//start coding a new program

public class EspressoMachine {

private static EspressoMachine instance;

public static EspressoMachine getInstance(){

if(instance==null){

instance = new EspressoMachine();

}

return instance;

}

public void getCoffee(){

System.out.println("Bushzzz....");

}

public static void main(String args[]){

getInstance();

getInstance();

getInstance();

getInstance();

}

}

//ClassC.java

package com.examples.anotherpackage;

public class ClassC {

protected int a=10;

public void methodC(){

System.out.println("Method C");

}

}

//ClassB.java

//ClassA

package com.examples.mypackage;

public class ClassA {

public void methodA(){

System.out.println("Method A");

}

}

//singleton design pattern

public class Browser {

    private static Browser tab;

    public static Browser getTab() {

        if (tab == null) {

            tab = new Browser();

        }

        return tab;

    }

    public static void main(String[] args) {

        getTab();

        getTab();

        getTab();

    }

}

**EXAMPLES OF SINGLETON DESIGN PATTERNS: -**

Logger:  
In a logging system, you may want to have a single instance of a Logger class that handles logging operations throughout your application. The Singleton pattern ensures that all log entries are funneled through a single logging instance, providing centralized control and coordination of logging activities.  
  
Database Connection Pool:  
In a multi-threaded environment where multiple clients need to access a database, it is efficient to use a pool of database connections. The Singleton pattern can be used to manage and provide access to a single instance of the database connection pool, ensuring that clients share and reuse connections effectively.  
  
Configuration Manager:  
In an application that requires access to configuration settings or properties, a Singleton configuration manager can be used to provide a single point of access to the configuration data. This allows different components of the application to retrieve configuration values without creating multiple instances of the configuration manager.  
  
Print Spooler:  
In a print spooling system, where multiple processes or threads need to manage and coordinate printing tasks, a Singleton print spooler can be used to ensure that print jobs are serialized and handled by a single instance. This avoids conflicts and ensures orderly printing.  
  
GUI Components:  
In graphical user interface (GUI) frameworks, certain components, such as dialog boxes, message boxes, or application windows, may need to have only one instance throughout the application's lifetime. The Singleton pattern can be used to control the creation and access to these GUI components, ensuring consistency and preventing multiple instances from being opened.

//singleton design pattern, and stativ variable

class Demo {

    static int a;

    public void addA(int a) {

        this.a = a;

        System.out.println("A updated..");

    }

}

public class Main {

    public static void main(String[] args) {

        Demo one = new Demo();

        System.out.println(one.a);

        one.addA(5);

        Demo two = new Demo();

        System.out.print(two.a);

    }

}

//singleton design pattern

import java.time.LocalDateTime;

class Logger {

public static Logger instance;

public static Logger getInstance(){

if(instance == null) instance = new Logger();

return instance;

}

Logger(){

System.out.println("Logged: " + LocalDateTime.now());

    }

}

//Factory method design pattern

interface ChocolateBar {

    public void taste();

}

class DarkChocolateBar implements ChocolateBar {

    public void taste() {

        System.out.println("Dark Chocolaet Bar");

    }

}

class MilkChocolateBar implements ChocolateBar {

    public void taste() {

        System.out.println("Milk Chocolaet Bar");

    }

}

class ChocolateFactory {

    public ChocolateBar createChocolateBar(String type) {

        if (type == "dark") {

            return new DarkChocolateBar();

        } else if (type == "milk") {

            return new MilkChocolateBar();

        } else {

            return new DarkChocolateBar();

        }

    }

}

public class Main {

    public static void main(String[] args) {

        ChocolateFactory cf = new ChocolateFactory();

        ChocolateBar dc = cf.createChocolateBar("dark");

        ChocolateBar mc = new ChocolateFactory().createChocolateBar("milk");

        dc.taste();

        mc.taste();

    }

}

**EXAMPLES OF FACTORY METHOD: -**

Report Generation:  
In a report generation system, you may have different types of reports with varying structures or formats. The Factory Pattern can be used to create instances of the appropriate report classes based on user selections or report templates. It abstracts the creation process, allowing for easy addition or customization of different report types.  
  
