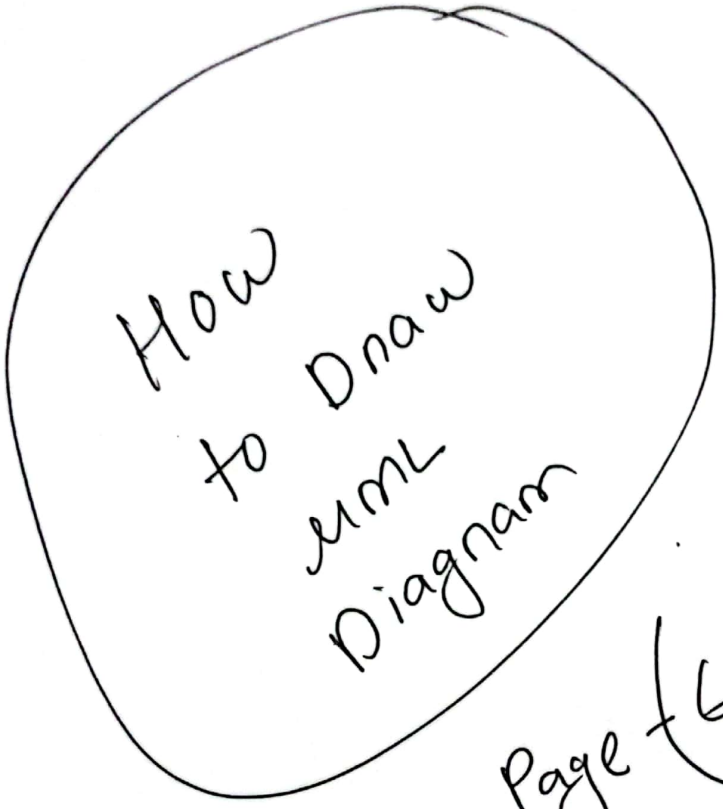
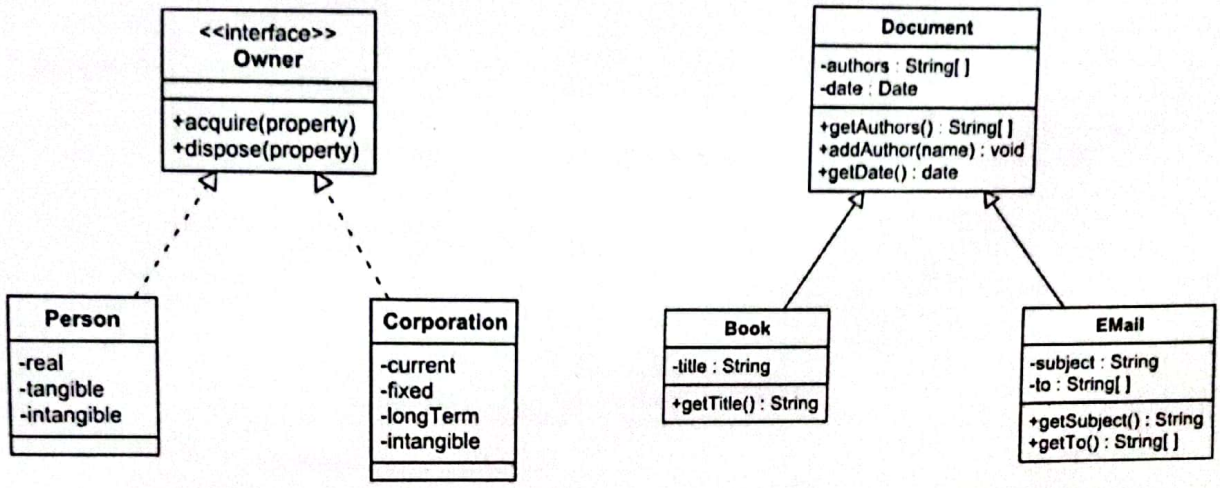


Page (6-7)

A class extends another class. For example, the Book class might extend the Document class,

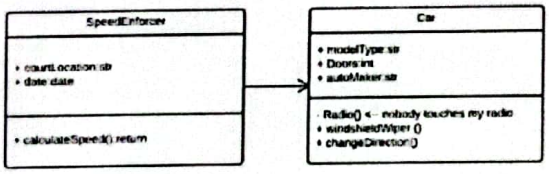


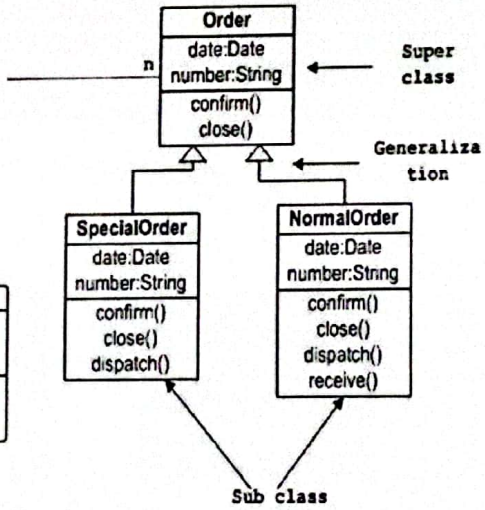
A class implements an interface.

Member access modifiers

All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding symbols:

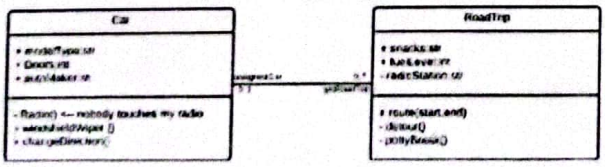
- Public (+)
- Private (-)
- Protected (#)
- Package (-)
- Derived (/)
- Static (underlined)

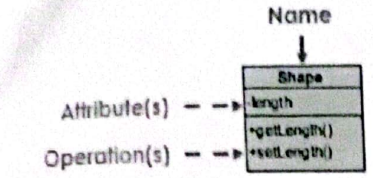




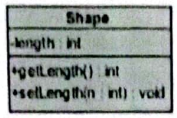
Unidirectional association: A slightly less common relationship between two classes. One class is aware of the other and interacts with it. Unidirectional association is modeled with a straight connecting line that points an open arrowhead from the knowing class to the known class.

- **Bidirectional association:** The default relationship between two classes. Both classes are aware of each other and their relationship with the other. This association is represented by a straight line between two classes.



