Object Oriented Programming

Procedure oriented vs. object oriented programming

Procedure oriented : the problem is viewed as a sequence of things.

→ The primary focus is on functions.

→ The data is completely passive.

Object oriented : OOPs treats data as a critical element.

→ There is an emphasis on data rather than procedure.

→ Programs are divided into what is known as objects.

→ Functions that operate on the data of the object are tied together in the data structure

* Class : A class is a user-defined blueprint that has a set of attributes and behaviours that define the item that it represents

→ Attributes are properties of the class.

For eg. in a class Triangle,

the attributes may be base, height,

→ Behaviours | Methods : functions which are related to and are located inside the class.

For eg. the Triangle class may have a findArea() function

Objects : → objects are the basic run-time entities in an object-oriented system.

→ Objects may represent a person, place, thing, that the program must handle.

→ Every object is an instance of a class that it belongs to.

→ The object may have fields, variables or data. The class to which the object belongs describes those fields and methods.

* Constructors : → a special method inside every class that creates and initializes instances.

Defining a constructor

```
class Triangle {
    int base;
        int height;
    Triangle (int base, int height) {
            this.base = base;
            this.height = height;
    }
```

this : The 'this' keywords helps differentiate between the attribute variables and the parameter variable.

* Instantiation → Used to create an object of the class

→ the 'new' keyword is used.

syntax

<ClassName> <objname> = new <class name()>

Instance vs. Class methods

or

Non-static vs. Static methods

Instance methods: a method depends on the attribute value of a given instance of a class.

→ can only be used if there is an object

for eq. <stringvar>.charAt(pos)

The charAt method works on a specific string of the String class.

→ called non-static methods

Class methods: does not require an instance of the class for it to be used.

for eq. Math.pow(n1, n2)

an instance of the math class does not have to be made, the class name is used rather than the instance name.

* Instance vs. Class Variables

Class variables: value is the same for all instances of a class

for eq. in the Δ class

static int numsides = 3; is a class variable.

Instance Variables: hold information about a particular instance

inside the class, they are accessed as this.<instancevar>

outside, they are referred to as

<class name>.<varname>

Encapsulation

→ Wrapping up of data and functions into a single unit.

→ Encapsulation allows one to make modifications to a single class w/o affecting the rest of the program.

→ It allows a class' attributes to be hidden from other classes. This is called as data hiding or information hiding.

* Data Abstraction

→ to represent only essential features without including background details.

→ Abstraction can be achieved with abstract classes or interfaces

→ An abstract cannot be instantiated, it can only be inherited. The implementation of an abstract class lies in the subclasses it inherits.

→ a method can be declared as abstract, only if the class as a whole is abstract.

Consider the following example:

```
abstract class Animal {
        public abstract void animalSound();
        public void sleep() {
                System.out.println("Zzzzz");
    }
```

In the inherited classes, Pig & Cow, the animalSound() fn must be implemented.

```
class pig extends Animal {
    public void animalSound() {
        System.out.println("Oink");
    }
}

class cow extends Animal {
    void animalSound() {
        System.out.println("moo");
    }
}
```

* Inheritance : process of one class acquiring the properties of another class.

Inheritance helps avoid duplication of code.

Changes made in the parent class are seen in the subclass as well.

- The relationship between a subclass and a superclass is an Is-A relationship.

* Polymorphism : ability to take on more than 1 form. An operation may exhibit different behaviours in different instances.

For eg. addition

w/ numbers : a sum would be generated

w/ strings : concatenation

Polymorphism is of 2 types : runtime polymorphism

compile time polymorphism

Polymorphism is implemented through function overloading, operator overloading, function overriding

Operator overloading: + can be for summation / concatenation

Function overloading: a single function to perform different types of tasks

Function overriding: Same function points to different tasks during runtime = function overriding

* Message Communication

→ a set of objects that communicate with each other.

→ message communication involves specifying the name of the object, the name of the function (message) and the info to be sent

emp. salary(emp-name)
↓ ↓ ↓
object message information

Features of Java

- Object oriented
- Interpreted (Java program → compiler → byte code → JVM interpreter → output)
- Robust (Extensive exception & error handling)
- Architecture neutrals
- high performing, dynamic

Basic Java

Jump Statements : using labelled breaks

→ To create a label, use an identifier followed by a :

→ use break <label name> to jump out of that block.

for eg.

```
class jumpStatements {
    public static void main(String args[]) {
        outer : for(int i=0; i<3; i++) {
            System.out.println(i);
            for(int j=0; j<100; j++) {
                if(j==10) {
                    break outer;
                }
            }
        }
    }
// this breaks out of both loops
```

o/p : 0 1 2 3 4 5 6 7 8 9

Arrays

- Declaration :

```
int [] arr = new int[size]
```

or `int [][] arr = new int[size1][size]`

- Jagged Arrays

```
int [][] arr = new int[2][]
arr[0] = new int[3]   // first row has 3 columns
arr[1] = new int[2]   // second row has 2 columns
```

