Object Oriented Programming

let's look at an example :-

Example 1

```
class Pen {
    String color;
    String type;

    Public void write() {
        System.out.println("write Something");
    }
    Public void PrintColor() {
        System.out.println(this.color);
    }
}

Public class OOPs {
    Public Static void main(String args[])
    {
        Pen pen1 = new Pen();
        pen1.color = "blue";
        pen1.type = "gel";

        Pen pen2 = new Pen();
        pen2.color = "black";
        pen2.type = "ballpoint";

        Pen1.printColor();
        Pen2.printColor();

        Pen1.write();
    }
}
```

output :- blue
black
write Something

2022

JULY

		1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31							

M T W T F S S M T W T F S S

June

24th Wk • 161-204

10 Friday

~~Example :-~~ Explanation :-

✓ ~~objects~~ class are the blueprint of an ~~class~~ object. Every ~~object~~ class has some properties & methods.

✓ The properties define the characteristics of an ~~object~~ class. In this example, class pen has the properties color, type.

✓ Similarly all class has some members or function. In this example, class pen has 2 members - write() & printColor().

✓ Whenever an object of the class calls the function, the inner code gets executed. Object of one class can call the functions of that class from another class. If there are multiple objects of a class, depending on which object has called the function, the function gets executed.

✓ "this" is a keyword in Java. It tells the function, which object has called the respective function. In this example, when "pen1.printColor()" statement gets executed "this" keyword informs the function "printColor()" that it has been called by pen1 and then the color property of pen1 is executed. Similar logic applies to pen2 also.

✓ "Pen pen1 = new Pen();" → this statement is very important. "new" is a keyword in Java to create the object and 'pen()' acts as a constructor. When "new" keyword is encountered, Java automatically creates a space in the memory for the object.

11 June
Saturday
24th Wk • 162-203

2022

JUNE

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

✓ Constructors has special features compared to normally defined methods —

(a) constructors can be called only once, unlike other methods that can called several times.

(b) Even if we donot define a constructor, Java by default creates a constructor.

(c) constructor doesn't have a return type and it never returns any value. It is only used to set up the properties or printing some value of the object.

(d) constructors can be 3 types —
(i) Non-parameterized constructor
(ii) parameterized constructor
(iii) copy constructor.

* Example of non-parameterized Constructor :-

```
class Student
{
    String name;
    int age;
    Student() {
        System.out.println("I am a student");
    }
}
class oops
{
    Student S1 = new Student();
}
```

→ here, when the object "S1" is created, automatically the constructor "Student()" is called and the print statement is executed.

output :- "I am a student."

2022 JULY

June 12 Sunday

24th Wk • 163-202

* Example of a Parameterized constructor

```
class Student
{
    String name;
    int age;

    Student( String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void PrintInfo () {
        System.out.println (this.name);
        System.out.println (this.age);
    }
}
class OOPS
{
    public static void main (String args[])
    {
        Student s1 = new Student ("aman", 24);
        s1.printInfo ();
    }
}
```

this 'name' is the variable of the class

this 'name' is the parameter of the constructor

this value is passed to the constructor parameter "name"

this to the "age"

output :- aman
24

13 June 2022
Monday
25th Wk • 164-201

JUNE
M T W T F S S M T W T F S S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

* Example of a copy constructor

A copy constructor simply copies the data / properties of an object and gives it to another object.

```
class Student {
    String name;
    int age;

    Public void printInfo ( )
    {
        System.out.println (this.name);
        System.out.println (this.age);
    }

    Student (Student s2) {
        this.name = s2.name;
        this.age = s2.age;
    }
    Student( ) {
    }
}

Public class oops {
    Public static void main (String args[])
    {
        Student s1 = new Student ();
        s1.name = "aman";
        s1.age = 24;

        Student s2 = new Student (s1);
        s2.printInfo ();
    }
}
```

here 's2' gets all values of data from 's1' and assign it to 'name' & 'age' of the class.

need to define the empty constructor for s1, bcz it is not created by default in this case

's1' is passed as an argument

output :- aman
24

So, basically s2 gets all values of it's properties from s1 and prints those values.

