData();
    }
}
```

Example - ~~Constructor~~

```
class Point {
    int x, y, z;
    void setData (int ix, int iy, int iz) {
        x = ix;
        y = iy;
        z = iz;
    }
}

class mainClass {
    public static void main (String [] args) {
        Point P1 = new Point(5,10,15);
        // P1.setData(5,10,15);
        Point P2 = new Point();
        P2.x = 50;
        P2.y = 60;
        P2.z = 70;
```

class

object

Constructor

Constructor - Examples

```
class Point {
    int x, y, z;
    Point () {          // Default Constructor
        x = 0;
        y = 0;
        z = 0;
    }

    Point (int ix, int iy, int iz) {
        x = ix;
        y = iy;
        z = iz;
        System.out.println("2nd const. invoked");
    }
    ~Point();
}
class mainClass {
    psvm (String [] args) {
        Point P1 = new Point (5, 10, 15);
        Point P2 = new Point ();
    }
}
```

Invoke second constructor

Invoke default constructor

Destructor would get invoked

Destructor

→ Syntax: ~classname();
→ Can't be overloaded
→ No return type

STATIC

INSTANCE METHOD / OBJECT METHOD

obj.attribute;
obj.method;

STATIC METHOD

<classname>.details();
Student.noofstudent();

EXAMPLE:

```
class Counter{
  static int count=0;
        Counter() {
            count++
            Sout(count);
        }
}

psvm (String[] args){
        Counter c1 = new Counter();
        Counter c2 = new Counter();
}
```

OUTPUT:

1
1 } Every object has its own copy of count

As, the variable (count) is initialized in instance method.

1
2

If static variable is used

Points to Ponder :-

- Static member can't call instance member
- Static method can call instance method
- instance method can call static method
- instance member can call static member

Variable Arguements {Varargs}

Syntax : public static void fun (int ... a)

- Internally treated as an array
- Same Datatype must be there
- Eg fun (100);
  fun (1, 2, 3, 4);
  fun ( );
- Varargs must be at last fun (3, 4, double ... a)
- Only one Vararg in a single method.

Copy Constructor

- Create an exact copy of existing object
- Changes made in copy are not reflected in original.
- Returns duplicate copy of existing object.
- Creating new Constructor at new memory location.

Example : class name : copyTest

CREATING COPY CONSTRUCTOR

```
copyTest ( copyTest m ) ⟶ Copy Constructor
    Sout (" In copy Constructor");
    a = m.a
    name = m.name
```

APPLYING COPY CONSTRUCTOR

```
copyTest t3 = new copyTest (10, 'IT');
copyTest tn = new copyTest (t3); } Creating copy constructor
```

↳ tn.name = "ICT"
↳ This will not change t3.name.

Final Keyword

- used to restrict user
- Value can't be changed
- It can be for Variable, Method, class.

→ Syntax for Variable (final)

```
final int speed = 70;
```

Questions

1. Difference between VarArgs and method Overloading
2. Difference between Copy constructor and ob.clone() method ?
3. Can we overload main() ?
4. Difference between Constructor & main ?

INHERITENCE

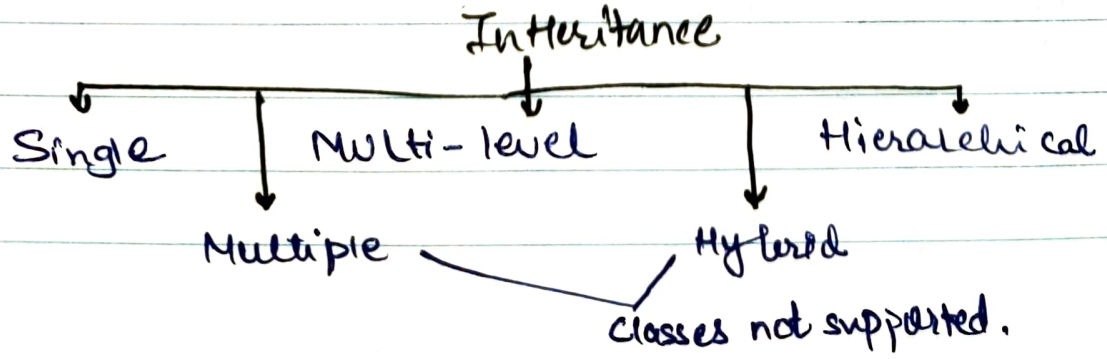
→ One object acquires all the properties and behaviours of a parent object.

→ Reusability
→ Method overriding (Run time polymorphism)

Syntax:

