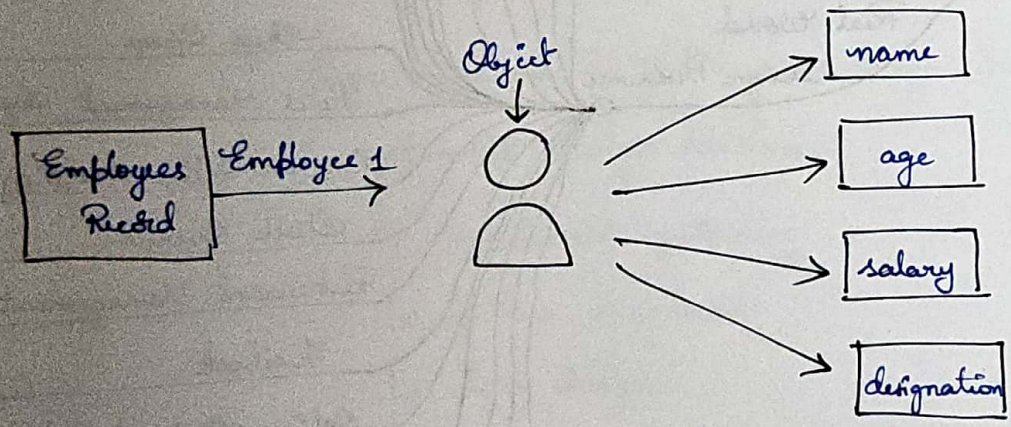
# 1. Background of Object-Oriented Programming (OOP)

* Definition
* Building blocks of OOP.
  - ↳ Classes and objects.
  - ↳ Attributes
  - ↳ Methods
* Principles of OOP.

## 1. Definition:-

Object-oriented programming, also called OOP, is a programming model that is dependent on the concept of objects and classes.




Fig, Real-life example of an employee record.

## 2. Building blocks of OOP.

* Attributes
* Methods
* Classes
* Objects

ⓐ **Classes and objects :-**

All objects have some state and behaviour.

Eg:- Calculator

- state : ON/OFF
- behaviour : addition, subtraction, multiplication, division etc.

A class can be thought of as a blueprint for creating objects.

ⓑ **Attributes :-**

Attributes are variables that represent state of the object.

ⓒ **Methods :-**

Methods represent the behaviour of the object.

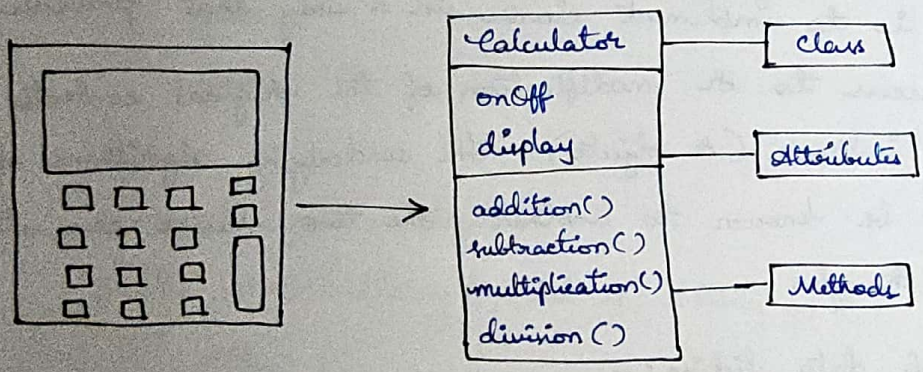



fig., Calculator class with attributes and methods

2. **Principles of OOP:-**

① Encapsulation

② Abstraction.

③ Inheritance

④ Polymorphism.

# 11. Encapsulation

1. Data hiding
2. Components of data hiding.
3. Encapsulation.
   * Implementing encapsulation in programming languages.
   * Advantages of encapsulation.

## 1. Data Hiding :-

It is an essential concept in object-oriented programming. It can be defined as masking a class's internal operations and only providing an interface through which other entities can interact with the class without being aware of what is happening within.

The goal is to implement classes in a way that prevents unauthorized access to or modification of the original contents of a class by its instances (or objects). The underlying algorithms of one class need not be known to another. The two classes can still communicate, though.

## 2. Components of data hiding :-

Data hiding can be divided into :

1) Encapsulation.
2) Abstraction.

## 3. Encapsulation :-

Encapsulation in OOP refers to binding data and the methods to manipulate that data together in a single unit - class.

- usually done to hide state and representation of an object from the outside.

