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

```
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());
    }
}
```

☹ Problem

```
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.
     private static Database instance;
     private String username;

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

     // Provide a public static method to get the instance of the class
     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
```

> *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".*

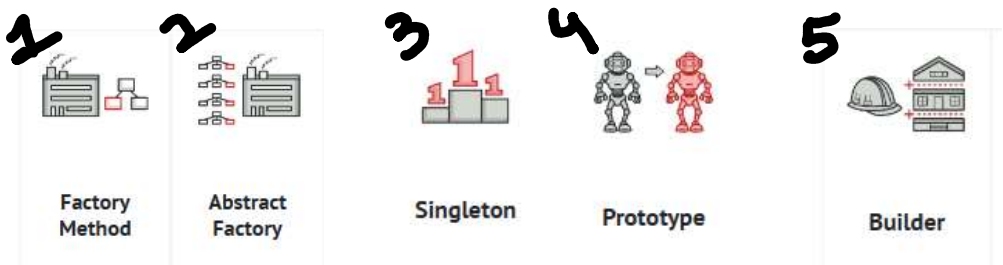
Creational patterns

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.**



Dependency Inversion Principle (DIP)

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.*

> *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!");
    }
}
```

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

> *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!");
    }
}
```

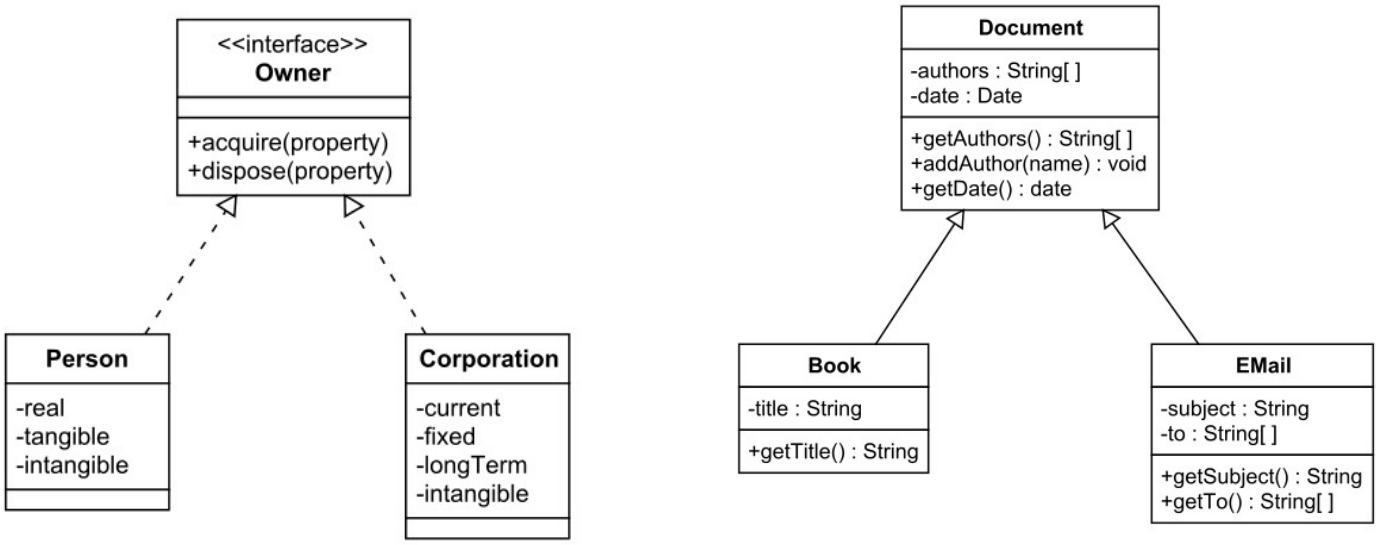
```
public class CharacterFactoryConcrete extends CharacterFactory {
    @Override
    public Character getcharcter(int level) {
        if (level == 1) {
            return new Warrior();
        }
        else if(level == 2) {
            return new Wizard();
        }
        else {
            return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

