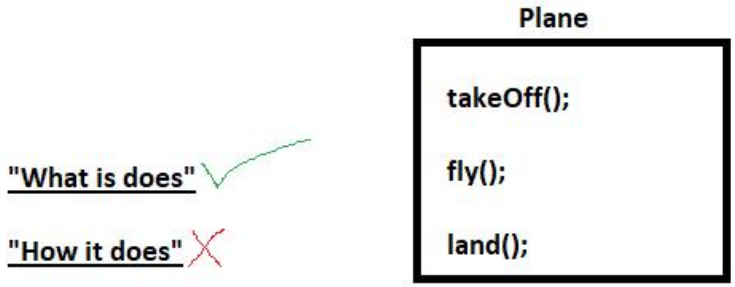



**Abstraction in Java:**

1. Abstract classes - 0% to 100%
2. Interfaces - 100%

**Example-**
**1. Plane Hierarchy**
**2. Animal Hierarchy**
**3. Car Hierarchy**

abstract + final

---

**Abstraction allows us to expose only the neccessary details and hide irrelevant information.**

**"What it does"**
**"How it does"**

**Email**
**SMS**
**ATM**
**Accelerators**
**Switches, etc.**

---

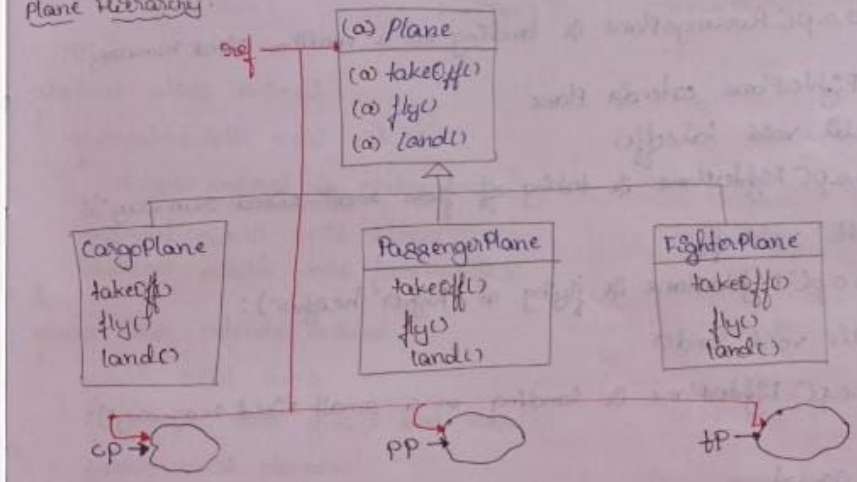
**OOP Application:**
**Non-Object Oriented Version ---> Object-Oriented Version (Shape...)**

---

**Bird Hierarchy**

# Abstraction in Java :

Plane Hierarchy :




```
abstract class Plane
{
    abstract public void takeOff();
    abstract public void fly();
    abstract public void land();
}
class CargoPlane extends Plane
{
    public void takeOff()
    {
        s.o.p(" CargoPlane is taking off from a long-sized runway");
    }
    public void fly()
    {
        s.o.p(" CargoPlane is flying at low heights");
    }
    public void land()
    {
        s.o.p(" CargoPlane is landing on a long-sized runway");
    }
}
class PassengerPlane extends Plane
{
    public void takeOff()
    {
        s.o.p(" PassengerPlane is taking off from medium-sized runway");
    }
    public void fly()
    {
        s.o.p(" PassengerPlane is flying at medium heights");
    }
    public void land()
    {
        s.o.p(" PassengerPlane is landing on a medium-sized runway");
    }
}
class FighterPlane extends Plane
{
    public void takeOff()
    {
        s.o.p(" FighterPlane is taking off from small-sized run-way");
    }
    public void fly()
    {
        s.o.p(" FighterPlane is flying at higher heights");
    }
    public void land()
    {
        s.o.p(" FighterPlane is landing on a small-sized run-way");
    }
}
class Airport
{
    public void permit(Plane ref)
    {
        ref.takeOff();
        ref.fly();
        ref.land();
    }
}
```

overridden methods
(Not specialized methods)

## Abstract classes:

There are situations in which we want to define a super class that only defines a generic form without providing implementation, that will be shared by all of the subclasses, leaving the responsibility of providing the implementation details to each of the subclass. Such class must be declared with the 'abstract' keyword.

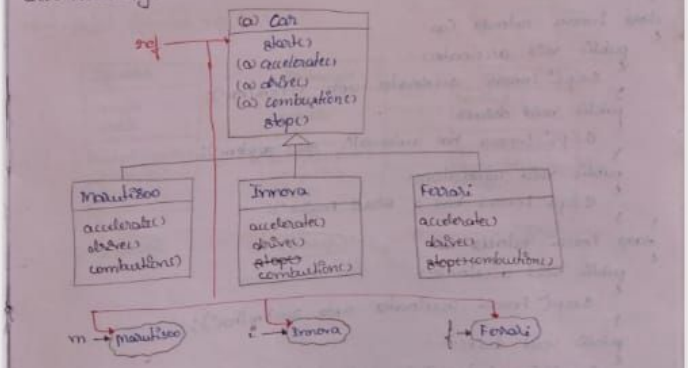
## Abstract methods:

Abstract methods are such methods which contains only the signature of the method but no implementation (method body).