* Implementing Encapsulation in programming languages.

class - Movie

attributes - title, year, genre.

```
class Movie {

    //Data members.
    private String title;
    private int year;
    private String genre;

    //Default constructor
    public Movie() {
        title = "";
        year = -1;
        genre = "";
    }
    //Parameterized constructor
    public Movie(String t, int y, String g) {
        title = t;
        year = y;
        genre = g;
    }
```

Data member initialization.

```
    //getters and setters
    public String getTitle(){
        return title;
    }
    public void setTitle(String t){
        title = t;
    }

    public String getGenre(){
        return genre;
    }
    public void setGenre(String g){
        genre = g;
    }

    public int getYear(){
        return year;
    }
    public void setYear(int y){
        year = y;
    }
```

* Advantages of encapsulation :-

→ Classes are simpler to modify and maintain.

→ Which data member we wish to keep hidden & accessible can be specified.

→ We choose which variables are read-only and write-only (increases flexibility).

## III. Abstraction

1. Definition.
2. Example.
3. Implementation of abstraction in programming languages.
4. Advantages of abstraction.
5. Abstraction vs. encapsulation.

1. Definition :-

Abstraction is a technique used in object-oriented programming that focuses only on revealing the necessary details of a system and hiding irrelevant information to minimize the complexity. In other words, it means to show what an object does and hides how it does it.

2. Examples :-

→ TV remote's volume up button

→ Race peddle of a vehicle.

③ Implementation of abstraction in programming languages.

```
class Circle {

    //define data attributes
    private double radius;
    private double pi;

    //define constructors
    public Circle() {
        radius = 0;
        pi = 3.142;
    }
    public Circle(double r) {
        radius = r;
        pi = 3.142;
    }

    //define methods
    public double area() {
        return pi * radius * radius;
    }
    public double perimeter() {
        return 2 * pi * radius;
    }
}.

public class Main {

    psvm() {
        Circle circle = new Circle(5);
        sout("Area: " + circle.area());
        sout("Perimeter: " + circle.perimeter());
    }
}
```

As you can see, we only need to define radius in the constructor. After that, the area() and perimeter() functions are available to us. This interface is part of encapsulation.

We use functions to calculate the area and perimeter. Users do not need to know implementation details of the functions. Even pi is hidden since it's a constant. This is how we achieve abstraction.

④ Advantages of Abstraction :-

(1) Reduces complexity of the system from a user's perspective.
(2) Makes the code extendable and reusable.
(3) Refines the modularity of the application or the system.
(4) Makes code more maintainable.

⑤ Abstraction vs Encapsulation :-

| Abstraction | Encapsulation. |
|---|---|
| ① It focuses on design level of the system | ① It focuses on application level of the system. |
| ② It hides unnecessary data to simplify the structure. | ② It restricts access to the data to prevent its misuse. |
| ③ It highlights the work that the object performs. | ③ It deals with the internal working of the object. |
| ④ Abstraction means to hide implementation using interface and abstract classes. | ④ Encapsulation means to hide data using getter and setter methods. |

# IV. INHERITANCE

1. Definition.
2. The IS-A relationship.
3. Modes of inheritance.
4. Types of inheritance.
   - (a) Single inheritance
   - (b) Multiple inheritance
   - (c) Multilevel inheritance
   - (d) Hierarchical inheritance
   - (e) Hybrid inheritance.
5. Implementation.
6. Advantages of inheritance.

---

## (1) Definition :-

Inheritance provides a way to create new class from an existing class. The new class is a specialized version of the existing class such that it inherits all the public attributes (variables) and methods of the existing class.

## (2) The IS-A relationship :-

Whenever we come across IS-A relationship between objects, we can use inheritance.

| Square | Dog | Car |
|---|---|---|
| IS-A | IS-A | IS-A |
| Shape. | Animal | Vehicle. |

| Existing class | Derived class |
|---|---|
| Shape | Square |
| Animal | Dog. |
| Vehicle. | Car. |

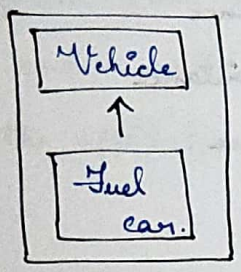
③ Modes of inheritance :-

Access modifiers are tags we can associate with each member to define the parts of the program they can access directly. By using these modifiers, we define the scope of the data members and member functions for the other classes and main.