```
class subClass extends superClass { ... }
```

→ Variables declared in superclass & not in subclass can be used in sub class.

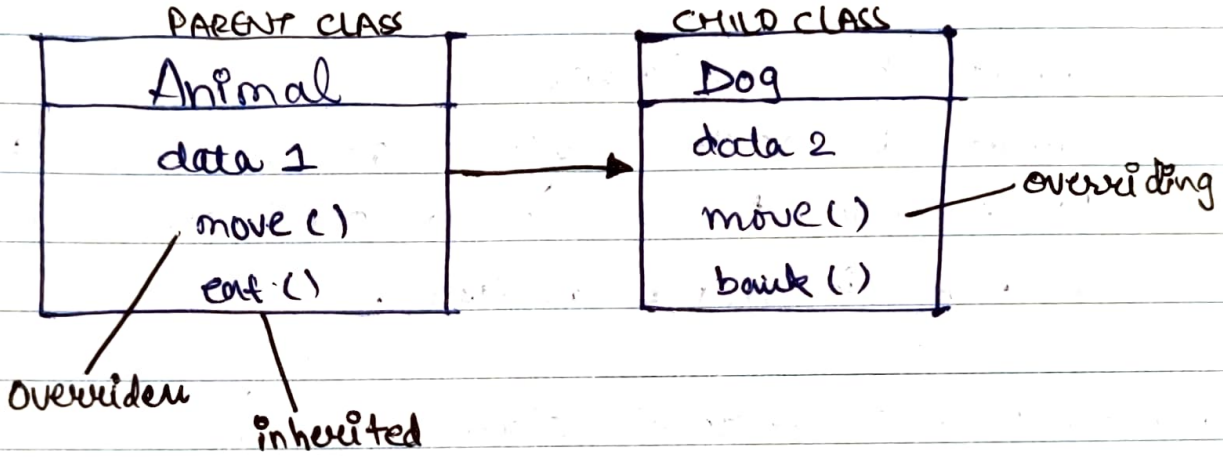


Use of Super class :

It is used to initialise attributes of superclass in child class. super (Att 1 , Att 2 , - - - -)

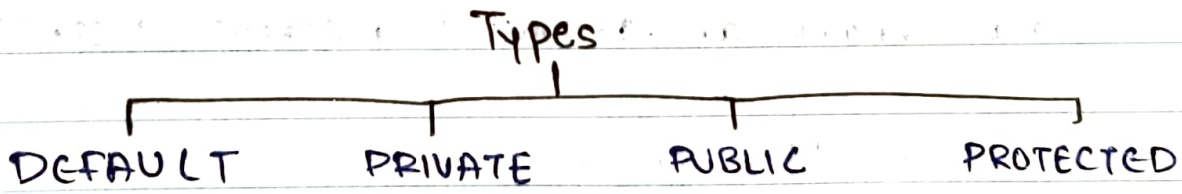
METHOD OVERRIDING

→ The method must have the same name as in parent class and must have the same parameter



ACCESS MODIFIERS

- To restrict the scope of a class, constructor, variable, data member, method.



IMPORTANT: Sequence of Constructor Calling

Brain Storming Session - 02

1. ~~Nature~~ Famous Painting By MF Hussain Nature
2. 2
3. Compile time Polymorphism error

Abstraction

- Helps to hide implementation details.
- Cannot be subclassed
- Cannot be create object of abstract class
- Necessary to create abstract class when method is made abstract.

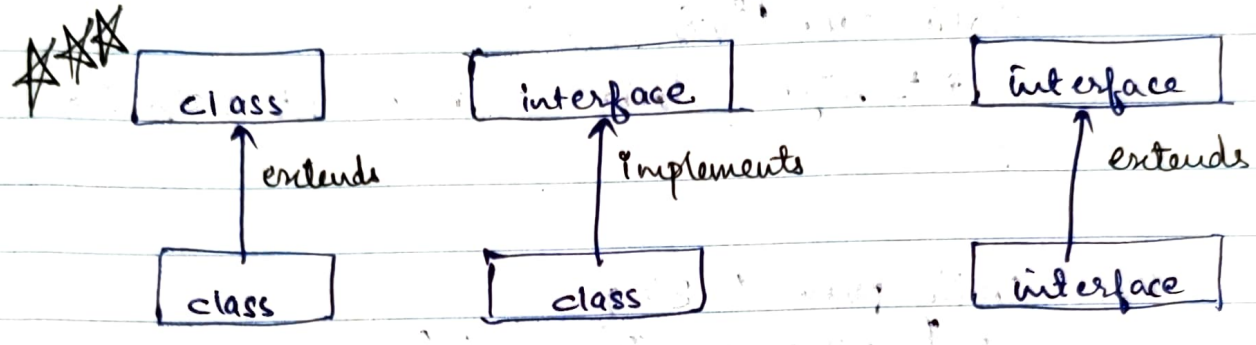
Syntax: → abstract class { }
→ abstract method ( ) ;

- Achieved through method overriding
- Implements RUN TIME POLYMORPHISM also known as DYNAMIC METHOD DISPATCH

INTERFACE IN JAVA

→ A Blueprint of a class
→ A class that includes an interface must implement all method with public access.

→ Cannot have a method body
→ It has IS-A relationship
→ Cannot be Instanciated (object can't created) .



A class can ~~Inherit~~ Implement more than one interfaces but can Inherit only a superclass.

SYNTAX : Interface <classname> { }
subclass implements <class name> { }

→ Method by default becomes " public abstract "
→ Method Overriding also occurs in Interface.

Q Implement Interfacing in stack