```
public 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 ...
    }
}
```

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

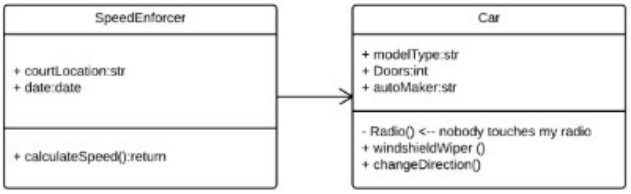


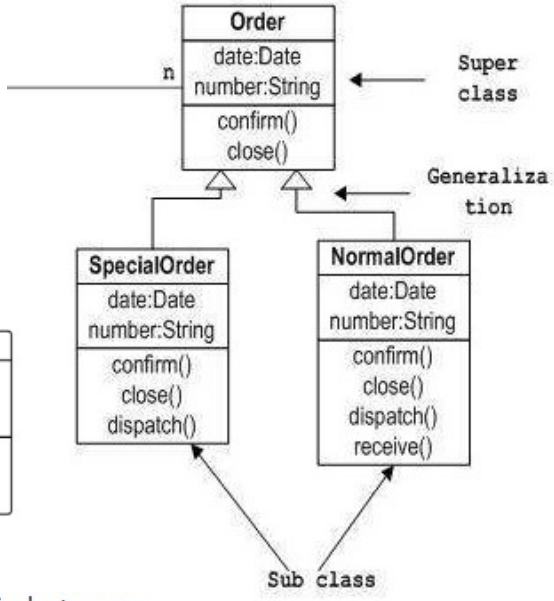
A class implements an interface.

Member access modifiers

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

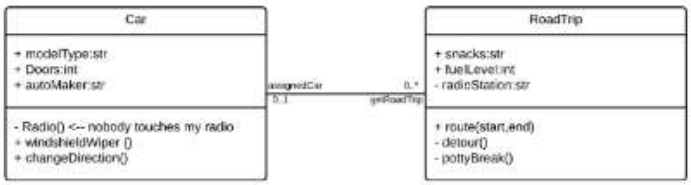
- Public (+)
- Private (-)
- Protected (#)
- Package (~)
- Derived (/)
- Static (underlined)

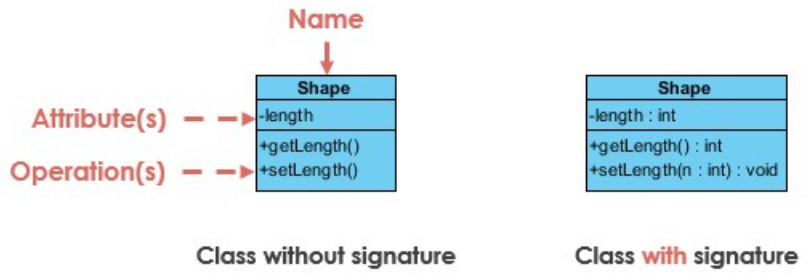




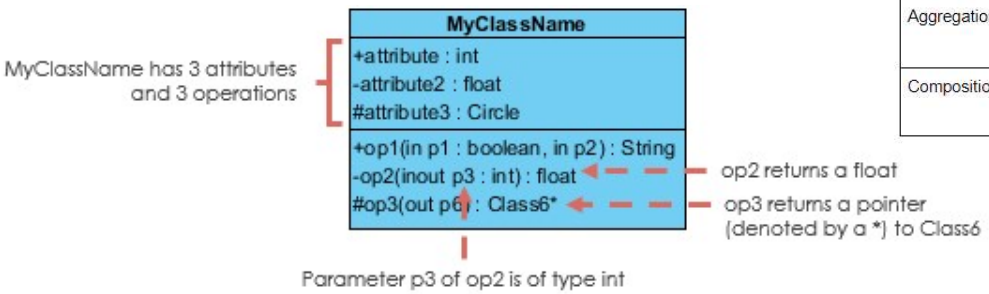
Unidirectional association: A slightly less common relationship between two classes. One class is aware of the other and interacts with it. Unidirectional