④ Types of inheritance :-

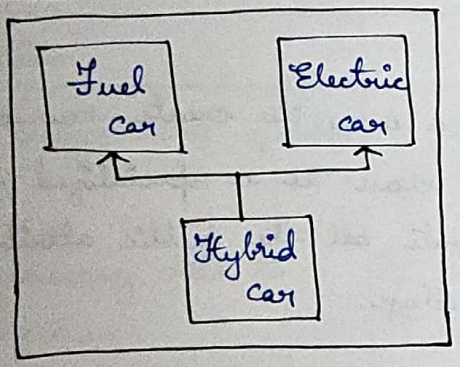
5 types.

(a) Single inheritance :




A fuel car is a vehicle.

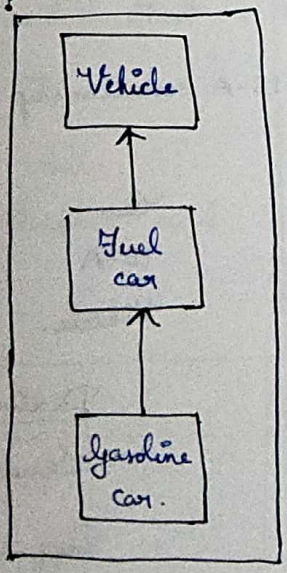
(b) Multiple inheritance :




The hybrid car IS-A fuel car.

The hybrid car IS-A electric car as well.

(c) Multi-level inheritance :



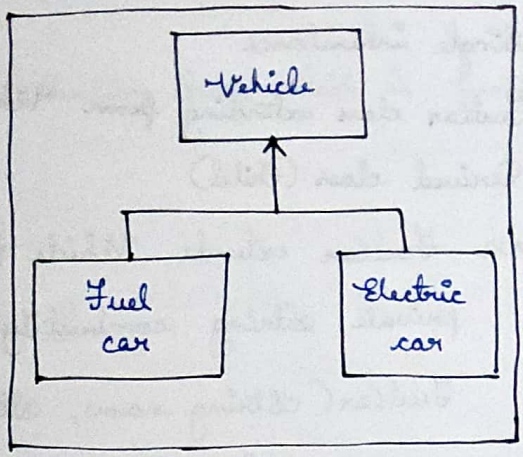

A fuel car is-a Vehicle

A gasoline car is-a Fuel car.

(d) Hierarchical inheritance :-

A fuel car IS-A vehicle.

An electric car IS-A vehicle.

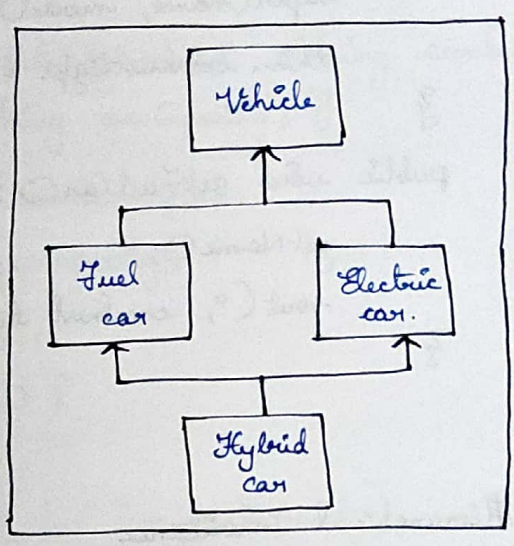



(e) Hybrid inheritance :-

A fuel car IS-A vehicle.

An electric car IS-A vehicle.

A hybrid car IS-A fuel car and IS-A electric car.




(5) Implementation :-

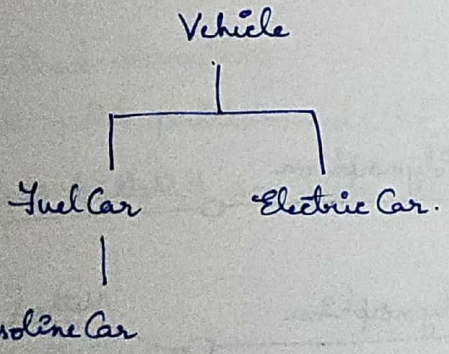
```
class Vehicle {
    private String name;
    private String model;
    Vehicle(String name, String model){
        this.name = name;
        this.model = model;
    }
    public void getName(){
        sout("The car is a "+name+" "+model);
    }
}
```

```
//Single inheritance
// FuelCar class extending from Vehicle class
// Derived class (Child)
class FuelCar extends Vehicle {
    private String combustType;
    FuelCar(String name, String model, String combustType){
        super(name, model);
        this.combustType = combustType;
    }
    public void getFuelCar() {
        getName();
        sout(", combust type is: " + combustType);
    }
}.

// Hierarchical inheritance
// alongside the Fuelcar class, the ElectricCar class is also extending from
// Vehicle class.
// Another Derived class (Child)
class ElectricCar extends Vehicle {
    private String batteryPower;
    ElectricCar(String name, String model, String batteryPower) {
        super(name, model);
        this.batteryPower = batteryPower;
    }
    public void getElectricCar() {
        getName();
        sout(", battery power is " + batteryPower);
    }
}
```