*** " == " vs .equals( )

- == is an operator
- == compares 2 objects based on memory reference
- It, compares memory references, and if they are the same, it returns true

- .equals() is a method
- compares content of Strings

Usage of " == "

String a = new String ("SSN");
String b = new String ("SSN")
String c = a

(1) a == b => False
(2) c = a => True
(3) a == SSN => false

2 new String objects are created

Usage of .equals()

(1) a.equals(b) => True
(2) a.equals(c) => True
(3) a.equals("SSN") =>

However, if strings are assigned " == " works.

String a = "ssn";
String b = "ssn";
String c = a
a == b
a == c
a == SSN
} true

Java automatically checks during compile time if string constants are assigned to the same value, and at the start assigns them to the same oblect for optimization.

* Garbage Collection in Java

Object destruction happens via in-built garbage collection.

Consider the following example

aCircle, bCircle refer to 2 circle objects.

They are not objects themselves, as they point to nothing

To dynamically make an object:

```
aCircle = new Circle();
bCircle = new Circle();
```

bCircle = aCircle

⇒ The object of bCircle does not have a reference.

⇒ The object becomes a candidate for garbage collection.

⇒ Java automatically collects garbage periodically, and releases the memory to be used in the future.

* Constructors in detail

→ are automatically invoked at the time of object creation

→ constructors have the same name as the class name

→ there can be multiple constructor

→ If no constructor is declared, a default constructor is called

Constructor Overloading & Method Overloading

Method Overloading is a kind of compile-time (= static) polymorphism

In method overloading, there are multiple methods with the same name, but w/

diff no / type of arguments

(or) different order of args

Note that the return type does not participate in method overloading.

Changing the order of variables of the same datatype would also raise an error.

For example

```
class Sample {
    int add (int a, int b) {
        return a+b;
    }
    float add (int a, int b, float c ) {
        return a+b+c;
    }
    String add ( String a, String b) {
        return a+b;
    }
}
class Test Driver {
    public static void main ( String, args[]) {
    Sample obj = new Sample();
    s.o.p ( obj.add (2,3)); // fn ① is called
```

```
s.o.p (obj.add (2,2,2));   // fn (2)
s.op.(obj.add ("cat" ,"bat"));   // fn (3)
s.op (obj.add (2.0 . 4.0 ,2)) ;  // would raise an error
                                    lossy conversion
s.op (obj.add (2, 2, 2.2));  // 2.2 taken as a double
                                must be converted to float
```

⇒ use

```
s.o.p(obj.add(2,2,2.2f));
```

Constructor Overloading

→ Constructors can be overloaded.

→ Multiple conshuctors can be defined, and then called based on the arguments supplied.

For example :

```
class Circle {
    int radius;
    String color = "red";
    Circle ( String color, int radius ) {
        this. color =color;
        this. radius =radius;
    }
    Circle ( int radius) {
        this . radius= radius;
    }
}
```

```
public class Test {
    public static void main ( String args [ ] ) {
        int radius
        Circle c1 = new Circle(5);          ⇒ fn (2)
        Circle c2 = new Circle ("blue", 10);  = fn (1)
    }
}
```

* Constructor Chaining

The process of calling one constructor from another constructor with respect to the current object.

→ Reduces code redundancy by involving an already created constructor.

for eg.

```
class Multiplication {
    Multiplication () {
        s.o.p ("Default constructor");
    }
    Multiplication( int x) {
        this( ),
        s.op (x),
    }
    Multiplication (int x, int 4)
        this (5);
        s.op (x*4)
```

```
class Test {
    public static void main (String.args[]) {
        Multiplication obj = new Multiplication (8, 10);
    }
}
```

// Initially fn (3) is called.

then (2)

then (1)

o/p : Default constructor

5

80

* Encapsulation with Access Specifiers

→ The visibility of variables and methods in a class can be controlled using access specifiers.

→ The 4 access specifiers are public, private, protected, package private.

	inside same class	in a diff. class	inherited class	inherited, diff package	outside class, diff package
public	✓	✓	✓	✓	✓
private	✓	X	X	X	X
protected	✓	✓	✓	✓	X
package private	✓	✓	✓	X	X

→ In order to access / modify private variables, public methods can be written in the class in which they are defined.

→ This allows for indirect modification

→ This is done through getters and setters.

* Destructors

→ Java allows one to define a finalizer method, which is invoked just before object destruction.