There are 4 Pillars of OOPs —

(a) Polymorphism
(b) Inheritance
(c) Encapsulation
(d) Abstraction.

** Polymorphism **

Polymorphism basically means using the same function but in different forms or manners. There are 2 types of Polymorphism —

- Compile-time Polymorphism or function overloading
- Run-time polymorphism or function overriding

Let's see compile-time Polymorphism

```
class Student {
String name;
int age;

Public void printInfo (String name)
{
S.o.pln(name);
}

Public void printInfo (int age)
{
S.o.pln(age);
}

Public void printInfo (String name, int age) {
S.o.pln(name);
S.o.pln(age);
}
}
```

15 June 2022
Wednesday
25th Wk • 166-199

JUNE
M T W T F S S M T W T F S S
1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30

```
class oop{  Public static void main (String args[]){
    Student S1 = new Student();
    S1.name = "aman";
    S1.age = 26;

    S1.printInfo(S1.name);
    S1.printInfo(S1.age);
    S1.printInfo(S1.name, S1.age);
}
```

✓ So, here we have 3 different function with the same but performing different activities.

✓ In compile-time Polymorphism, if return type of 2 functions are same, then return type of parameters has to be different.

✓ In compile-time polymorphism, signature has to different for different functions.

✓ Functions are overloaded here and with the help of the parameters, the right function is called and executed.

✓ We shall study Run-time Polymorphism or function overriding in Inheritance.

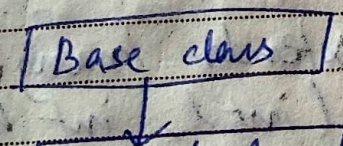
** Inheritance **

Inheritance basically means inheriting the property of one class by another class.

The class from which the property is inherited is called the parent class or base class. The class which derives the properties is called the child class or Sub class.

In Java, there are 4 types of inheritance —

✓ Single-level inheritance

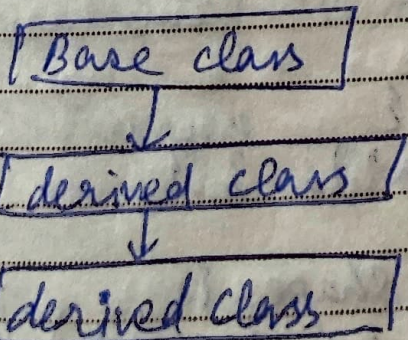


Eg.

```
class Shape {
    public void area() {
        S.o.pln ("displays area");
    }
}
class Triangle extends Shape {
    public void area (int l, int h) {
        S.o.pln (1/2 * l * h);
    }
}
```

child class deriving the properties of parent class.

✓ Multi-level inheritance



17 June 2022
25th Wk • 168-197
Friday

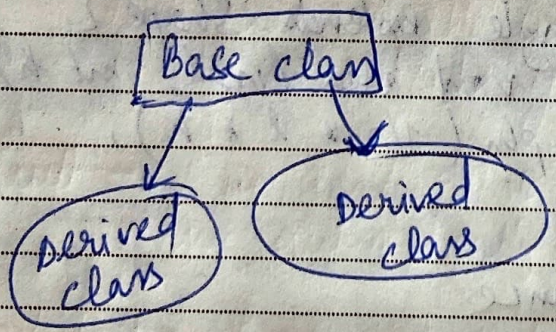
E.g,

```
class Shape {
    Public void area() {
        System.out.println("displays area");
    }
}
class Triangle extends Shape {
    Public void area(int l, int h) {
        S.o.pln(1/2 * l * h);
    }}
class EquilateralTriangle extends Triangle {
    Public void area(int l, int h)
    {
        S.o.pln(1/2 * l * h);
    }
}
```

extends the base class (Shape)

extends 'Triangle' which becomes it's base class.