// Multi-level inheritance
// Gasoline Car is derived from the FuelCar class, which is further derived
// from the Vehicle class.
// Derived class (Grandchild)

```
class GasolineCar extends FuelCar {
    private String combustType;
    private String gasCapacity;
    GasolineCar (String name, String model, String combustType,
                 String gasCapacity) {
        super(name, model, combustType);
        this.gasCapacity = gasCapacity;
    }.
    public void getGasolineCar() {
        getName();
        sout("; gas capacity is: " + gasCapacity);
    }
```




⑥ Advantages of inheritance.

Reusability.

Code modification.

Extensibility.

Data hiding.

# V. POLYMORPHISM

① Introduction to polymorphism

② Types of polymorphism.

(a) Dynamic polymorphism.

* Method overriding

(b) Static polymorphism.

* Method overloading.

* Operator overloading.

(c) Dynamic polymorphism vs. Static polymorphism.

---

① Introduction to polymorphism.

poly - many  morph - forms.

In programming, polymorphism is a phenomenon that allows an object to have several different forms and behaviours.

Eg:- "Animal" class can have method "makeNoise" whose implementation should be different for lion, deer, etc., This is called Polymorphism.

② Types of polymorphism.

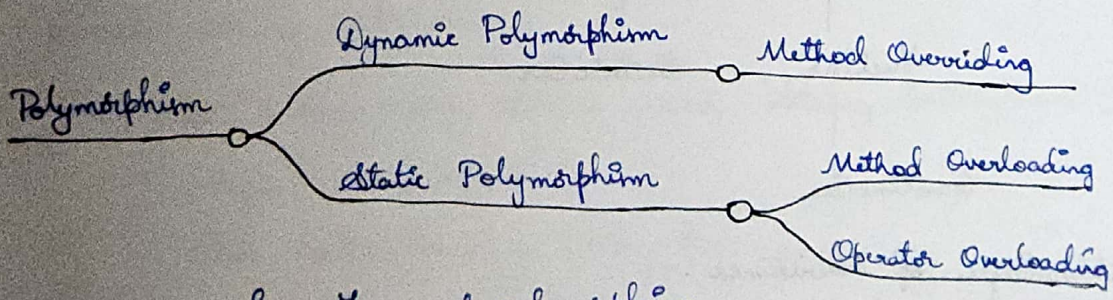



Fig., Types of polymorphism.

(2) Dynamic Polymorphism :- is the mechanism that defines the methods with the same name, return type and parameters in the base class and derived classes. Hence, the call to an overriden method is decided at runtime. That is why, dynamic polymorphism is also known as runtime polymorphism. It is achieved by method overriding.

* Method overriding :-

In object-oriented programming, if a subclass provides a specific implementation of a method that had already been defined in one of its parent classes, it is known as method overriding.

```
class Animal {
    public void printAnimal() {
        sout("I am from the Animal class");
    }
    void printAnimalTwo() {
        sout("I am from the Animal class");
    }
}
class Lion extends Animal {
    //method overriding
    public void printAnimal() {
        sout("I am from the Lion class");
    }
}
public class Main {
    main() {
        Animal animal;
        Lion lion = new Lion();
        animal = lion;
        animal.printAnimal();
        animal.printAnimalTwo();
    }
}
```

(b) Static Polymorphism:- also known as compile-time polymorphism and it is achieved by method overloading or operator overloading.

* Method Overloading:-

Methods are said to be overloaded if a class has more than one method with the same name, but either the number of arguments is different, or the type of arguments is different.

```
class Sum {

    public int addition (int a, int b) {
        return a+b;
    }
    public int addition (int a, int b, int c) {
        return a+b+c;
    }
    public double addition (double a, double b) {
        return (a+b);
    }
}.
```

* Operator Overloading:-

Operators can be overloaded to operate in a certain user-defined way. Its corresponding method is invoked to perform its predefined function whenever an operator is used.