association is modeled with a straight connecting line that points an open arrowhead from the knowing class to the known class.

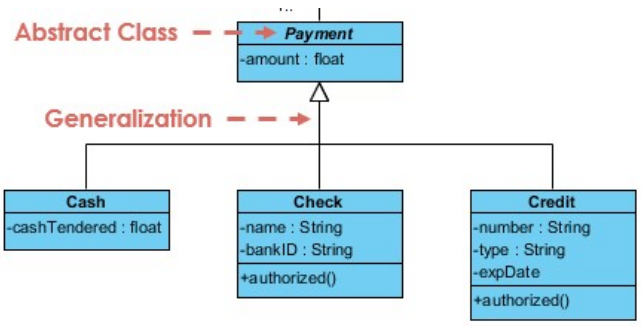
- **Bidirectional association:** The default relationship between two classes. Both classes are aware of each other and their relationship with the other. This association is represented by a straight line between two classes.

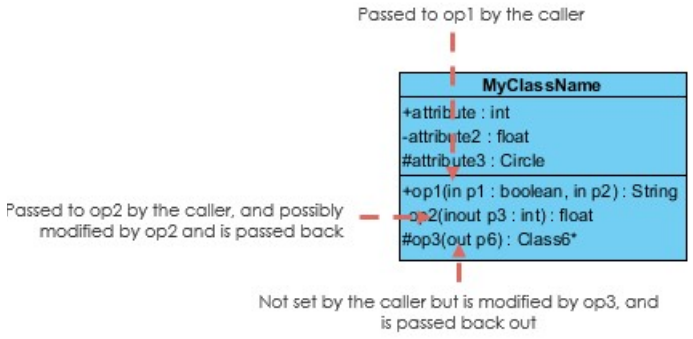


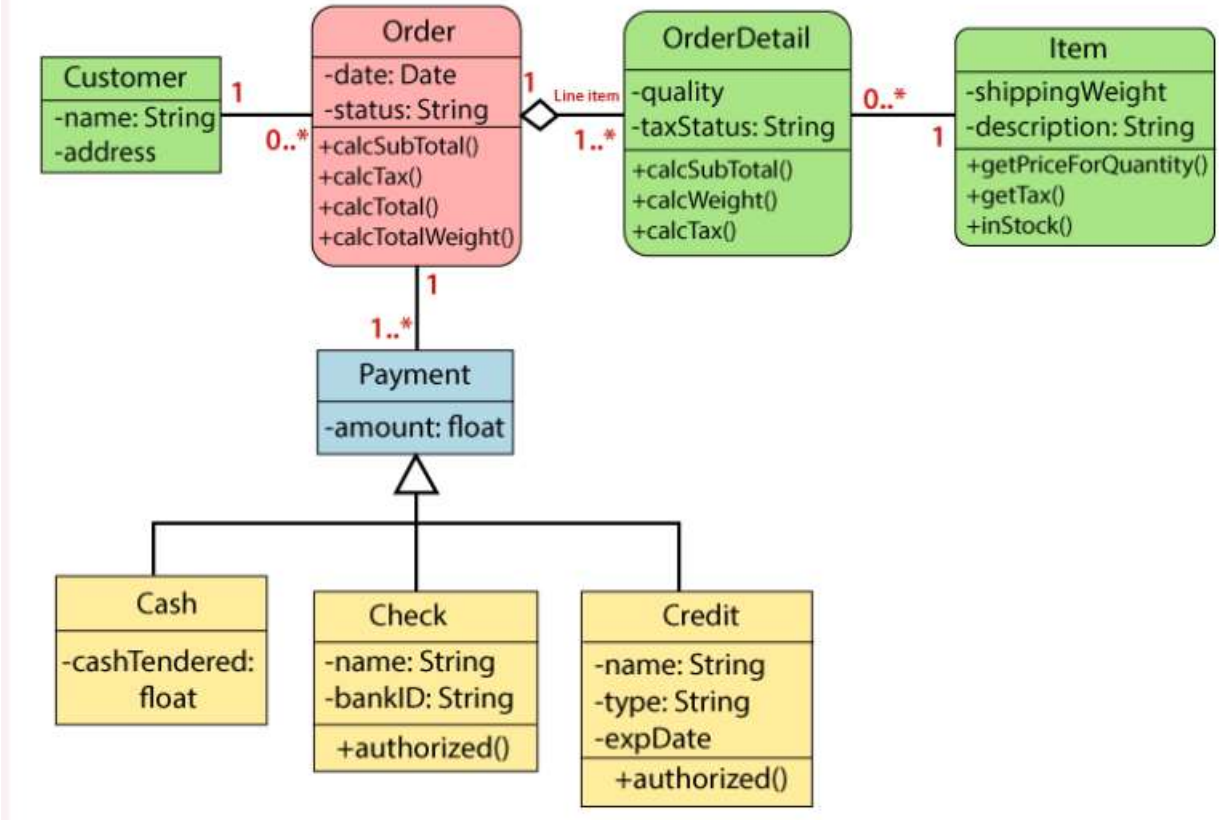


Class Diagram Relationship Type	Notation
Association	
Inheritance	
Realization/ Implementation	
Dependency	
Aggregation	
Composition	









Abstract Product

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