→ This presents an opportunity to perform record-maintenance operations or clean up of any special allocations made by the user

→ used as public void finalize()

* Input and Output Processing

(1) using the Scanner class

(2) using the Buffered Reader Class

→ To read text from a character-based input stream.

```
BufferedReader br = new BufferedReader (
                    new InputStreamReader (System.in));
// read data using readLine();
```

(3) using System.console

```
String name = System.console.readLine();
```

(4) Data Input Stream:

```
DataInputStream dis = new DataInputStream (System.in),
```

Inheritance : an object-oriented principle where code is organized through class hierarchies. Classes inherit properties and behaviours from other classes.

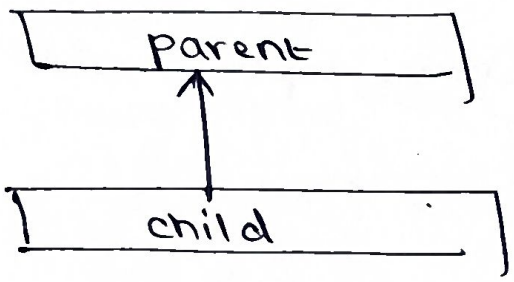
Subclasses inherited from a parent class can:

(i) add new functionality
(ii) use inherited functionality
(iii) override inherited functionality

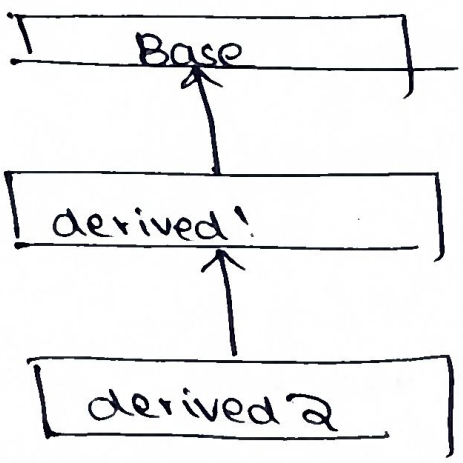
Inheritance is implemented using the extends keyword.