Note:- Java doesn't support operator overloading (except for '+' operator).

(c) Dynamic polymorphism vs. static polymorphism.

| Static Polymorphism | Dynamic Polymorphism |
|---|---|
| Polymorphism that is resolved during compile-time is known as static polymorphism. | Polymorphism that is resolved during runtime is known as dynamic polymorphism. |
| Method overloading is used in static polymorphism. | Method overriding is used in dynamic polymorphism. |
| Mostly used to increase the readability of the code. | Mostly used to have separate implementation for a method that is already defined in base class. |
| Arguments must be different in the case of overloading. | Arguments must be the same in the case of overriding. |
| Return type of the method does not matter. | Return type of the method must be the same. |
| Private and sealed methods can be overloading. | Private and sealed methods cannot be overridden. |
| Gives better performance because the binding is being done at compile-time. | Gives worse performance because the binding is being done at runtime. |

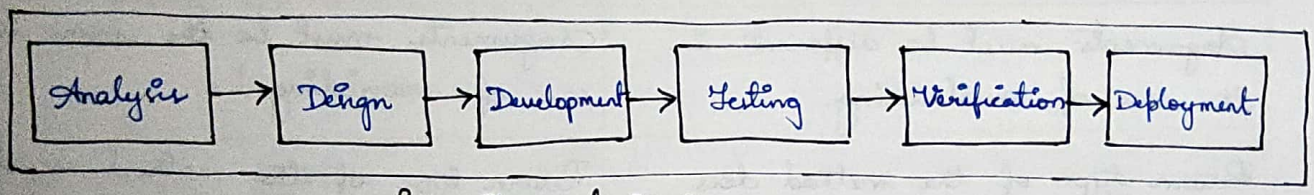
# III. OBJECT-ORIENTED DESIGN.

## 1. Introduction to Object-oriented Analysis and Design (OOAD)

1. OOAD in the SDLC.
2. Object-oriented analysis
3. Object-oriented design.
4. Advantages of OOAD.

### (1) OOAD in the SDLC :-

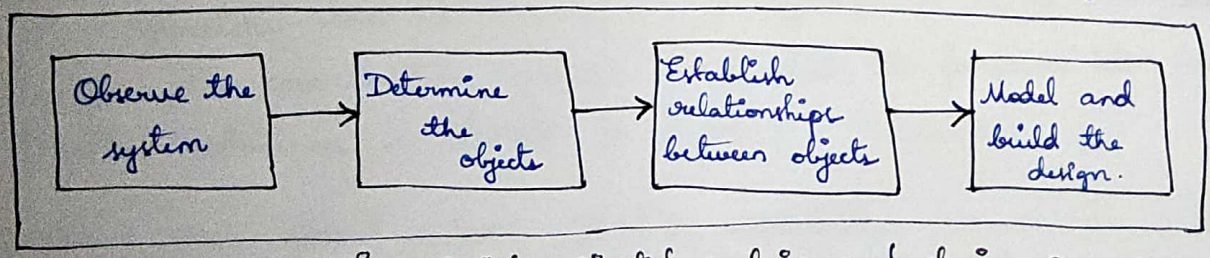
The Software Development Life Cycle (SDLC) refers to the sequence of phases involved in software development. The stages of SDLC are shown below:




Fig, Stages of SDLC

The Object-oriented analysis and design (OOAD) in SDLC allows one to analyze and design a system using object-oriented concepts and visualize each aspect of the OOAD process.

This entire process is summarized in the following diagram:




Fig, Object-oriented analysis and design process.

### (2) Object-oriented analysis.

Object-oriented analysis is the method where the system requirements are identified, which are then used to create a model based on defining the roles of the objects present in the system and tasks for which the system is responsible. The analysis step excludes any implementation details. Rather, we identify different use cases here.

(3) Object-oriented design :-

Object-oriented design is the phase where the implementation of the defined requirements and the models created during the analysis phase begins.

In this step, models are further refined by adding additional constraints and making decisions such as adding new elements to the given initial structure.

At this stage, the designer also goes into the inner details of the ~~design~~ defined models and elaborates if any object-oriented principle is required.

These models are usually built using UML diagrams such as class or sequence diagrams.

(4) Advantages of OOAD :-

OOAD process is well-known and following are reasons:

* It's extremely easy to understand, which helps in creating models of complex problems.
* Concepts such as inheritance help make the data reusable and scalable
* It's easily maintainable, which helps identify the issues in the early processes and saves time.