```
Interface stackInterface {
    void push (int ele);
    ~~void push ();~~
    int pop();
}
```

```
class staticStack implements stackInterface {
    int stackarr[];
    int top;
    staticStack(int size)
        stackarr = new int[size];
    }

public void push(3){
    if(TOP == size - 1){
        sout("overflow"); }
    else{
        stackarr[++top] = ELE;
    }
    return stack[top];
}

public void pop(  ){
    if( top==-1)
        Top==-1
    {
```

Java Principles

1. Encapsulation
2. Inheritance
3. Polymorphism

1) Encapsulation

→ Process of wrapping code & data together into a single unit.

→ Advantages of encapsulation

1) Read only / write only - class
2) Provides control over data structure
3) Data hiding
4) Easy & fast way to create class.

2) Polymorphism

→ Perform single actions in different ways.
→ Poly + morphism = many + forms
→ Two types — Compile time, Run time.

Abstraction

→ Hiding implementation details, showing only functionality to user.

3) Inheritance.

→ One object acquires all properties & behaviours of a parent object.

a) Object - Instance of a particular class
b) Class - Blueprint of properties & behaviour
- Group of objects with common properties

OOP	POP
→ Objected oriented	→ Structure oriented
→ Divided into objects	→ Divided into functions
→ Bottom up approach	→ Top-Down approach
→ Inheritance used	→ Inheritance not allowed
→ Code reusability	→ No Code reusability
→ Data hiding through Encaps.	→ Less Secure / No data hiding
→ Access modifiers used	→ No access modifiers
→ Overloading can be done	→ No Overloading
→ Eg: C++, Java, Python	→ Eg: C, Fortran

Package

→ Group of similar types of classes, interfaces & sub packages.

Constructors

→ Block of codes similar to method.
→ Initialize state of an object
→ No Return Type
→ Invoked implicitly
→ Same name as class name.
→ Two Types - Default, Parameterized.
↓ ↓
No Parameter Specific no. of parameters.

Constructor	Method
→ To initialise state of object	→ To expose behaviour of object
→ No return type	→ Must have a return type
→ Invoked implicitly	→ Invoked explicitely
→ Same name as class	→ Not necessary for same name
→ Default Constructor provided by Compiler	→ No default method provided by compiler.

Method Overloading

→ Method with same name & different parameters.
→ Increases Readability of program.

Method Overriding

→ Sub class having same method in parent class.
→ Run time Polymorphism.

Method Overloading	Method Overriding
→ Compile time Polymorphism	→ Run time Polymorphism
→ Occurs within class	→ performed in 2 classes
→ May/May not require Inheritance	→ Always need Inheritance
→ Same name, different parameters	→ Same name, same parameters.
→ Return type may or may not be same but parameter should be chd	→ Return type must be same/covariant.
→ Private and final method can be overloaded	→ Private and final metho can't be overriden.
→ Static Binding	→ Dynamic Binding

Featurs of Java

1. Simple
2. Object oriented
3. Platform Independent
4. Secured
5. Robust
6. Portable
7. High Performance
8. Multi-threaded
9. Distributed
10. Dynamic.