Types of Inheritance

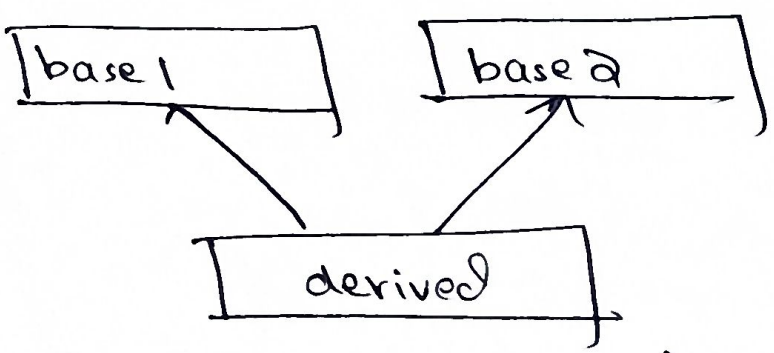
① Single - level



② multilevel

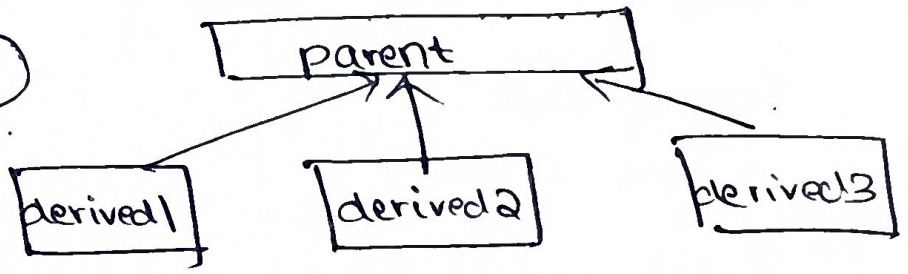


③ multiple

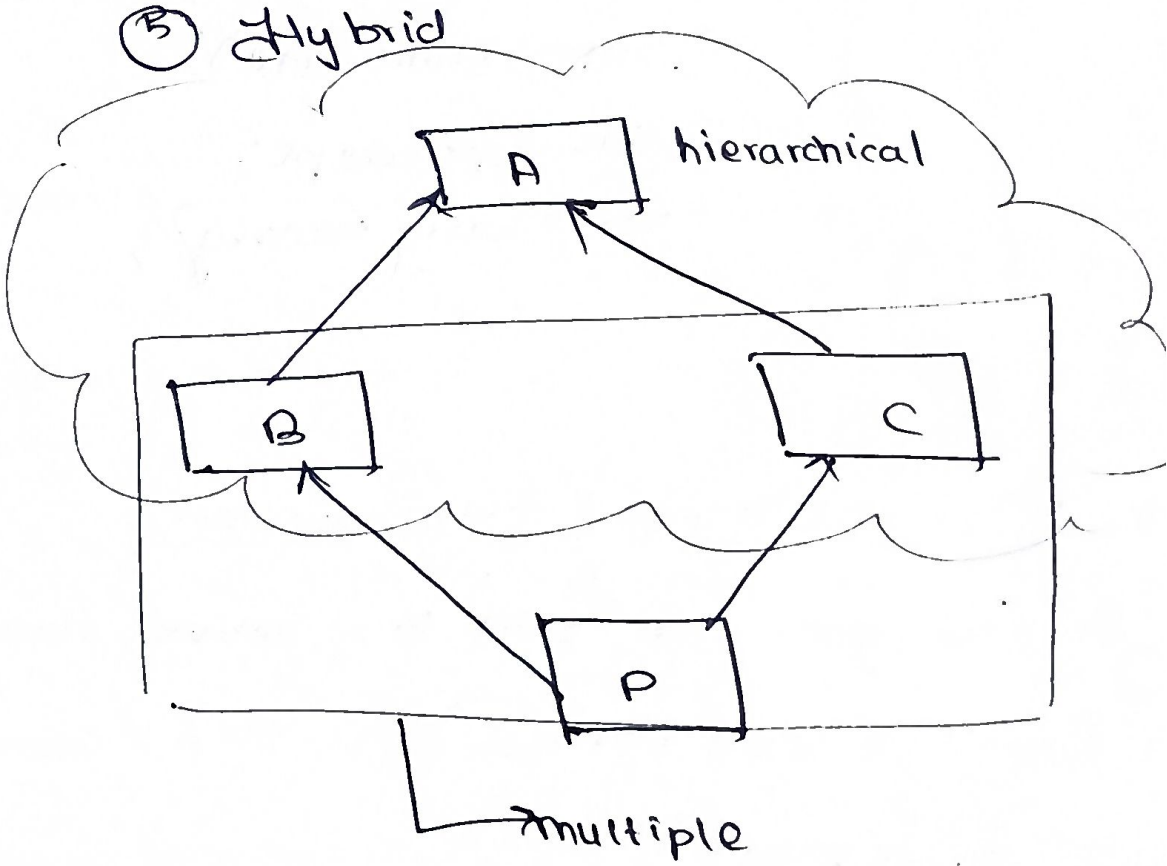


(not supported in Java)

④



⑤ Hybrid



For Ex, consider the class Employee that inherits from person

```
class Person {
        String name;
         int age;
        Person ( String name, int age) {
                this.name = name;
                this. age = age;
         }
}

class Employee extends Person {
        String dept;
        double salary;
        Employee ( String name, int age, String dept, double
                                                salary) {
                super(name, age);
                this. dept=dept;
                this. salary =salary;
         }
}
```

* Base and derived class references

A base class can point to a derived class, but not vice versa.

eg. Base d = new Derived()

d. display();

Method Overriding

→ Subclasses inherit all methods from their superclass.

→ Sometimes the implementation in the superclass is not sufficient.

→ In these cases, the method must be overridden,

→ Overriding is done by providing the same fn. with a diff implementation in the subclass, with an identical signature.

→ a kind of runtime polymorphism, also called Dynamic Method Dispatch

For eg:

```
class Phone {
    void showTime() {
        s.o.p ("Time is 8 am");
    }
    void on() {
        s.op ("Phone is on");
    }
}
class Smartphone extends Phone {
    void on() {
        s.op. ("Smartphone is on");
    }
}

public class Test {
    public static void main (String args[]) {
        Phone p = new Smartphone();
        p.showTime();        // Time is 8 am
        p.on();              // Smartphone is on
    }
}
```

*** Overloading vs Overriding

Overloading: same fn, have different signatures

constructors are often overloaded

static / compile-time polymorphism

static and dynamic binding

↓ ↓

compile runtime polymorphism

Final Keyword

on a

(i) variable => a / constant variable

(ii) methods => to prevent overriding

(iii) classes => no child class

* Restricting Inheritance

Abstract methods => no implementation inside

→ a class w/ abstract methods, must be abstract class itself

→ inherited classes must necessarily override the base class abstract definitions

→ abstract classes cannot be instantiated

Interfaces: used to achieve multiple inheritance

In interfaces, all variables are final and methods are abstract and public.

→ They cannot have any implementation at all

Syntax

class <ClassName> implements interface1, interface2 {

}

* Interfaces extend other interfaces

interface interface1 extends interface a {

}

* a combination of inheritance & interface

class ClassName extends <parents> implements <interface1>, <interface 2> {

}

Class Diagram Notation

- private
+ public
# protected
~ package private