✓ Hierarchical Inheritance



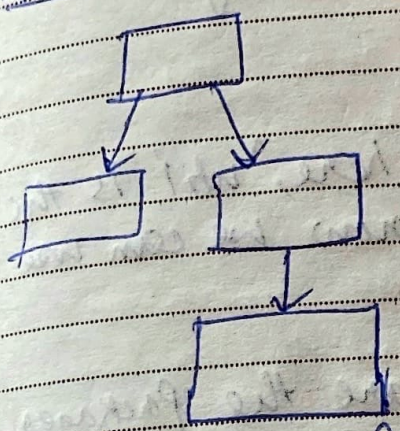
E.g,

```
class Shape {
    Public void area() {
        S.o.pln("area"); } }

class Triangle extends Shape {
    Public void area(int l, int h) {
        S.o.pln(1/2 * l * h); } }

class circle extends Shape {
    Public void area(int r) {
        S.o.pln(3.14 * r * r); } }
```

Hybrid Inheritance



So, in this type of inheritance some classes are derived from an already derived class (multi-level inheritance) and some are derived from the main base class. Both can be defined parallely.

✱ note :-

- In Inheritance, we use Run time polymorphism or function overriding. Unlike compile-time polymorphism, here we see same function name and signature is used multiple time but in different classes.
  Compiler cannot differentiate between such functions and only in Run-time the intended function is called and executed.

Before moving to the other 2 concepts of OOPs, let's try to understand 2 major things -

- Packages
- Access modifiers

A Package is like a storehouse that can contain several classes. one class belonging to a package can access another class defined in another package, if made public.

In Java, there are 2 types of Packages —
- Built-in Package
- user-defined Package.

E.g. import java.util.* → here, 'util' is the package that we are importing and now we can use any of its classes or functions.

user-defined Packages are the Packages that we can write and insert classes as per our wish.

✓ Access Modifiers

They are the type of class/methods that define it's accessibility. There are 4 types of access modifiers —

• Public • Protected • Private • default.

• Public → If we define any variable of a class as public, then that variable will accessible by any class inside or outside of that package.

• default → If we don't define any access modifier, Java automatically makes it as default.

• Protected → If we use protected for any variable, then only the child classes of the parent class outside the package can use it. classes within the same package can use it also.

• Private → If we use private for any variable, then no ~~only~~ classes within the same package ~~can use it~~ or outside the package can use it. If we need to access, we can use getters & setters.

E.g.

```
Package bank;

class Account {
    Public String name;
    Protected String email;
    Private String Password;

// getters & setters - to access Private Properties
    Public String getPassword() {
        return this.password;
    }
Public void setPassword (String Pass) {
    this.password = pass;
}
}

Public class Bank {
Public Static void main (String args[])
{
    Account account1 = new Account();
    account1.name = "Apna college";
    account1.email = "apna978@gmail.com";
    account1.setPassword ("abcd");
    System.out.println (account1.getpassword());
}
}
```

output :- abcd

** Encapsulation **

Creating classes and objects basically means combining the data and ~~properties~~ methods into a single unit (class) and that process is called encapsulation. Data means the properties and methods meaning the functions.

By the help of access modifiers, we can perform data hiding also, while performing Encapsulation.

21 June 2022
Tuesday
26th Wk • 172-193

JUNE
M T W T F S S M T W T F S S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

** Abstraction **

Data hiding is the process of protecting members of class from unintended changes.

whereas, Abstraction is hiding the implementation details and showing only the important/useful parts to the users. We can implement abstraction in two ways - (i) using abstract class (ii) using interfaces

E.g ⇒ (1) using abstract class

```
abstract class Animal {
    abstract void walk();
}

class Horse extends Animal {
    public void walk() {
        System.out.println("walks on 4 legs");
    }
}

class Chicken extends Animal {
    public void walk() {
        System.out.println("walks on 2 legs");
    }
}

public class OOPs
{
    public static void main(String args[])
    {
        Horse horse = new Horse();
        horse.walk();
        Animal animal = new Animal();
        animal.walk();
    }
```

abstract class or method is like a concept. Every child class can have it but it has no implementation of it's own.

will throw error because Java can't instantiate abstract class. Runtime error.

