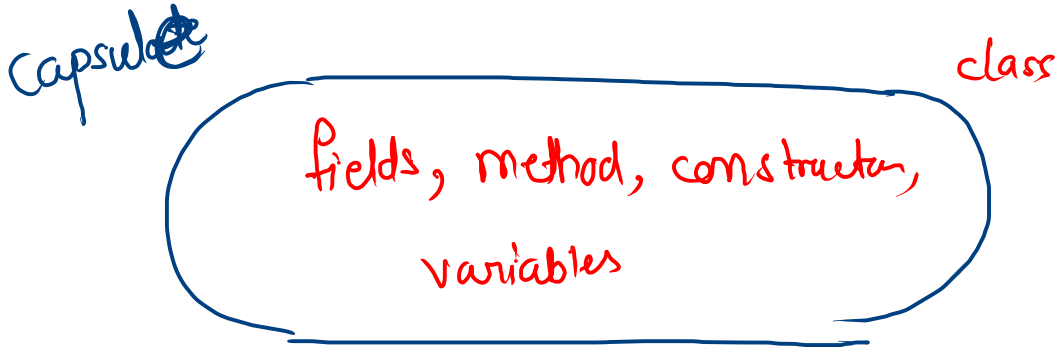
⇒ Encapsulation (Data Binding)

↳ Data encapsulation can defined as wrapping the code or method and related fields together as a single unit.



Encapsulation is required so that an unauthorised person should not be able access field, methods, etc.

↳ But how an authorised person will access those fields

answer is using getters and setters

Code

```java
class Person {
    private String name; // unable to access directly
    private int age;

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

    // create getters and setters
    // getters
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }

    // setters
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```java
public class Main { // client
    public static void main(String[] args) {
        Person obj = new Person("Kunal", 25);

        System.out.println("Name : " + obj.getName());
        System.out.println("Age : " + obj.getAge());

        obj.setName("John");
        obj.setAge(30);

        System.out.println("Name : " + obj.getName());
        System.out.println("Age : " + obj.getAge());
    }
}
```

Encapsulation

output

```
Finished in 68 ms
Name : Kunal
Age : 25
Name : John
Age : 30
```

1) Private attributes :-

attributes of a class are declared as private to restrict direct access from outside the class

2) Public methods :-

Public methods, such as getters and setters are provided to access and modify the private attributes.

3) Controlled access :-

Outside class interact with the object's attributes only through these public method.

This way, the internal representation of data is hidding and class can enforce its own rules and validations.

⇒ advantages :-

- Data hiding
- Enhance security
- maintanibility
- Code reusability
- Easier testing
- facilitates abstraction

⇒ Inheritence

↳ Inheritence is a OOP's concept where one class acquires the properties and behaviour of another class.

→ Subclass / Child class / Derived class

→ Superclass / Parent class / Base class

⇒ Types of Inheritence

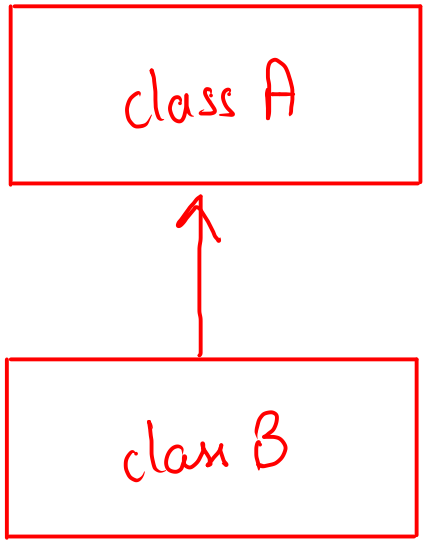
- single level inheritence
- multilevel inheritence
- Hybrid inheritence
- Hierarchical inheritence
- Multiple Inheritence (not allowed in java language)

example of Inheritence

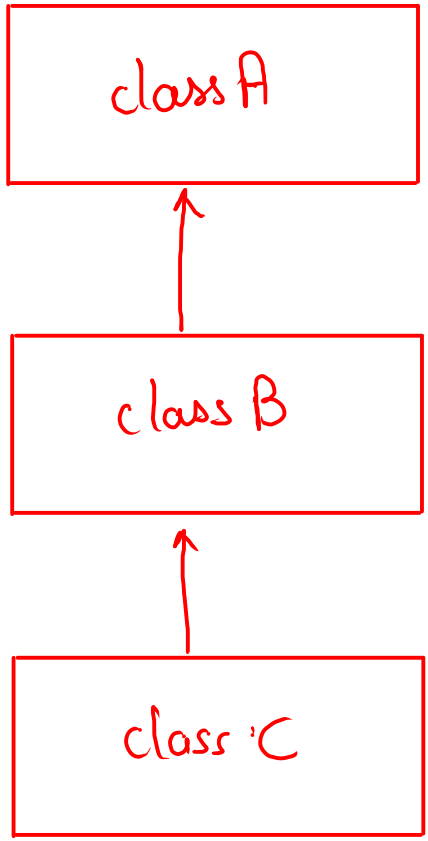
```java
// parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
// child class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
public class Main { // client
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.bark();
        myDog.eat();
    }
}
```