Game Development:  
In game development, you often have different types of game objects or characters that need to be instantiated based on game levels or player actions. The Factory Pattern can be used to create instances of these game objects, abstracting the creation process and allowing for easy addition of new game objects or characters.  
In a modular or plugin-based system, you may have different modules or plugins that extend the functionality of the application. The Factory Pattern can be used to dynamically create instances of these modules or plugins based on configuration or user selection. It allows for easy integration and extensibility without tightly coupling the core application with specific modules.

//Factory method design pattern

interface AWS {

    public void instanceType();

}

class Linux implements AWS {

    public void instanceType() {

        System.out.println("Running Linux OS..");

    }

}

class Windows implements AWS {

    public void instanceType() {

        System.out.println("Running Windows OS..");

    }

}

class Mac implements AWS {

    public void instanceType() {

        System.out.println("Running Mac OS..");

    }

}

class PickInstance {

    public AWS createInstance(String type) {

        if (type == "linux") {

            return new Linux();

        } else if (type == "windows") {

            return new Windows();

        } else if (type == "mac") {

            return new Mac();

        }

         else {

            return new Linux();

        }

    }

}

public class Main {

    public static void main(String[] args) {

        AWS i1 = new PickInstance().createInstance("linux");

        i1.instanceType();

        new PickInstance().createInstance("mac").instanceType();

    }

}

**Exercise**

Implement a Shape Factory that creates instances of different shape objects based on user input. The program should prompt the user to enter the type of shape (circle, square, triangle) they want to create and then use the factory to create and display the shape.

//Factory method design pattern

import java.util.Scanner;

interface Shape {

    public void show();

}

class Circle implements Shape {

    public void show() {

        System.out.println("Circle");

    }

}

class Square implements Shape {

    public void show() {

        System.out.println("Square");

    }

}

class Triangle implements Shape {

    public void show() {

        System.out.println("Triangle");

    }

}

class PickInstance {

    public Shape createInstance(String type) {

        if (type == "circle") {

            return new Circle();

        } else if (type == "square") {

            return new Square();

        } else if (type == "triangle") {

            return new Triangle();

        }

         else {

            return new Circle();

        }

    }

}

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        String imp = sc.nextLine();

        Shape i1 = new PickInstance().createInstance(imp);

        i1.show();

    }

}

//command pattern

interface Command {

    void execute();

}

class TextEditor {

    public void cut(){

        System.out.println("Performing Cut");

    }

    public void copy(){

        System.out.println("Performing Copy");

    }

    public void paste(){

        System.out.println("Performing Paste");

    }

}

class CutCommand implements Command {

    private TextEditor textEditor;

    CutCommand(TextEditor textEditor){

        this.textEditor = textEditor;

    }

    public void execute(){

        textEditor.cut();

    }

}

class CopyCommand implements Command {

    private TextEditor textEditor;

    CopyCommand(TextEditor textEditor){

        this.textEditor = textEditor;

    }

    public void execute(){

        textEditor.copy();

    }

}

class PasteCommand implements Command {

    private TextEditor textEditor;

    PasteCommand(TextEditor textEditor){

        this.textEditor = textEditor;

    }

    public void execute(){

        textEditor.paste();

    }

}

class TextEditorInvoker {

    private Command command;

    void setCommand(Command command){

        this.command = command;

    }

    void executeCommand(){

        command.execute();

    }

}

public class Main {

    public static void main(String args[]){

        TextEditor textEditor = new TextEditor();

        TextEditorInvoker invoker = new TextEditorInvoker();

        Command cutCommand = new CutCommand(textEditor);

        Command copyCommand = new CopyCommand(textEditor);

        Command pasteCommand = new PasteCommand(textEditor);

        invoker.setCommand(cutCommand);

        invoker.executeCommand();

        invoker.setCommand(copyCommand);

        invoker.executeCommand();

        invoker.setCommand(pasteCommand);

        invoker.executeCommand();

    }

}

**Exercise**

Imagine you have a smart home system with different smart devices like lights, a TV, and a music player. You want to control these devices using a remote control. The Command Design Pattern can be applied here.