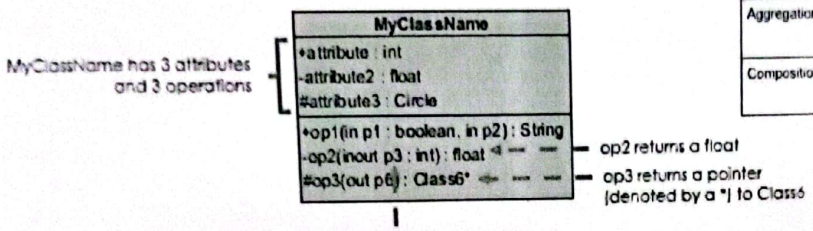


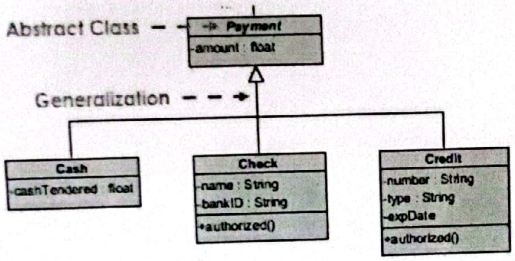
Class without signature

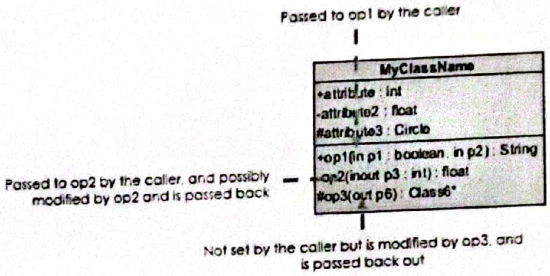


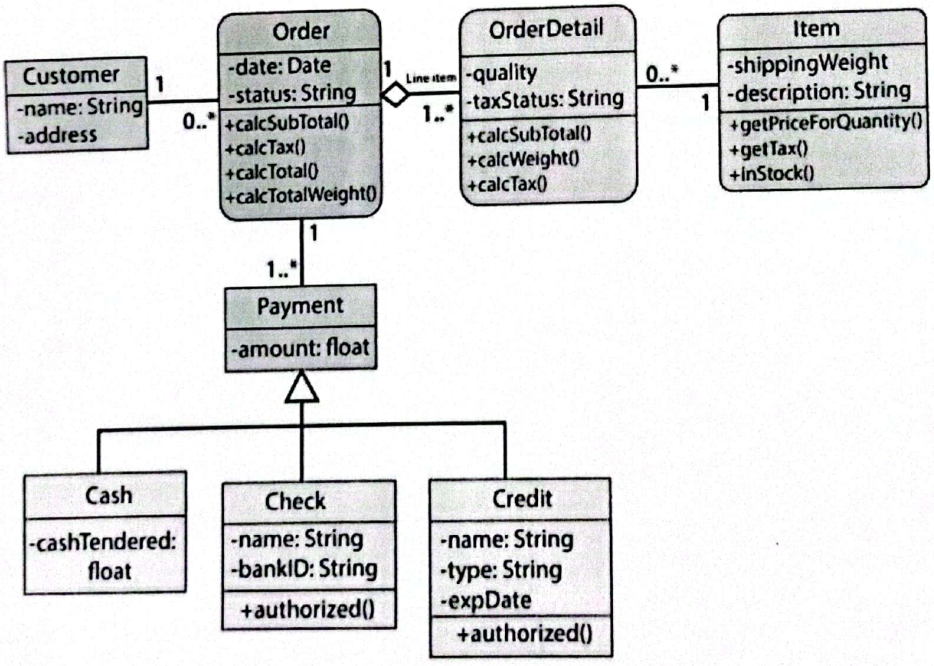
Class with signature

Class Diagram Relationship Type	Notation
Association	——→
Inheritance	——▷
Realization/ Implementation	- - - - ▷
Dependency	- - - - →
Aggregation	——◇
Composition	——◆









Singleton Design pattern

1. **Intent** :- " a class has only onE instance while providing a global access point to the instance "

what do you mean by that?

1. → There will be only one object of class in entire application (cannot have multiple object)

2. → Instead of creating a new object, using the old one which already have been created

3. → global access point means the instance can be accessed from anywhere in the application

2. Motivation

what is the motivation of singleton design pattern?

→ motivated by the need to ensure a that only one instance of class exists in a program providing a global access point to that instance

→ To control the instantiation process of class & limit the number of instance of class to one

→ Common Reason
- → Resource sharing
- → Global access
- → controling object creation

3 **Applicability :-**

what is the applicabity of singleton, where to use it ?

(i) **Resource management :-** when you have a limited resource that need to be shared in multiple parts of system.

Ex:- Database Connection object, file system access, a thread pool

(ii) **Global access :-** If a class need globally accessible throught the application

Ex:- a logger, a cache, configuration manager

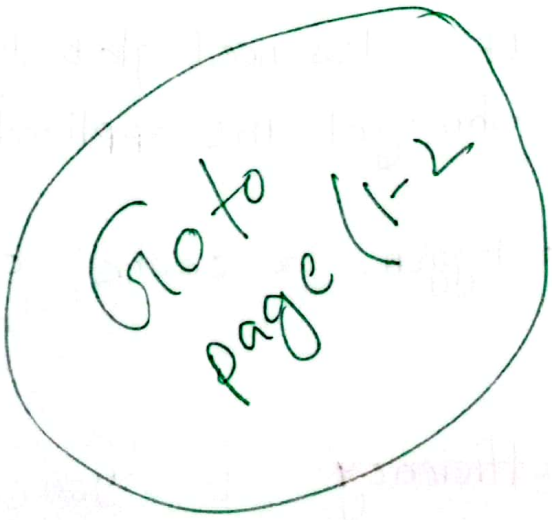
(iii) **less costly and efficiency System :-** you can instantiate the object once and reuse it throught the program

Pros (Problem that singleton solves)

1. ensure a class has just single instance
2. provide a global access point to the instance
3. object is initialize only once

Cons

1. Violates Single Responsibility principle



P-1

Here's what the implementation might look like before using the Singleton Design Pattern:

☹ Problem

```
public class Client {

    public static void main(String[] args) {
        Database s1 = new Database("Mridul");
        System.out.println(s1.getUserName());

        Database s2 = new Database("Promi");
        System.out.println(s2.getUserName());
    }
}
```

```
public class Database {

    private String username;

    // public Constructor
    public Database(String username) {
        this.username = username;
    }

    public  String getUserName(){
        return username;
    }
}
```

To implement the Singleton Design Pattern in these examples, we will make the following changes to our class:

☺ Solution

1. Create a private static instance variable of the class type.

```
public class Database {

    // Create a private static instance variable of the class type. (1)
    private static Database instance;
    private String username;

    // private constructor to prevent instantiation from outside the class (2)
    private Database(String username) {
        this.username = username;
    }

    // Provide a public static method to get the instance of the class (3)
    public static Database getInstance(String username) {
        if (instance == null) {
            instance = new Database(username);
        }
        return instance;
    }

    public  String getUserName(){
        return username;
    }
```

2. Make the constructor private to prevent instantiation from outside the class.

3. Provide a public static method to get the instance of the class. If an instance of the class already exists, return it. Otherwise, create a new instance and return it.

4. Now in Client class, we can now use the Database class by calling the public static method("getInstance") to get the single instance of the class.

```
public class Client {

    public static void main(String[] args) {
        Database s1 = Database.getInstance("Mridul");
        System.out.println(s1.getUserName());

        Database s2 = Database.getInstance("Promi");
        System.out.println(s2.getUserName());
    }
}
```

Now the Output of Client Class

```
Output :
        Mridul
        Mridul
```

P-2

Let's Understand the whole process.

In this implementation, the Database class has a private constructor, which means that it can only be instantiated from within the class itself. By making the constructor private, we ensure that no other instances of the class can be created and prevent other objects from using the new operator with the Singleton class.

We have also created a public static instance variable to hold the single instance of the class, and a public static getInstance() method to get the instance. The getInstance() method checks if an instance of the class already exists. If it does, it returns that instance. If it doesn't, it creates a new instance and returns it. Providing a public static method called "getInstance", *creates a global access point to that instance*. The first time this method is called, it creates a new instance of the class. Subsequent calls to this method will return the same instance.

In output we can see even if we have created another object "s2" of the Database Class, we are getting the same instance "Mridul" created by object "s1".

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

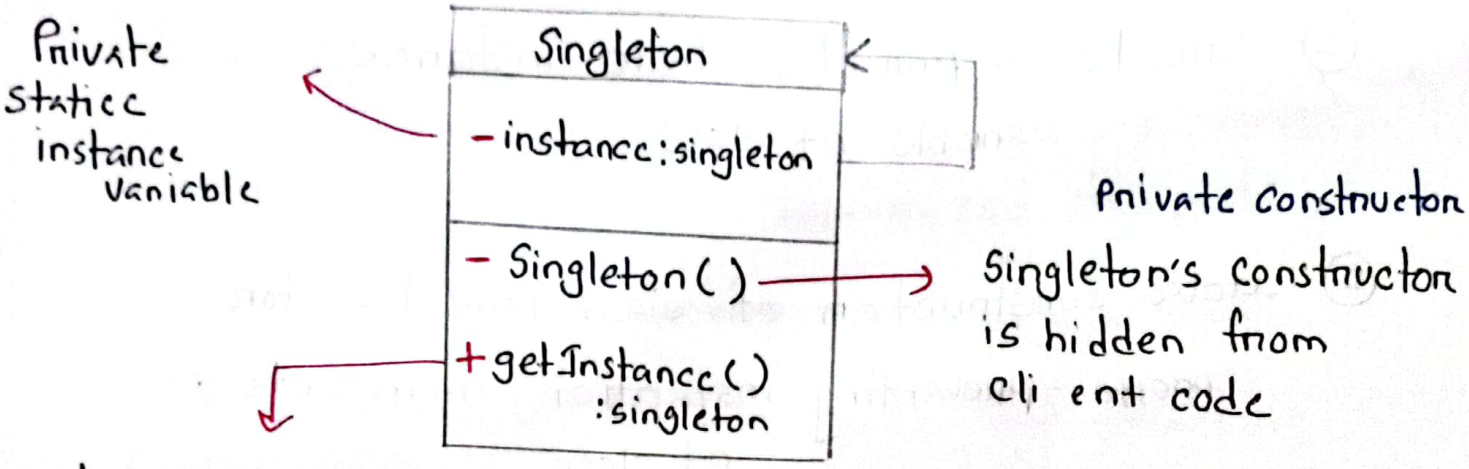
Creational Patterns

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Creational design patterns are concerned with **the way of creating objects.**

Factory Method | Abstract Factory | Singleton | Prototype | Builder

UML Diagram of singleton



Q:- what are participant object of singleton ?