Diagram

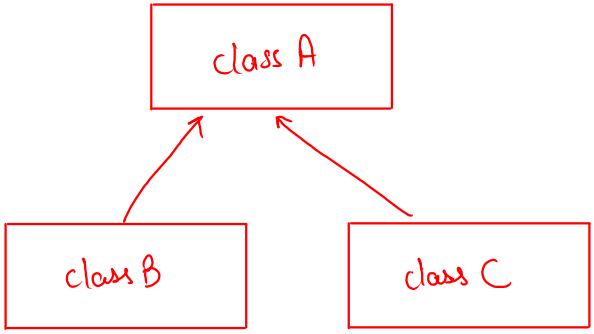
Single inheritence



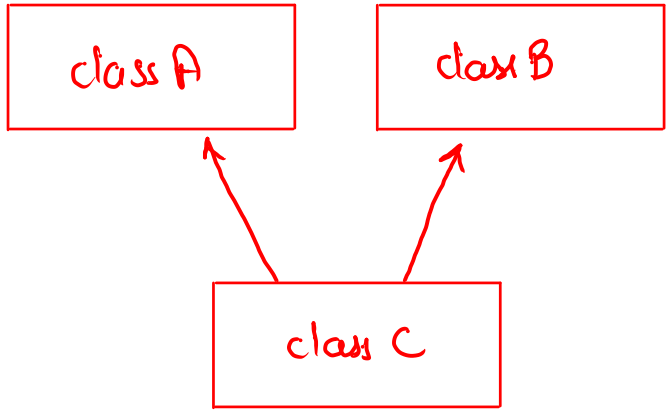
multiple inheritance



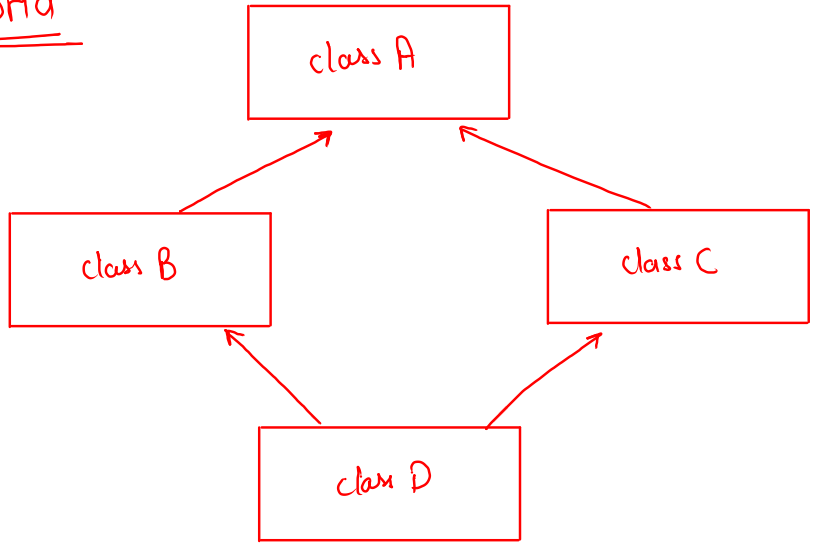
Hierarchical



Multiple inheritence



Hybrid



Imp

Diamond Pattern Problem

⇒ Single inheritence

```java
// parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
// child class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
public class Main { // client
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.bark();
        myDog.eat();
    }
}
```

output

```
Dog is barking
Animal is eating
```

⇒ Multi-level inheritence

Code

```java
// parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
// child class inheriting animal class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// sub child class inheriting Dog
class DesiKutta extends Dog {
    void display() {
        System.out.println("Desi kutta is type of dog");
    }
}
public class Main { // client
    public static void main(String[] args) {
        DesiKutta dog = new DesiKutta();
        dog.eat();
        dog.bark();
        dog.display();
    }
}
```

⇒ Hierarchical Inheritence

↳ when 2 different classes have same parent.

```java
// parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
// child class inheriting animal class
class Dog extends Animal { // jackie
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Child class inheriting Animal
class Cat extends Animal { //pucchu
    void meow() {
        System.out.println("Cat is meowing");
    }
}
```

```java
public class Main { // client
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
        dog.bark();

        Cat cat = new Cat();
        cat.eat();
        cat.meow();
    }
}
```

output

```
Finished in 60 ms
Animal is eating
Dog is barking
Animal is eating
Cat is meowing
```

→ Issue with multiple inheritence

```java
class Cat {
    void eat() {
        System.out.println("Cat is eating");
    }
    void meow() {
        System.out.println("Cat is meowing");
    }
}

class Dog {
    void eat() {
        System.out.println("Dog is eating");
    }
    void bark() {
        System.out.println("Dog is barking");
    }
}

class Animal extends Dog, Cat {

}
```

when animal object is calling eat function, then which eat function to print

that's why error

⇒ Hybrid inheritence (not supported with classes)

↳ Hybrid inheritence in java refers to combination of multiple and hierarchical inheritence.

```java
// parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
// child class inheriting animal class
class Dog extends Animal { // jackie
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Child class inheriting Animal
class Cat extends Animal { //pucchu
    void meow() {
        System.out.println("Cat is meowing");
    }
}
```

```java
// multiple inheritence --> using interface

interface Domestic {
    void play();
}

interface DogBehaviour  {
    void guard();
}

class DomesticDog implements DogBehaviour, Domestic {
    public void play() {
        System.out.println("Dog is playing");
    }
    public void guard() {
        System.out.println("Dog is guarding");
    }
}
```

Imp

Classes can extends other classes

Interface can extends other interfaces

Class can implements interfaces

Diamond Pattern Problem

```
class A {
    void fun() {
        System.out.println("hi1");
    }
}

class B {
    void fun() {
        System.out.println("hi2");
    }
}

class Main extends A, B {
    public static void main(String[] args) {
        Main obj = new Main();
        obj.fun();
    }
}
```

Solution using interfaces

```
interface A {
    void fun();
}

interface B {
    void fun();
}

class C implements A, B {
    public void fun() {
        System.out.println("Hi");
    }
}
```

⇒ Advantages

- Code Reusability
- Maintenance
- Organizable
- Polymorphism
- Efficiency

etc.