```
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

```
public class Ice_Asteroids implements Asteriods {

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

```
public class Iron_Asteroids implements Asteriods {

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

Product B

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

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

Abstract Factory

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

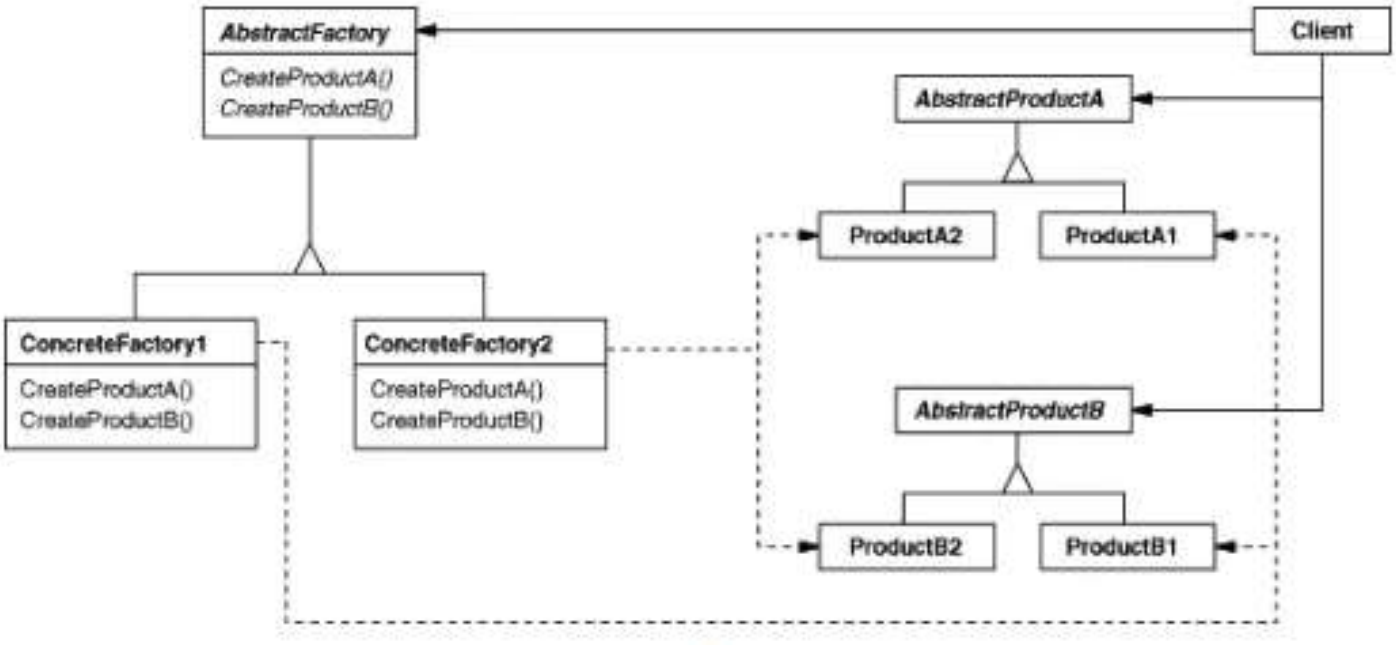
Concrete Factory

```
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;
    }
}
```

```
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