1. Singleton :- provides getInstance()
2. Client :- - request to the instance of singleton using getInstance()
   - use it to access or perform operation resource

How to implement singleton design pattern ?

1. Create a private static instance variable of class

2. Make constructon of class private for preventing instantion from outside of class

3. Providing a public static getinstance() method of class ; returning the instance of class
(if no instance, create a new instance first)

4. In client code, now call getinstance() method of class instead of direct calls to constructor

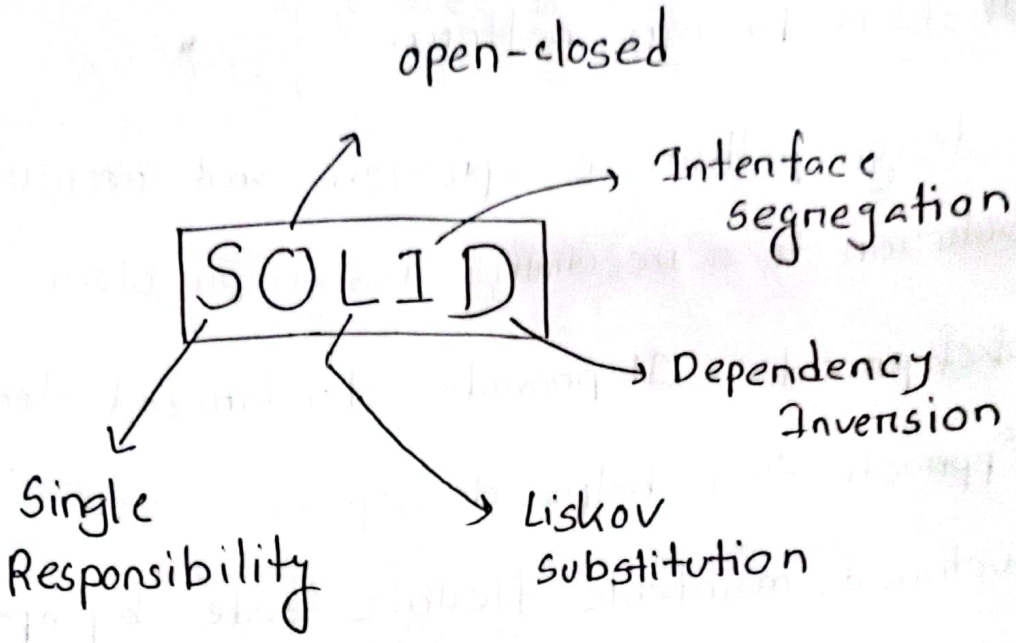
what is Design pattern ?

a design pattern is a proven and reusable solution to a recurring design problem in software development. It provides standarized template or approach that helps developers create a well structured, maintable, flexible code by optimizing the best pratices and collective experience

what does pattern consist of ?

1. Intent (describes pattern with problem and solution)
2. Motivation
3. Structure (UML Diagram)
4. code Example

Principles


open-closed
SOLID
Interface segregation
Dependency Inversion
Single Responsibility
Liskov substitution

(SRP) Single Responsibility Principle

Principle :- states that a class should have only one reason to change

multiple Responsibility

what do you mean?

a class have only single responsib.. or single functionality or single purpose of job

```
publicc class BankService
{
    public void login (String User, int pass)
    {
    }
    public void deposit (string Account, int amount)
    {
    }
    public void withdraw ( string Account, int amount
    {
    }
}
```

Here, BankService class has the responsibility of login, deposit and withdraw. It means it has more that one responsibility which violates the single Responsibility principle.

To achieve the goal of SRP Here,

You need to divide the responsibility to different class

∴ Bank class divided into 3 different class

1. BankService Login class
2. BankService Deposit class
3. BankService Withdraw class

New code

① Class-1

```
public class BankService_Login {
    public void login (string user, int pass) { }
}
```

② class-2

```
public class BankService_Deposit {
    public void deposit ( string Account, int amount)
    { }
}
```

③ class-3

```
public class BankService_withdraw {
    public void withdraw ( string Account, int
                                        amount)
    { }
}
```

(ocp) open closed principle

principle states that software entities (class, modules, functions) should be open for extensions but closed for modification

→ what mean?

able ta extend class behaviour without modifying it

need to modify if want any extentension

```
Public class payment {

    public void bkash ( string id , int amount, int pass)
        {
        }

    public void card ( string id , int amount, int pss)
        {
        }

}
```

Here, if i want to add "nagad payment" in payment service (to add extension), I need to modify the payment service class. which Violates open closed principle.

To achieve the goal of open-closed principle :-

① Creating a payment service Interface which can be easily extended without modifying in future

```
public Interface paymentservice {

    public void payment (id, pass, amount)
    {
    }

}
```

② Creating new payment classes and extending payment service without modifying it.

```
public class bkash implements paymentservice {

    public void payment ( id, amount, pass )
        {
        // proced bkash service
        }
    }

public class card implements paymentservice
    {
    public void payment (id, amount, pass)
        {
        // proceed card
        }

public class nagad implements paymentservice
    {
    public void payment (id, amount, pass)
        {}
        }
```

LSP (Liskov substitution Principle)

Principle states " The object of superclass should be neplaceable with the objects of it subclass without breaking the application "

Derived / child classes must be substituable for base/panent class

ensune inheritance used properply

what Do you mean?

object of subclass should behave the same way as the object of supercless

NOTE → If an ovennide method does nothing on just thnow exception then it probably violating Liskov

Parent class → Public class Bird {

has two methods

```
public void flying()
{ // flying }
public void walking()
{ // walking // }
}
```

Subclass →

Dove is a Chid class of Bird

As dove can both fly and walk. Dove is a complete sobstitute of Bird class

```
public class Dove extends Bird{
{
    public void flying()
    { // Dove fly }

    public void walking()
    { // Dove walk }
.
}
```

∴ doesn't bneak LSP

subclass

penguine subclass of bird. But penguine can't fly

So it is not complete substitution of Bird class

Breaks LSP

```
public class penguine
        extends Bird {

public void flying()
  { throws exception }

                walking()
public void penguine
  { // penguine walk }

}
```

Soluting the whole code

(1)

```
Public class Bind {
public void walking( )
{ // walking }
}
```

(2)

```
public class Flying bind
extend Bind {
public void flying( )
{ // flying }
}
```

(3)

```
public class
Dove extends flyingBind {
// can fly and walk methods
}
```

(4)

```
public class penguine extends Bind {
// can walk , no fly
}
```

LSP

① ensunes subtypes can be placed in their base type without breaking the application

② focus on inhenitance is used connectly

③ concenned with inhenitance & polymorphism

ISP

① ensune that interface are designed in a specific and relevant way to the needs of client

② focus on avoiding unnecessary interface implementation

③ concenned with interface and their usages

Isp (Interface segregation principle)

Principle states that client should not be forceed to implement methods of an interface, they don't use

Social media interface

```
public interface socialmedia {

    public void chatwithfriend();
    public void post();
    public void sendphoto();

}
```

Facebook (as client) using the functionalites of social media

```
public class Facebook implements SocialMedia {

    public void chatwithfriend ( )
        { // chat FB // }

    public void post ( )
        { // posting // }

    public void sendphoto ( )
        { // sending // }
}
```

Here Facebook as client has all feature of this.

Not violating ISP

whatsapp using the functionalties

```
public class whatsapp implements SocialMedia {

    public void chatwithFriend ( )
    {          }

    public void post ( )
    {  cannot post in whatsapp      }

    public void sendphoto ( )
    {          }

}
```

As whatsapp don't have posting feature. Interface is forcing whatsapp to implement post() method

Viaolating ISP

Solution

① `public interface SocialMedia {`
`public void chatwithfriend();`
`public void sendphoto();`
`}`

→ now it contains methods only both facebook and whatsapp can use

② `public interface PostManagen {`
`public void post();`
`}`

→ new interface for post method

③ `public class Facebook implements SocialMedia, PostManagen {`
`}`

→ facebook use both interfaces

④ `public class whatsapp implements SocialMedia {`
`}`

→ whatsapp only use socialmedia cause it can't post

DIP

Dependency Inversion principle

Go to page-3

Go to page-4

Dependency Inversion Principle (DIP)

P-3

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module, but both should depend on the abstraction.

