M2006C3MII

6:14 PM

UNIT – 3

Structural Pattern:

The adapter pattern:

We can learn the concept of adapter pattern using many real world examples of adapter. In real word adapter is generally used to provide suitable interface for the standard plugs so that using standard plug the user device receive the desired power. (Some time there is need to step up or down to match the connecting device need)

Object oriented adapters are also play the same role like real word adapters. The object oriented adapter acts as a middleman by receiving request from the client and converting them into the request that make sense to the vendor class.

Consider the example as shown below the existing system and new vendor class designed is

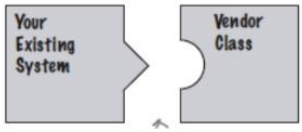
Type here to search 28°C ENG 18:12 03-08-2023

M2006C3MII

6:14 PM

them into the request that make sense to the vendor class.

Consider the example as shown below the existing system and new vendor class designed is different than the old one so to solve the problem we need to change the existing code .(but we can't change the vendor code)



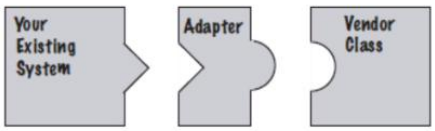
The solution of above problem is to create an adapter class that adapts the new vendor interface into the one we are expecting as shown below.

Type here to search 28°C ENG 18:12 03-08-2023

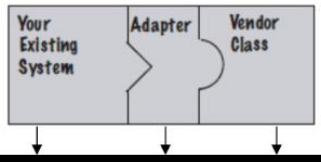
M2006C3MII

6:14 PM

The solution of above problem is to create an adapter class that adapts the new vendor interface into the one we are expecting as shown below.



The adapter implements the interface which is required and connect (message passing) with the vendor interface to service new vendor request as shown below.

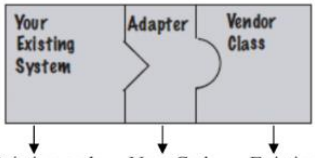


Type here to search

28°C ENG 18:12 03-08-2023

M2006C3MII

6:14 PM



Existing code New Code Existing code

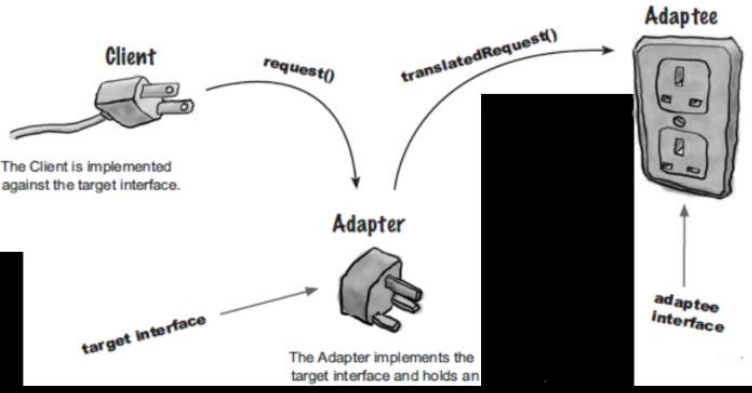
Type here to search

28°C ENG 18:12 03-08-2023

M2006C3MII

6:14 PM

Working of Adapter Pattern:



Type here to search

28°C ENG 18:12 03-08-2023

M2006C3MII

6:14 PM

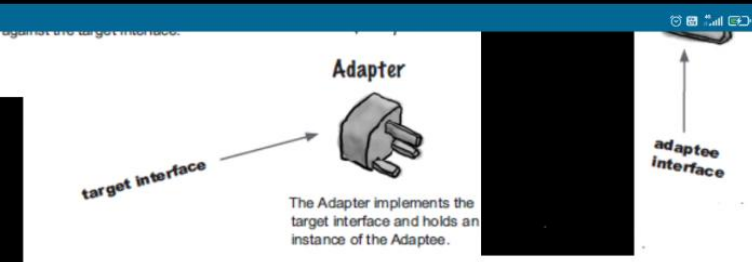


Fig 3.1 Working of Adapter pattern

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request into one or more calls on the adaptee using the adaptee interface.