In this analogy:

* The remote control represents the client.
* Each button on the remote control represents a command.
* The smart devices (lights, TV, music player) are the receivers, which know how to perform specific actions.
* interface Command {
* public void execute();
* }
* class Light {
* public void lightOn() {
* System.out.println("Light on");
* }
* }
* class TV {
* public void tvOn() {
* System.out.println("TV on");
* }
* }
* class MusicPlayer {
* public void musicPlayerOn() {
* System.out.println("Music Player on");
* }
* }
* class LightOnCommand implements Command {
* private Light light;
* public LightOnCommand(Light light) {
* this.light = light;
* }
* public void execute() {
* light.lightOn();
* }
* }
* class TvOnCommand implements Command {
* private TV tv;
* public TvOnCommand(TV tv) {
* this.tv = tv;
* }
* public void execute() {
* tv.tvOn();
* }
* }
* class MusicPlayerOnCommand implements Command {
* private MusicPlayer musicPlayer;
* public MusicPlayerOnCommand(MusicPlayer musicPlayer) {
* this.musicPlayer = musicPlayer;
* }
* public void execute() {
* musicPlayer.musicPlayerOn();
* }
* }
* class RemoteControl {
* private Command command;
* public void setCommand(Command command) {
* this.command = command;
* }
* public void pressButton() {
* command.execute();
* }
* }
* public class Main {
* public static void main(String[] args) {
* Light livingRoomLight = new Light();
* RemoteControl control = new RemoteControl();
* Command lightOnCommand = new LightOnCommand(livingRoomLight);
* control.setCommand(lightOnCommand);
* control.pressButton();
* }
* }

**Exercise**

Implement a simple order system for a restaurant using the Command Pattern. The system should allow customers to place orders for different food items and have the orders executed by the kitchen staff.

In this analogy:

* KitchenStaff will be receiver
* Waiter will be invoker
* OrderCommand will be concrete command

**Exercise**

Implement a coffee machine program that supports operations like brewing coffee, adding ingredients, and undoing the previous action using the Command Pattern.

Design your own Receiver, Command etc

interface Command {

    public void make();

}

class Machine {

    public void brewing() {

        System.out.println("Brewing Coffee..");

    }

    public void addIngredience() {

        System.out.println("Adding Ingredience");

    }

    public void undo() {

        System.out.println("Undoing the previous action");

    }

}

class BrewingCoffee implements Command {

    private Machine m;

    public BrewingCoffee(Machine m) {

        this.m = m;

    }

    public void make() {

        m.brewing();

    }

}

class AddingIngredients implements Command {

    private Machine m;

    public AddingIngredients(Machine m) {

        this.m = m;

    }

    public void make() {

        m.addIngredience();

    }

}

class UndoPreviousAction implements Command {

    private Machine m;

    public UndoPreviousAction(Machine m) {

        this.m = m;

    }

    public void make() {

        m.undo();

    }

}

class Reciver {

    private Command c;

    public void addMachine(Command c) {

        this.c = c;

    }

    public void prepare() {

        c.make();

    }

}

public class Main {

    public static void main(String[] args) {

        Machine machine = new Machine();

        Reciver reciver = new Reciver();

        Command brew = new BrewingCoffee(machine);

        reciver.addMachine(brew);

        reciver.prepare();

        Command undo = new UndoPreviousAction(machine);

        reciver.addMachine(undo);

        reciver.prepare();

    }

}