Let assume you have ShoppingMall class and it only takes debit card payment

```
public class DebitCard{
public void doTransaction(int amount){
        System.out.println("Done with DebitCard");
    }
}
```

```
public class ShoppingMall {

    private DebitCard debitCard;

    public ShoppingMall(DebitCard debitCard) {
        this.debitCard = debitCard;
    }

    public void doPayment(Object order, int amount){
        debitCard.doTransaction(amount);
    }

    public static void main(String[] args) {
        DebitCard debitCard=new DebitCard();
        ShoppingMall shoppingMall=new ShoppingMall(debitCard);
        shoppingMall.doPayment("some order",5000);
    }
}
```

Here, ShoppingMall class is dependent on DebitCard. Now, the shopping mall wants to introduce CreditCard payment. As ShoppingMall class is tightly coupled with DebitCard, you cannot apply credit card payment in ShoppingMall.

To simplify this designing principle, i am creating a interface called **Bankcards**

```
public interface BankCard {
  public void doTransaction(int amount);
}
```

Now both **DebitCard** and **CreditCard** will use this **BankCard** as abstraction.

```
public class DebitCard implements BankCard{

    public void doTransaction(int amount){
    System.out.println("Done with DebitCard");
    }
}
```

```
public class CreditCard implements BankCard{

    public void doTransaction(int amount){
    System.out.println("Done with CreditCard");
    }
}
```

Now you need to redesign **ShoppingMall** implementation

```
public class ShoppingMall {

    private BankCard bankCard;

    public ShoppingMall(BankCard bankCard) {
        this.bankCard = bankCard;
    }

    public void doPayment(Object order, int amount){
        bankCard.doTransaction(amount);
    }

    public static void main(String[] args) {
        BankCard bankCard=new CreditCard();
        ShoppingMall shoppingMall1=new ShoppingMall(bankCard);
        shoppingMall1.doPayment("do some order", 10000);
    }
}
```

> *ShoppingMall class (high-level module) was dependent on DebitCard (low-level module). As it violated the DIP, we created BankCard interface and thus lessened the dependency on DebitCard.*
> *Now ShoppingMall class depends on BankCard which can implement several low-level modules and thus doesn't violate the DIP.*

Factory
Design pattern

Intent :- " Provide an interface for creating objects in a superclass but allow the subclass to alter the type of object that will be created "

→ define an interface or abstract class for creating object

→ Let the subclass decide which class to instantiate

Motivation the motivation of factory design pattern?

→ motivated by the need of creating object without specifying the exact class of object that will be instantiated.

→ The goal is to encapsulate object creation and provide a level of abstraction.

1. Decoupling object creation
2. Abstraction and encapsulation
3. Simplifying object creation logic
4. Flexibility

Applicability

when and where to use factory method?

① Use factory method when dont know since before the exact type and dependencies of the object your code should work with.

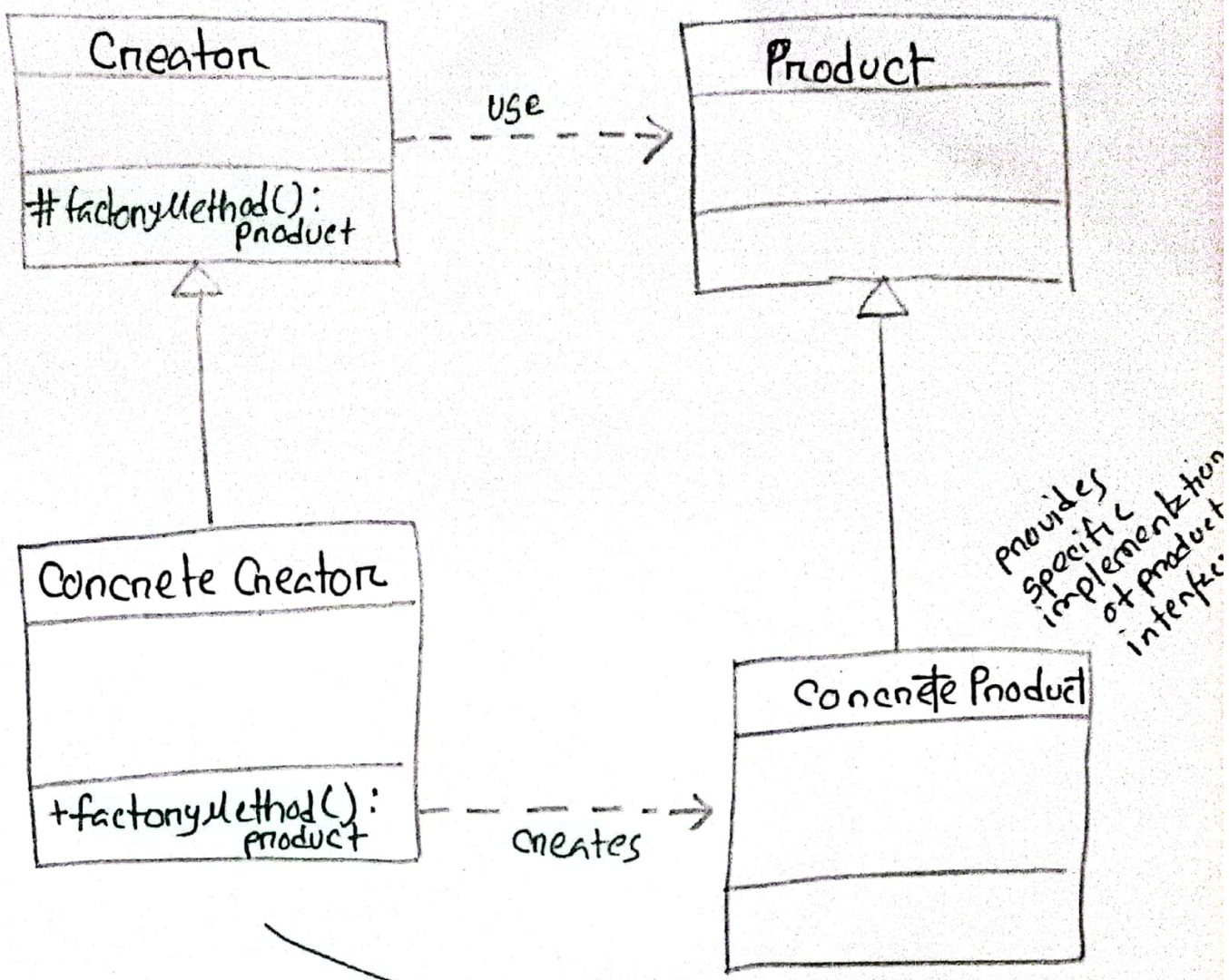
(factory method separates product construction code that actually uses the product)

② Use when you want to provide users of your library / framework with a way to extend its internal components

③ use factory method when you want to save system resources be reusing existing object instead of building them each time

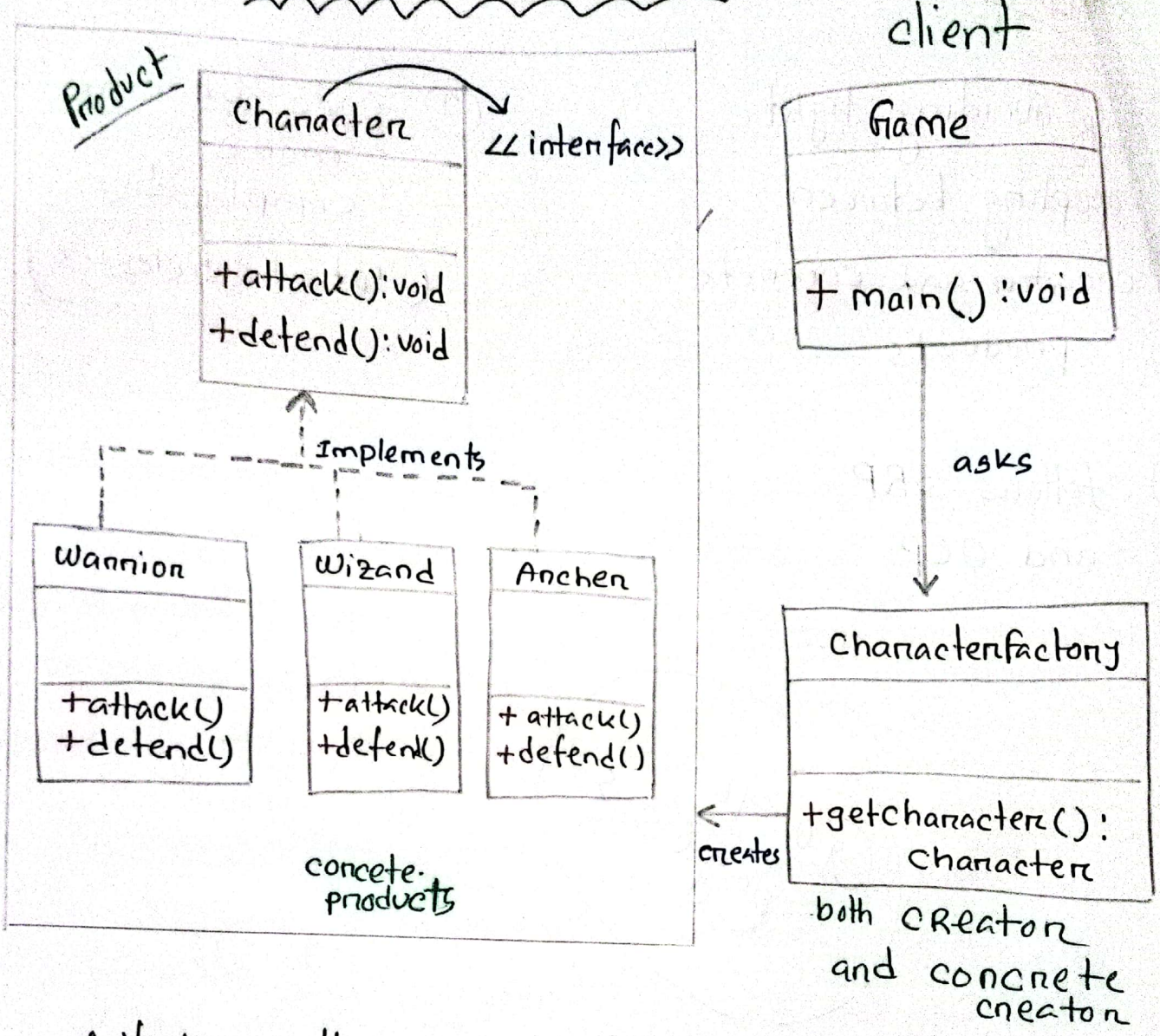
declare factorymethod()
for creating products