```
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

```java
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" ;
    }
}
```

```java
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:

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.

```java
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);
    }
}
```

```java
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

```java
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
                            );
    }
}
```

```java
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
        */
    }
```

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

```java
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);
    }
}
```

```java
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();
    }
}
```

Explaining the following code:

1. The `Client` class creates an instance of the `JSON` class, which represents some JSON data.
2. It also creates an instance of the `JsonToXmlAdapter` class, which implements the `IAdapter` interface.
3. The `convert` method in the `JsonToXmlAdapter` class takes a `JSON` object as a parameter and converts it to an `XML` object.
4. The client code calls the `convert` method on the `iAdapter` object, passing the `json` object as the argument: `XML xml = iAdapter.convert(json);`
5. Inside the `JsonToXmlAdapter` class, the `convert` method receives the `JSON` object and calls the `convertToXML` method on it.
6. The `convertToXML` method in the `JSON` class performs the logic to convert the JSON data to XML format and returns an `XML` object representing the converted data.
7. The `convert` method in the `JsonToXmlAdapter` class receives the `XML` object from the `convertToXML` method and returns it.
8. The `XML` object returned from the `convert` method of the adapter is assigned to the `xml` variable in the client code: `XML xml = iAdapter.convert(json);`

```
public class XML {
    public XML(){}
    public XML(String data){}
}
```

```
public class JSON {

    public JSON(){};
    public JSON(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified JSON data");
    }
}
```

```
public interface IAdapter {
    XML convert(JSON json);
}
```

```
public class JsonToXmlAdapter implements IAdapter {
    private JSON json;

    public JsonToXmlAdapter(JSON json){
        this.json = json;
    }

    @Override
    public XML convert(JSON json) {
        return this.json.convertToXML();
    }
}
```

```
public interface IAdapter<TypeA , TypeB> {
    TypeB convert(TypeA source);
}
```

```
public class PROTOBUFFER {
    public PROTOBUFFER(){};
    public PROTOBUFFER(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified PROTOBUFFER data");
    }
}
```

```
public class ProtobufferToXmlAdapter implements IAdapter<PROTOBUFFER, XML>{
    private PROTOBUFFER Protobuffer;

    public ProtobufferToXmlAdapter(PROTOBUFFER Protobuffer){
        this.Protobuffer =  Protobuffer;
    }

    @Override
    public XML convert(PROTOBUFFER PROTOBUFFER) {
        return this. Protobuffer.convertToXML();
    }
}
```

```
public class Client {
    JSON json = new JSON("json data");
    IAdapter iAdapter = new JsonToXmlAdapter(json);
    XML xml = iAdapter.convert(json);
}
```

Participants of the following code:

1. Client: The `Client` class
2. Client Interface: The `IAdapter` interface
3. Adapter: The `JsonToXmlAdapter` class
4. Service: The `JSON` and `XML` classes

```java
public interface TravellingStrategy {
    void travelBy(String location);
}
```

```java
public class BikeTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bike");
        // System holds bike travelling operations
    }
}
```

```java
public class BusTravelling implements TravellingStrategy{
    @Override
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bus");
        // System holds bus travelling operations
    }
}
```

```java
public class BoatTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Boat");
        // System holds boat travelling operations
    }
}
```

```java
public class TravelManager {
    private TravellingStrategy travellingStrategy;

    public void setTravellingStrategy(TravellingStrategy travellingStrategy){
        this.travellingStrategy = travellingStrategy;
    }

    public void travelBy(String Location) {
        travellingStrategy.travelBy(Location);
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        // Suppose I am planning a trip
        // Sylhet - Shreemangal - MadhabpurLake - Sylhet
        // Want to travel to sylhet with bus
        // when I reach sylhet
        // I want to switch to bike to go to shreemangal
        // To travel MadhabpurLake, I switch to boat
        // Then I again switch to bike to come back to sylhet

        // Strategy pattern provide the facility
        // to interchange between object (even in runtime)

        TravelManager travelManager = new TravelManager();
        travelManager.setTravellingStrategy(new BusTravelling());
        travelManager.travelBy("Sylhet");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Shreemangal");

        travelManager.setTravellingStrategy(new BoatTravelling());
        travelManager.travelBy("MadhabpurLake");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Sylhet");

    }
}
```

Advantage :

1. Switch between algorithms in runtime
2. Interchangeable objects strategy

Suppose you have a Three step authentication system.

1. First Ipwhitlist authentication: If successful, then move to next authentication(Two Factor Authentication), Otherwise authenticationA fails
2. (If successful) Two factor authentication, if successful, then move to next authentication (Usernname Password Authentication), Otherwise authentication fails
3. (If successful) Username Password authentication ,If successful, then move whole authentication Process successful .Otherwise authentication fails

1

```java
public interface AuthenticationHandler {
    void setNextHandler(AuthenticationHandler authenticationHandler);
    boolean authenticate(String userName, String password);
}
```

2

```java
public class IPWhitelistingHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating IP whitelisting
        String clientIP = getClientIP();
        if (!clientIP.contains("192.168.192.")) {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        } else if (authenticationHandler != null) {
            System.out.println("IPWhitelistingHandler: Authentication successful.");
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        }
    }

    private String getClientIP() {
        // Simulated method to get client IP address
        return "192.168.192.";
    }
}
```

3

```java
public class TwoFactorAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;
    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating two-factor authentication
        if (userName.equals("user") && password.equals("user123") && verifyOTP("123456")) {
            System.out.println("TwoFactorAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("TwoFactorAuthenticationHandler: Authentication failed.");
            return false;
        }
    }

    private boolean verifyOTP(String otp) {
        // Simulated OTP verification logic
        return otp.equals("123456");
    }
```

4

```
public class UsernamePasswordAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        if (userName.equals("admin") && password.equals("admin123")) {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication failed.");
            return false;
        }
    }
}
```

5

```
public class Client {
    public static void main(String[] args) {
        AuthenticationHandler upHandler = new UsernamePasswordAuthenticationHandler();
        AuthenticationHandler tfaHandler = new TwoFactorAuthenticationHandler();
        AuthenticationHandler ipHandler = new IPWhitelistingHandler();

        ipHandler.setNextHandler(upHandler);
        upHandler.setNextHandler(tfaHandler);

        boolean isAuthenticated = ipHandler.authenticate("user", "user123");
        if (isAuthenticated) {
            // Proceed with server access
            System.out.println("Access granted.");
        } else {
            // Handle authentication failure
            System.out.println("Access denied.");
        }
    }
}
```