Properties of abstract class

- ✓ An abstract class must be declared with an abstract keyword.
- ✓ It can have abstract and non-abstract methods.
- ✓ It can't be instantiated.
- ✓ It can have constructors and static methods also.
- ✓ It can final methods which will force the sub-class not to change the body or the method.

** chaining of constructors **

If constructors are created both in the main class and in the sub-class, then the constructor of the main class shall be called first, then the constructor of the child class shall be called, at the time of creating object of the child class.

Eg:-

```
abstract class Animal {
    abstract void walk();
    Animal() {
        System.out.println("Animal created");
    }
}
Public class Horse extends Animal {
    Horse() {
        System.out.println("Horse created");
    }
}
class oops {
    public static void main(String args[]) {
        Horse horse = new Horse();
    }
}
```

O/P:- Animal created
Horse created

(11) using Interfaces

using Interfaces, we can achieve pure abstraction. previously, using abstract class we see that the class have non-abstract method also, which is allowed by Java. However, in Interface we shall use only abstract methods, thus achieving pure abstraction.

- Interfaces cannot have constructors.
- Interfaces cannot have any non-abstract methods.

E.g., → interface Animal {
    public void walk(); → abstract method
}

class Horse implements Animal { → implemented abstract method
    public void walk() {
        System.out.println("walks on 4 legs");
    }
}

class OOPS {
    public static void main(String args[])
    {
        Horse horse = new Horse();
        horse.walk();
    }
}

int eye = 2; ← if this is a variable in the interface, the return type is static, fixed, value can't be changed, it's final and it's accessible to all the child classes.

Properties:-

(i) All fields in Interfaces are public, static and final by default.

(ii) All methods are public & abstract by default.

(iii) A class that implements an interface must implement all the methods declared in that interface.

(iv) Interfaces support the functionality of multiple inheritance.

interface Animal {
    public void walk();
}

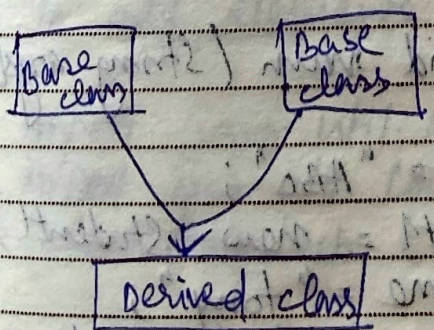
→ we can or can't write public. Both are correct. Because it is by default considered as public.

class Horse implements Animal {
    public void walk() {
        S.o.pln("horse");
    }

we have to mention it as public or else error shall be thrown. The methods derived from the interface has to be public in sub-classes.

Multiple Inheritance

By using class, it is not possible to implement multiple inheritance. However, using Interfaces, it is possible.



There are 2 interfaces Animal & Herbivore. Both of them are extended/implemented by child class 'Horse'.

E.g.

interface Animal {
    void walk();
}

interface Herbivore {
    void eat();
}

class Horse implements Animal, Herbivore {
    Public void walk()
    { S.o.pln("walking"); }
    Public void eat()
    { S.o.pln("eating"); }
}

Static

Static is a special type of method that is accessible to all and whose value remains the same throughout be it used by any classes or sub-classes.

We can use static in 4 different ways -

i) Static variable
ii) Static functions
iii) Static block
iv) Static nested class

E.g.

```
class Student {
    String name;
    static String school;          // static variable

    public static void changeSchool() {
        school = "newschool";
    }
}

public class OOPs {
    public static void main(String args[]) {
        Student.school = "ABC";
        Student student1 = new Student();
        student1.name = "tony";
        System.out.println(student1.school);
    }
}
```

o/p → newschool

(public static function)

note:- For static variables or functions, memory is allocated only once, whereas for object memory is allocated again & again. So, if we need to save memory, we can use static