(common interface
for the objects
created by factory)



(override the
factory methods
to create specific
concrete product
objects)

UML case Diagram



what are the participant object of factory design pattern?

1. Game
2. characterfactor
3. character — Warrior, Wizard, Archer

Pros

1. help avoiding tight coupling between creator and concrete products
2. follows SRP and OCP

Cons

1. code become more complicated (lot of subclasses)

Go to page-5

First, we'll define an interface called `Character` that represents a generic character in the game:

```
public interface Character {
    void attack();
    void defend();
}
```

Next, we'll create concrete classes that implement the `Character` interface. Here's an example of a `Warrior` class:

```
public class Warrior implements Character {
    @Override
    public void attack() {
        System.out.println("Warrior attacks with a sword!");
    }

    @Override
    public void defend() {
        System.out.println("Warrior defends with a shield!");
    }
}
```

Similarly, we'll create `Wizard` and `Archer` classes that implement the `Character` interface.

```
public class Wizard implements Character {
    @Override
    public void attack() {
        System.out.println("Wizard attacks with magic!");
    }

    @Override
    public void defend() {
        System.out.println("Wizard defends with a spell!");
    }
}
```

```
public class Archer implements Character {
    @Override
    public void attack() {
        System.out.println("Archer attacks with a bow!");
    }

    @Override
    public void defend() {
        System.out.println("Archer defends with a dodge!");
    }
}
```

```
public class CharacterFactoryConcrete extends CharacterFactory {
    @Override
    public Character getcharcter(int level) {
        if (level == 1) {
            return new Warrior();
        }
        else if(level == 2) {
            return new Wizard();
        }
        else {
            return new Archer();
        }
    }
}
```

```
public abstract class CharacterFactory {
    public abstract Character getcharcter(int level);
}
```

```
public class Game {

    public static void main(String[] args) {
        CharacterFactory factory = new CharacterFactoryConcrete();
        Character character = factory.getcharcter(2);
        character.attack();
    }
}
```

```
public class Game {

    public static void main(String[] args) {
        Character character = new Wizard();
        // Revealing the character class to Game Client
        // Break Factory Design Pattern
        character.attack();
    }
}
```

```
public class Game {
    public static void main(String[] args) {
        // Create a Warrior character
        Character warrior = new Warrior();
        warrior.attack();
        warrior.defend();

        // Create a Wizard character
        Character wizard = new Wizard();
        wizard.attack();
        wizard.defend();

        // Create an Archer character
        Character archer = new Archer();
        archer.attack();
        archer.defend();

        // do something with the characters ...
    }
}
```

Factory Design

(1) It aims to encapsulate object creation by providing separate factory class responsible for creating an object of particular type.

(2) It provides a way to create object without specifying their concrete class

(3) consists of a factory interface or abstract class that declare a factory method for creating object

Abstract factory

(1) It aims to provide to provide an interface for creating families of related or dependent object without specifying concrete classes

(2) It abstracts the creation of multiple related objects ensuring created objects are compatible with each other

(3) consists of an abstract interface or class that declare factory method for creating families of Related object

④ Factory method is useful when there is a need to decouple the client code from specific classes being instantiated

⑤ is responsible for creating products belong to one family

⑥ You have a factory method that creates objects that derive from particular base class

Abstract

④ Factory method is useful when there is a need to create families of objects that are designed to work together

⑤ multiple families of products

⑥ You have a factory that creates other factories and these factories in turn create objects derived form base classes.

Abstract factory Method

Intent :- Abstract factory pattern let you produce families of related objects without specifying their concrete classes.

Pros

→ You can be sure that product you are getting from factory are compatiable

→ avoid tight coupling between concrete product and client code

→ Follow OCP

Cons

→ complex code structure (so many interface & classes)

Motivation

The motivation behind it is to provide an abstract an interface for creating families of related or dependent objects without specifying concrete class.

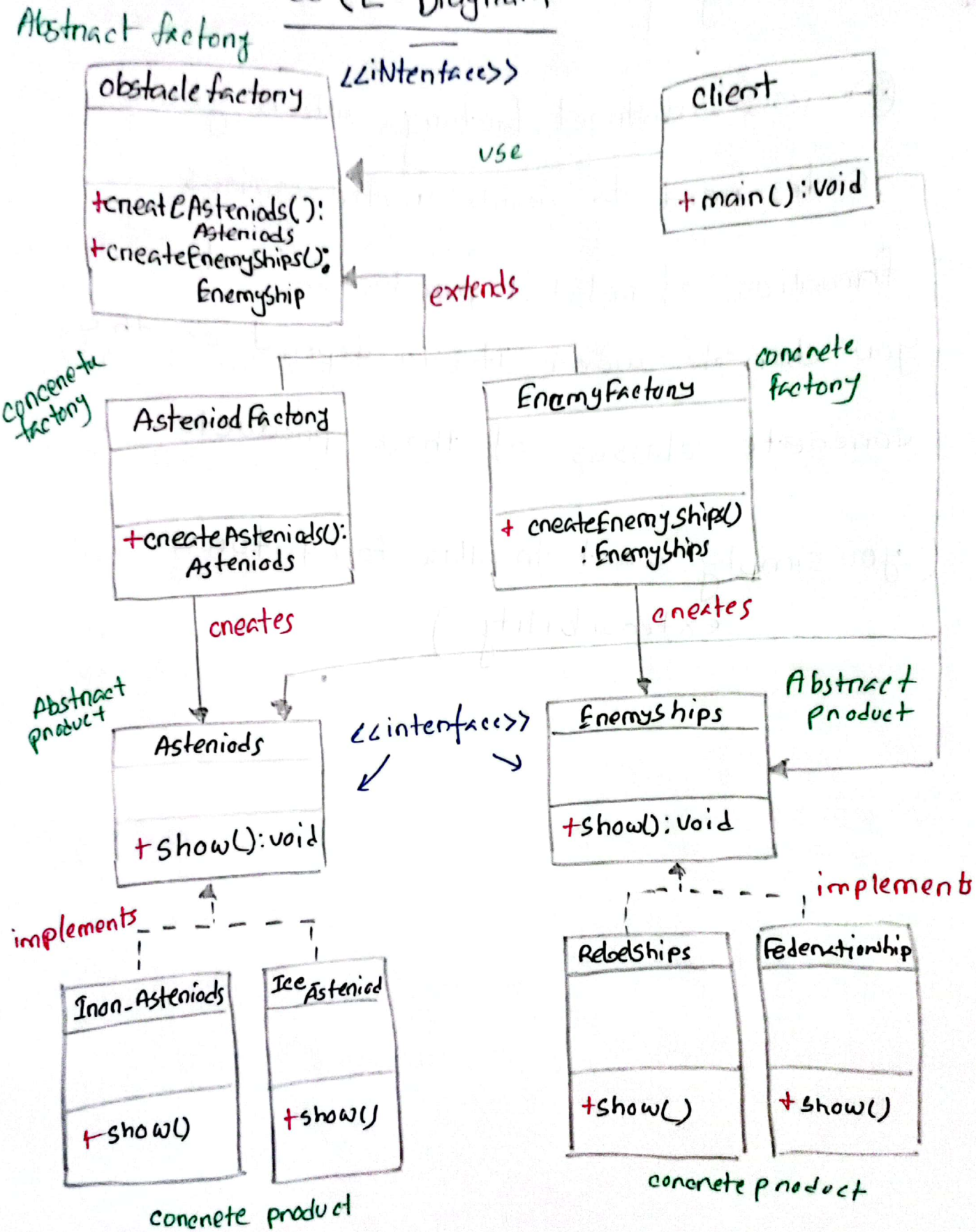
1. Encapsulation of object creation
2. Creation of families of ~~related~~ related objects
3. flexibility and interchangeability
4. Adherence to the open-closed principle

Applicability

① use Abstract factory when your codes need to work with various families of related products, but you don't want it to depend on the concrete classes of those products.