The advantage of abstract methods is that abstract methods ensures that the methods in the child classes do not become specialized methods instead they remain as overridden methods. Hence, loose coupling & polymorphism can be achieved.

NOTE: If a class contains atleast one abstract method then the class must be declared as abstract.

Car Hierarchy:



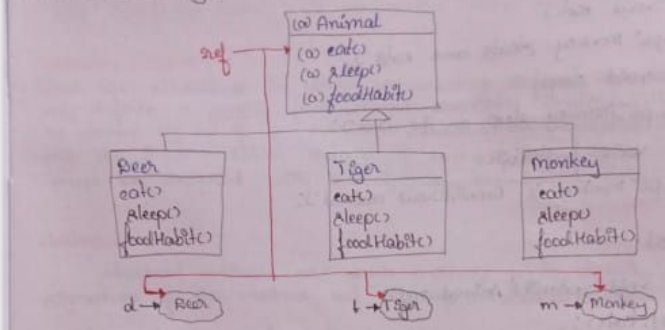

```
abstract class Car
{
    public void start()
    {
        S.o.p("car is starting");
    }
    abstract public void accelerate();
    abstract public void drive();
    abstract public void combustion();
    public void stop()
    {
        S.o.p("car is stopping");
    }
}
class Maruti800 extends Car
{
    public void accelerate()
    {
        S.o.p("Maruti800 accelerates upto 144km/hr");
    }
    public void drive()
    {
        S.o.p("Maruti800 has manual gear system");
    }
    public void combustion()
    {
        S.o.p("Maruti800 has a Petrol Engine");
    }
}
class Innova extends Car
{
    public void accelerate()
    {
        S.o.p("Innova accelerates upto 179km/hr");
    }
    public void drive()
    {
        S.o.p("Innova has automatic gear system");
    }
    public void combustion()
    {
        S.o.p("Innova has a Diesel Engine");
    }
}
class Ferari extends Car
{
    public void accelerate()
    {
        S.o.p("Ferari accelerates upto 340km/hr");
    }
    public void drive()
    {
        S.o.p("Ferari has Turbo gear system");
    }
    public void combustion()
    {
        S.o.p("Ferari has a white-petrol Engine");
    }
}
```

```
class Road
{
    public void permit(Car ref)
    {
        ref.start();
        ref.accelerate();
        ref.drive();
        ref.combustion();
        ref.stop();
    }
}
```

```
class Launch
{
    P.S.V m(String[] args)
    {
        Maruti800 m = new Maruti800();
        Innova i = new Innova();
        Ferari f = new Ferari();
        Road r = new Road();
        r.permit(m);
        r.permit(i);
        r.permit(f);
    }
}
```

```
class Launch
{
    psvm(String[] args)
    {
        CargoPlane cp = new CargoPlane();
        PassengerPlane pp = new PassengerPlane();
        FighterPlane fp = new FighterPlane();
        Airport a = new Airport();
        a.permit(cp);
        a.permit(pp);
        a.permit(fp);
    }
}
```

Animal Hierarchy:




```
abstract class Animal
{
    abstract public void eat();
    {
        S.o.p("Animal is eating....");
    }
    abstract public void sleep();
    abstract public void foodHabit();
}
class Deer extends Animal
{
    public void eat()
    {
        S.o.p("Deer grazes and eats");
    }
    public void sleep()
    {
        S.o.p("Deer is sleeping under the tree");
    }
    public void foodHabit()
    {
        S.o.p("Deer is Herbivorous animal");
    }
}
class Tiger extends Animal
{
    public void eat()
    {
        S.o.p("Tiger hunts and eats");
    }
    public void sleep()
    {
        S.o.p("Tiger is sleeping in the cave");
    }
    public void foodHabit()
    {
        S.o.p("Tiger is carnivorous animal");
    }
}
```

```
class Monkey extends Animal
{
    public void eat()
    {
        S.o.p("Monkey steals and eats");
    }
    public void sleep()
    {
        S.o.p("Monkey sleep on the tree");
    }
    public void foodHabit()
    {
        S.o.p("Monkey is Omnivorous animal");
    }
}
class Forest
{
    public void permit(Animal ref)
    {
        ref.eat();
        ref.sleep();
        ref.foodHabit();
    }
}
class Launch
{
    psvm(String[] args)
    {
        Forest f = new Forest();
        f.permit
        Deer d = new Deer();
        Tiger t = new Tiger();
        Monkey m = new Monkey();
        f.permit(d);
        f.permit(t);
        f.permit(m);
    }
}
```

INTERFACES → "STANDARDISATION"

contract interface Connection

```
getConnection();
executeQuery();
close();
```

Full-Stack Application

Web Applications
(Internet - 1990s)


HTML,
CSS, Bootstrap,...
JavaScript,
Angular, React, ....

Java
Spring
Hibernate,
etc.

Python
Django,
etc.

C#,...
.Net,...

SQL
Oracle

NoSQL
MongoDB

Programming Language

Database (DBMS)


```
class OracleDB implements Connection
{
    getConnection()
    openConnection()
    executeQuery()
    processQuery()
    close()
    closeConnection()
}

class SynbaseDB
{
    getConnection()
    runQuery()
    terminateConnection()
}

class DB2DB
{
    establishConnection()
    executeQuery()
    stopConnection()
}
```

Java 1.0 ✗ Database

Java 1.1 X

Java 1.2 ----> JDBC API