1. We start by defining the `AuthenticationHandler` interface, which represents the base handler in the chain. It has two methods: `setNextHandler()` to set the next handler in the chain and `authenticate()` to perform authentication.
2. Next, we have three concrete implementations of the `AuthenticationHandler` interface: `IPWhitelistingHandler`, `TwoFactorAuthenticationHandler`, and `UsernamePasswordAuthenticationHandler`. Each handler implements the `setNextHandler()` and `authenticate()` methods according to its specif authentication logic.
3. In the `Client` class, we create instances of the authentication handler `upHandler` for username/password authentication, `tfaHandler` for two factor authentication, and `ipHandler` for IP whitelisting.
4. We then set up the chain of responsibility by calling `setNextHandler()` on each handler, in the desired order. In this case, the request will flow from `ipHandler` to `upHandler`, and then to `tfaHandler`.
5. Finally, we call the `authenticate()` method on the `ipHandler` and pass the username and password for authentication. The request will propagate through the chain of handlers until it is handled or reaches the end of the chain.
6. Each handler performs its specific authentication logic and decides whether to handle the request or pass it to the next handler in the chain. If a handler can handle the request, it returns `true`. Otherwise, it delegates the request to the next handler.
7. If the request is handled successfully by any of the handlers, the `isAuthenticated` variable in the `Client` class will be `true`, indicating successful authentication. Otherwise, it will be `false`, indicating authentication failure.
8. Based on the value of `isAuthenticated`, we can proceed with server access if authentication is successful or handle authentication failure accordingly.

1. We start by defining an `Iterator` interface. It declares two methods: `hasNext()` to check if there are more elements, and `next()` to retrieve the next element. This interface serves as a contract for all iterators.
2. Next, we define a `Collection` interface. It declares a single method `getIterator()` that returns an instance of the `Iterator` interface. This interface represents a collection of elements and provides a way to access them using an iterator.
3. We implement the `NameIterator` class, which is a concrete implementation of the `Iterator` interface. It maintains a reference to an array of names (`names`) and a `position` variable to keep track of the current position while iterating. The `hasNext()` method checks if there are more names in the array by comparing the current position with the length of the array. The `next()` method retrieves the next name from the array and increments the position.
4. We implement the `NameCollection` class, which is a concrete implementation of the `Collection` interface. It takes an array of names in its constructor and stores them internally. The `getIterator()` method creates a new instance of the `NameIterator` class and passes the array of names to it. It returns the created iterator, which allows accessing the names in the collection.
5. In the client code (`Main` class), we create an array of names (`names`) containing "John," "Emily," "David," and "Sarah."
6. We create an instance of `NameCollection` called `collection` and pass the `names` array to its constructor. This initializes the collection with the names.
7. We retrieve an iterator from the collection by calling the `getIterator()` method. This gives us an instance of the `NameIterator` class.
8. We use a `while` loop to iterate over the collection. The loop condition checks if the iterator has more elements using the `hasNext()` method.
9. Inside the loop, we retrieve the next name from the iterator using the `next()` method and store it in the `name` variable.
10. Finally, we print each name to the console.

1

```
// Step 1: Iterator interface
interface Iterator {
    boolean hasNext();
    String next();
}
```

2

```
// Step 2: Collection interface
interface Collection {
    Iterator getIterator();
}
```

3

```
// Step 3: NameIterator implementation of Iterator interface
class NameIterator implements Iterator {
    private String[] names;
    private int position;

    public NameIterator(String[] names) {
        this.names = names;
        this.position = 0;
    }

    public boolean hasNext() {
        return position < names.length;
    }

    public String next() {
        String name = names[position];
        position++;
        return name;
    }
}
```

Participant:

1. Iterator
2. IteratableCollection : Collection
3. ConcreteIterator: NameIterator
4. ConcreteCollection : NameCollection

4

```
// Step 4: NameCollection implementation of Collection interface
class NameCollection implements Collection {
    private String[] names;
    public NameCollection(String[] names) {
        this.names = names;
    }
    public Iterator getIterator() {
        return new NameIterator(names);
    }
}
```

5

```
public class Client {
    public static void main(String[] args) {
        String[] names = {"John", "Emily", "David", "Sarah"};

        Collection collection = new NameCollection(names);
        Iterator iterator = collection.getIterator();

        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```