(you simply want to allow for future extensibility)

UML Diagram


Abstract factory
obstacle factory
<<interface>>
+creatEAsteniods(): Asteniods
+createEnemyShips(): Enemyship
client
+main():void
use
extends
concenete factory
Asteniod Factory
+createAsteniods(): Asteniods
concrete factory
EnemyFactory
+ createEnemyShips() : Enemyships
creates
creates
Abstract product
Asteniods
+show():void
<<interface>>
Enemyships
+show():void
Abstract product
implements
implements
Inon-Asteniods
+show()
Ice Asteniod
+show()
concrete product
Rebelships
+show()
Federationship
+show()
concrete product

Abstract Product

(1)

```
public interface Asteriods {
    void show();
}
```

(4)

```
public interface EnemyShips {
    void show();
}
```

Participants:

1. **AbstractFactory:** Declares an interface for operations that create abstract product objects.
2. **ConcreteFactory:** Implements the operations to create concrete product objects.
3. **AbstractProduct:** Declares an interface for a type of product object.
4. **ConcreteProduct:** Define a product object to be created by the corresponding concrete factory. Inplements the AbstractProduct interface.
5. **Client:** Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Concrete Product

Product A

(2)

```
public class Ice_Asteroids implements Asteriods {

    @Override
    public void show() {
        System.out.println("Ice_Asteroids popped up");
    }
}
```

(3)

```
public class Iron_Asteroids implements Asteriods {

    @Override
    public void show() {
        System.out.println("Iron_Asteroids popped up");
    }
}
```

Product B

(5)

```
Auto (TypeScript) ~
public class FederationShips implements EnemyShips {
    @Override
    public void show() {
        System.out.println("Federation Ships Appeared");
    }
}
```

(6)

```
public class RebelShips implements EnemyShips {
    @Override
    public void show() {
        System.out.println("Rebel Ships Appeared");
    }
}
```

(7) ## Abstract Factory

```
public abstract class ObstacleFactory {
    public abstract Asteriods createAsteriods(int score);
    public abstract EnemyShips createEnemyShips(int score);
}
```

Concrete Factory

(8)

```
public class AsteroidFactory extends ObstacleFactory {

    @Override
    public Asteriods createAsteriods(int score) {
        if(score > 500) return new Ice_Asteroids();
        else return  new Iron_Asteroids();
    }

    @Override
    public EnemyShips createEnemyShips(int score) {
        return null;
    }
```

(9)

```
public class EnemyFactory extends ObstacleFactory {
    @Override
    public Asteriods createAsteriods(int score) {
        return null;
    }

    @Override
    public EnemyShips createEnemyShips(int score) {
        if(score > 500) return new FederationShips();
        else return  new RebelShips();
    }
}
```

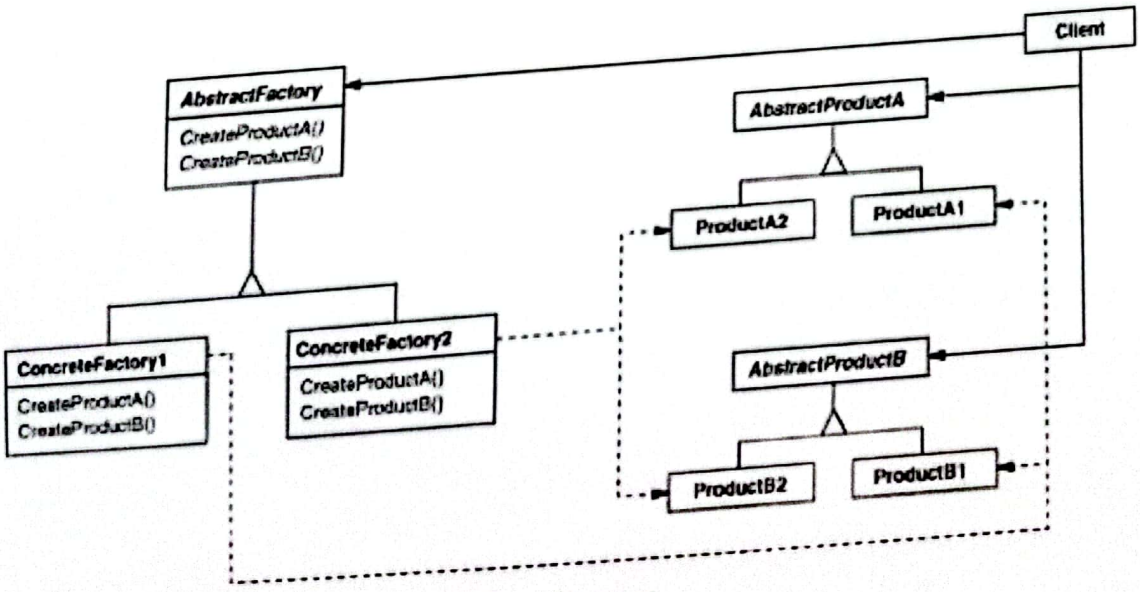
Client

(10)

```
public class Client {
    public static void main(String[] args) {

        ObstacleFactory factory ;
        int Score = (int)(Math.random() * 0.5) + 2;
        int level = (int)(Math.random() * 2) + 1;

        if (level == 1 ) {
            factory = new AsteroidFactory();
            Asteriods obstacle = factory.createAsteriods(Score);
            obstacle.show();
        }
        else {
            factory = new EnemyFactory();
            EnemyShips obstacle  = factory.createEnemyShips(Score);
            obstacle.show();
        }
    }
}
```



Class Diagram

Builder Method Design

Intent

Builder method ; Lets you construct complex ~~code~~ object step by step

The pattern allows you to produce different types and representations of an object using same construction code

Pros

- can construct object step by step
- defer construction step in recursisly
- Use the same construction code building various product

Cons

- complexity of code increases.

Motivation motivation behind builder pattern is to provide a flexible and step by Step approach to construct complex object

→ it aims to separate the construction of an object from its nepresentation, allowing the same construction process to create diffenent nepresentation of object.

1. Encapsulation of construction process
2. step by step construction
3. Creation of complex objects
4. Vaniation in object creation
5. separation of concenns

Applicability

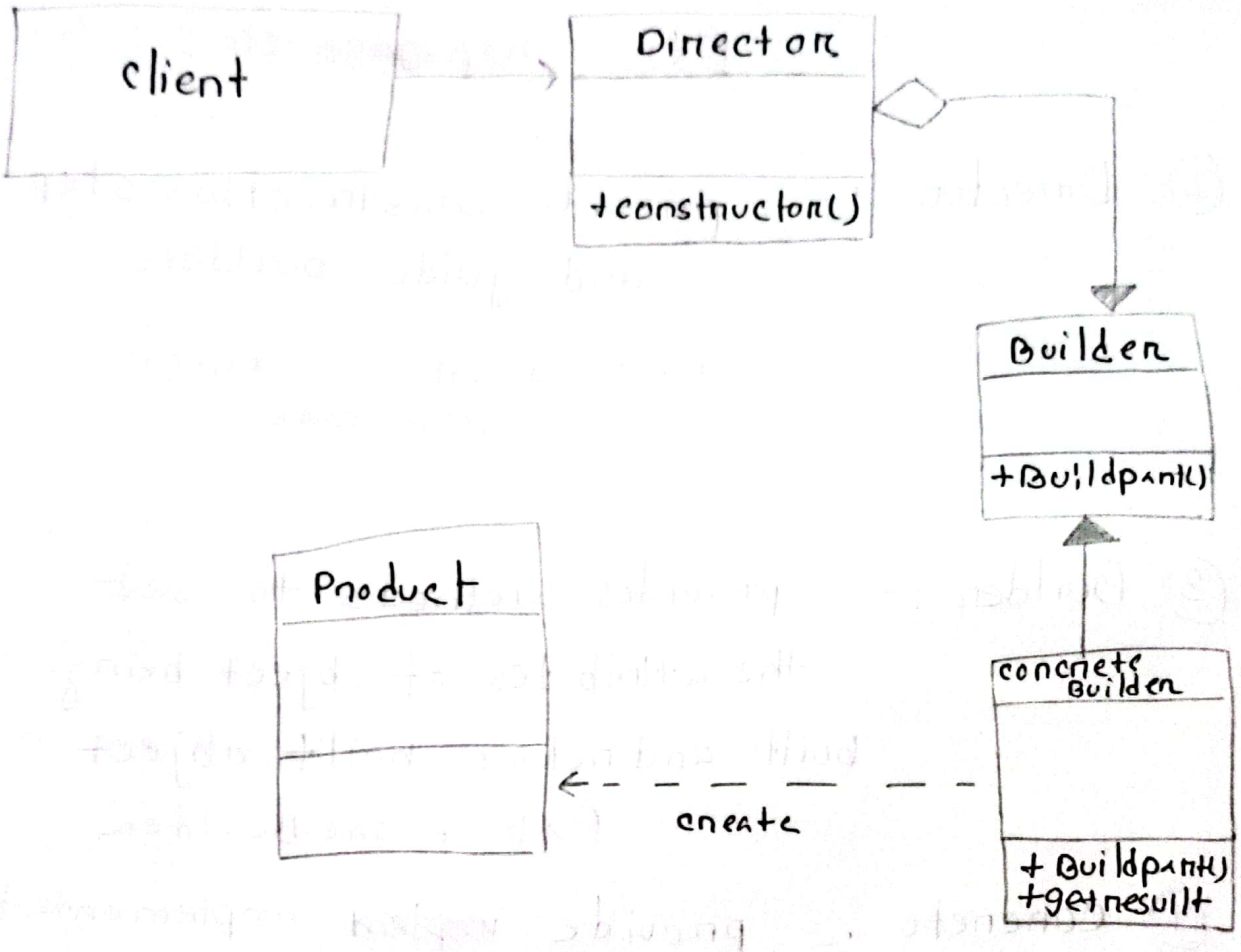
(1) Use builden method to get ride of telescoping constructor

```
class pizza {
Pizza (int size) {...}
Pizza ( int size, boolen cheese)
        {...}
...
}
```

(2) use builden method when you want your code able create diffenent representation of products with same construction code

(3) Use Builden method to construct composite tree or complex object

UML Diagram


client
Director
+construction()
Builder
+BuildPart()
Product
concrete Builder
+BuildPart()
+getResult
create

① Client :- Client interact with Builder to construct desire object

Ex: shop class

② Director :- provide construction step and guide builder

Ex :- do not have director for the code

③ Builder :- provides methods to set the attributes of object being built and return built object

Ex:- Phone Builder

④ Concrete Builder :- provide implementation of builder interface providing sepific implementation for setting attributes.

Ex:- Phone Builder

⑤ Product :- represent final object being built

Ex: Phone

Problem - blue

P-10

```
public class Phone {
    private String os;
    private int ram;
    private String processor;
    private double screenSize;
    private int battery;

    public Phone(String os,int ram,
                 String processor,double screenSize,int battery) {

        super();
        this.os = os;
        this.ram = ram;
        this.processor = processor;
        this.screenSize = screenSize ;
        this.battery = battery ;
    }

    public String toString() {
        return "Phone Specification : \n" +
                "OS = " + os + "\n" +
                "Ram = " + ram + "\n" +
                "Processor = " + processor + "\n" +
                "Screen Size = " + screenSize + "\n" +
                "Battery = " + battery + "\n" ;
    }
}
```

```
public class Shop {
    public static void main(String[] args) {
        Phone p = new Phone("Android",8,"Qualcomm 882",6.7,5000);
        System.out.println(p);
    }
}
```

Problem of this structure: (Problem)

While creating phone object I have to remember all the parameters & their sequence and must need to pass the parameters of phone in sequentially

I cannot pass parameter in any random order.

I need to pass every parameter for creating an object. I cannot create a phone object just passing any single/multiple parameters. (Example: If I just pass os parameter I cannot build the phone object)

If I don't want to send all parameter, system will not let me create a phone object.

```
public class PhoneBuilder {

    private String os;
    private int ram;
    private String processor;
    private double screenSize;
    private int battery;

    public PhoneBuilder setOs(String os){
        this.os = os;
        return this;
    }
    public PhoneBuilder setRam(int ram){
        this.ram = ram;
        return this;
    }
    public PhoneBuilder setProcessor(String processor){
        this.processor = processor;
        return this;
    }
    public PhoneBuilder setScreenSize(int screenSize){
        this.screenSize = screenSize;
        return this;
    }
    public PhoneBuilder setBattery(int battery){
        this.battery = battery;
        return this;
    }

    public Phone getPhone(){
        return new Phone(os,ram,processor,screenSize,battery);
    }
}
```

```
public class Shop {
    public static void main(String[] args) {

        Phone p = new PhoneBuilder().setOs("Android")
                .setBattery(4000).setRam(16).getPhone();

        System.out.println(p);
    }
}
```

1. Sequence Doesn't Matter
2. No Need to pass all parameter value
3. Flexible step by step building

Solution - Green

Prototype Design Pattern

Intent

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Pros

1) Clone objects without coupling to concrete classes
2) Produce complex objects more conveniently.

Cons

1) Cloning complex objects that have circular references might be tricky.

Motivation

The motivation behind the Prototype Design Pattern is to create new objects by cloning or copying existing objects.

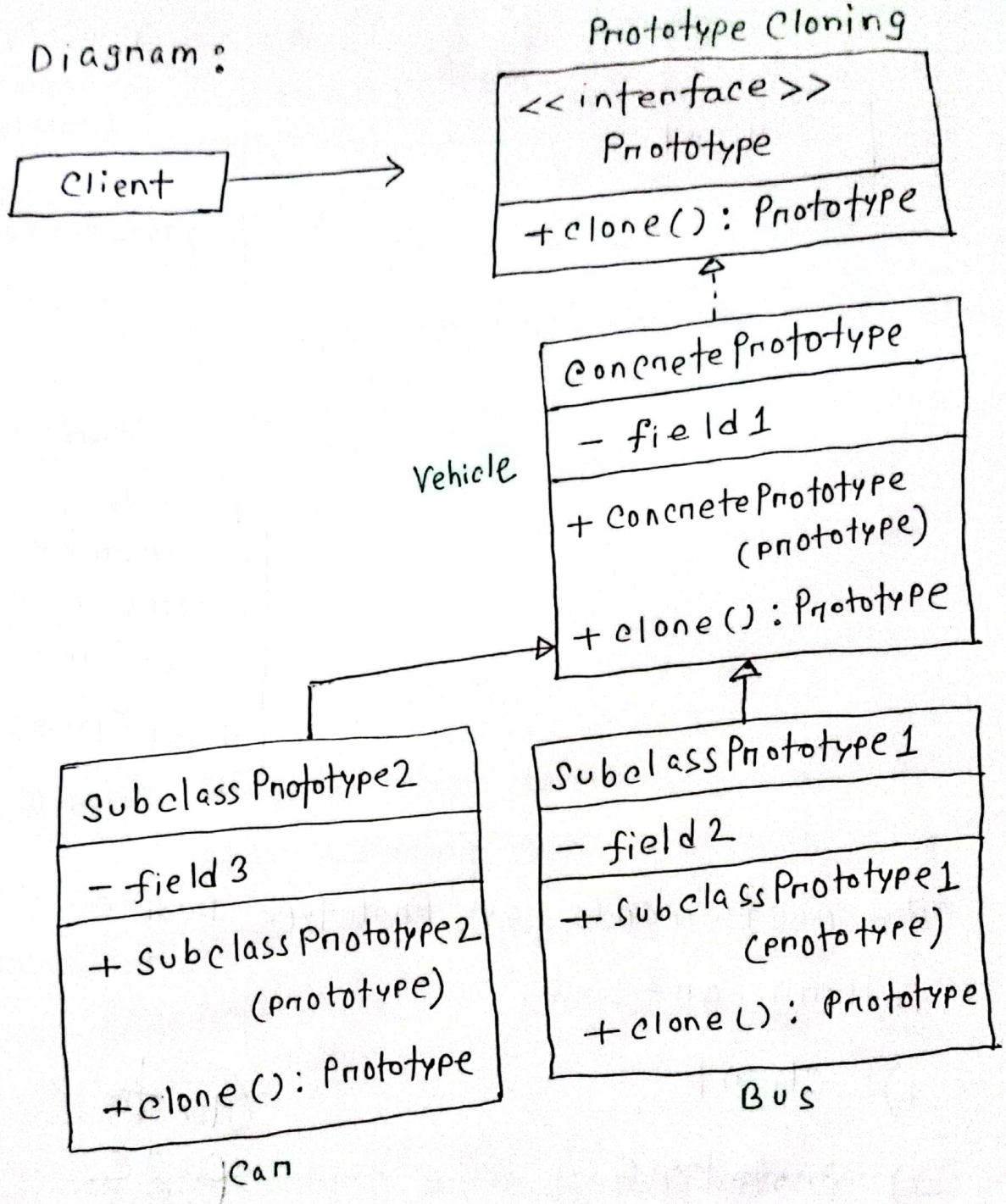
1) Object creation flexibility
2) Reducing object creation overhead
3) Managing object variations
4) Preserving object state
5) Hiding object creation details

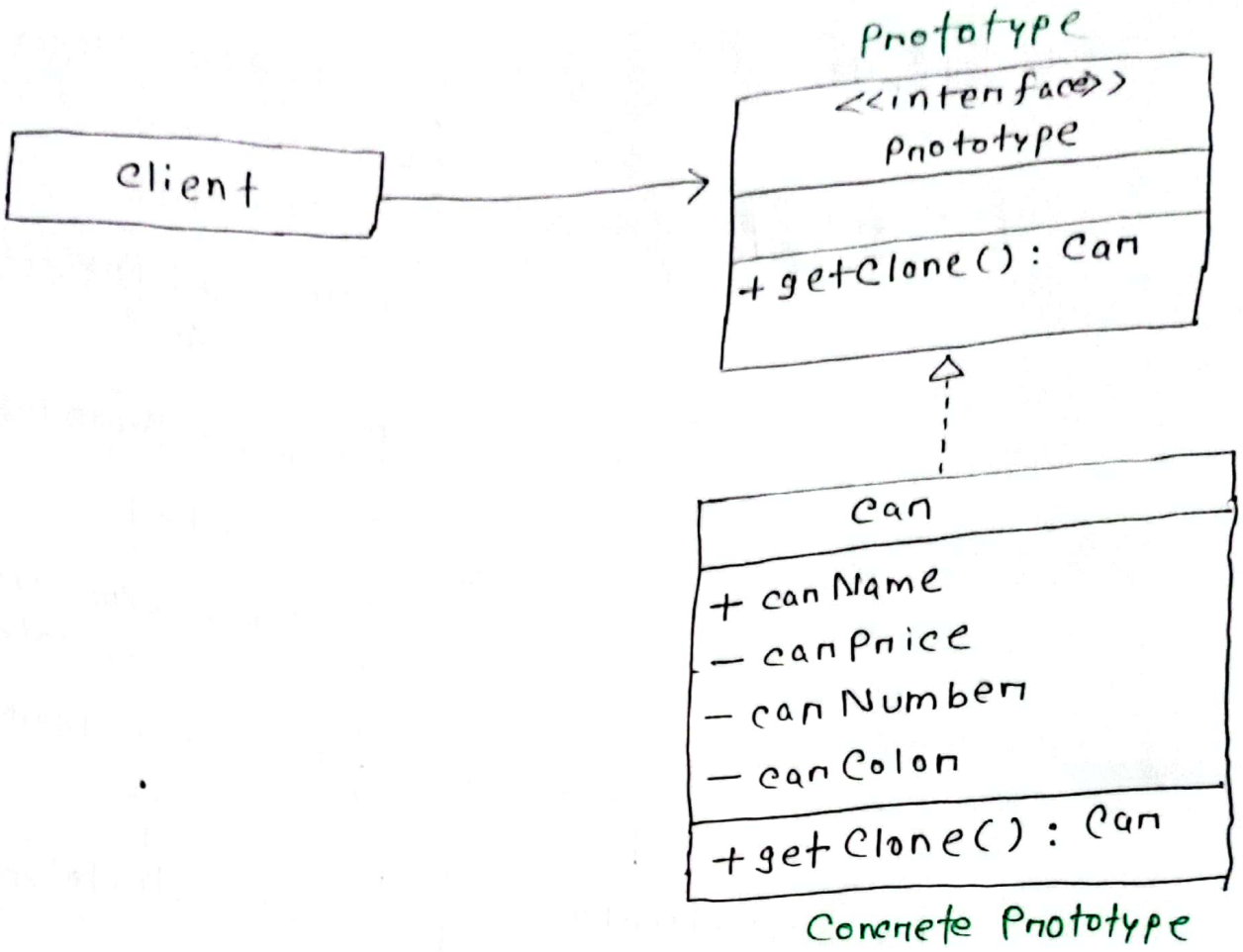
Applicability

1) Use the prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

2) Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

UML Diagram:


Prototype Cloning
Client
<<interface>>
Prototype
+clone(): Prototype
Vehicle
ConcretePrototype
- field1
+ ConcretePrototype (prototype)
+clone(): Prototype
Subclass Prototype2
- field3
+ Subclass Prototype2 (prototype)
+clone(): Prototype
Can
Subclass Prototype1
- field2
+ Subclass Prototype1 (prototype)
+clone(): Prototype
Bus



The participants of Prototype Design Pattern are:

1) Client
2) Prototype
3) Concrete Prototype

Go to page (11-12)

Problem (1)

```
public class Car {
    public String carName;
    private int carPrice;
    private int carNumber;
    private String carColor;

    public Car(){};
    public Car(String carName,int carPrice,int carNumber,String carColor) {
        this.carName = carName;
        this.carPrice = carPrice;
        this.carNumber = carNumber;
        this.carColor = carColor;
    }

    public void draw() {
        System.out.println("Car Specification" +
                           "Car Name : " + carName +
                           "Car Price : " + carPrice +
                           "Car Number : " + carNumber +
                           "Car Color : " + carColor
                               );
    }
}
```

(2)

```
public class Client {
    public static void main(String[] args) {
        Car car1 = new Car("Ferrari",20000000,2045,"Red");

//        Trying to copy car2 into car1
//        Car car2 = new Car();

//        car2.carName = car1.carName;  --> allows private memeber copy
//        car2.carColor = car1.carColor; --> Private Member can't be copied
//        car2.carNumber = car1.carNumber; --> Private Member can't be copied
//        car2.carPrice = car1.carPrice; --> Private Member can't be copied

        /*
            Problem 1 - you have to copy element one by one
            Problem 2 - Cannot copy private member
            Problem 3 - in Future if you add any member to car1
                        you also have to manually copy member car1 to car2
            Problem 4 - Copying is tightly coupled and depends on car class
            Problem 5 - Sometimes we may not have access to the class
                        In that case we can not copy a class object
        */
    }
```

Solution

p-12

(1)

```
public interface Prototype {
    Car getClone();
}
```

(2)

```
public class Car implements Prototype {
    public String carName;
    private int carPrice;
    private int carNumber;
    private String carColor;

    public Car(){}
    public Car(String carName, int carPrice, int carNumber, String carColor)
        this.carName = carName;
        this.carPrice = carPrice;
        this.carNumber = carNumber;
        this.carColor = carColor;
    }

//    public Car(Car car) {
//        if(car != null){
//            this.carName = car.carName;
//            this.carPrice =  car.carPrice;
//            this.carNumber =  car.carNumber;
//            this.carColor =  car.carColor;
//        }
//    }
//
//    public Car CLONE()
//    {
//        return new Car(this);
//    }

    public void draw() {
        System.out.println("Car Specification" +
                "Car Name : " + carName +  "\n" +
                "Car Price : " + carPrice +  "\n" +
                "Car Number : " + carNumber +  "\n" +
                "Car Color : " + carColor

        );
    }

    @Override
    public Car getClone() {
        return new Car(carName,carPrice,carNumber,carColor);
    }
}
```

(3)

```
public class Client {
    public static void main(String[] args) {
        Car car1 = new Car("Ferrari",20000000,2045,"Red");

        // Copying Object having private value
        Car car2 = car1.getClone();
        car2.draw();
    }
}
```