Type here to search

28°C ENG 18:13 03-08-2023

M2006C3MII

6:14 PM

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request into one or more calls on the adaptee using the adaptee interface.
3. The client receive the result of the call and never knows that there is an adapter doing the translation
4. The client and adaptee are decouple both don't know about each other.

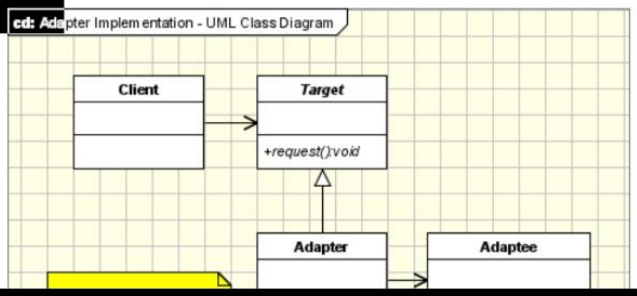
Type here to search 28°C ENG 18:13 03-08-2023

M2006C3MII

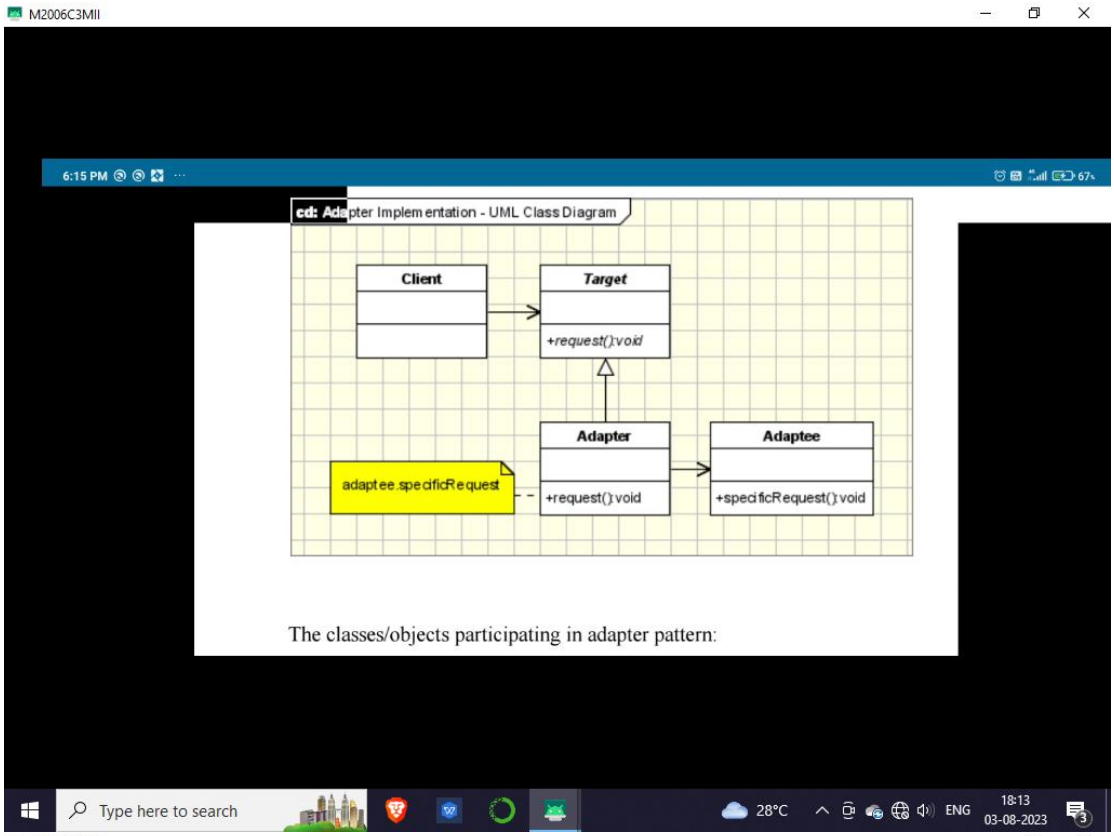
6:15 PM

Intent

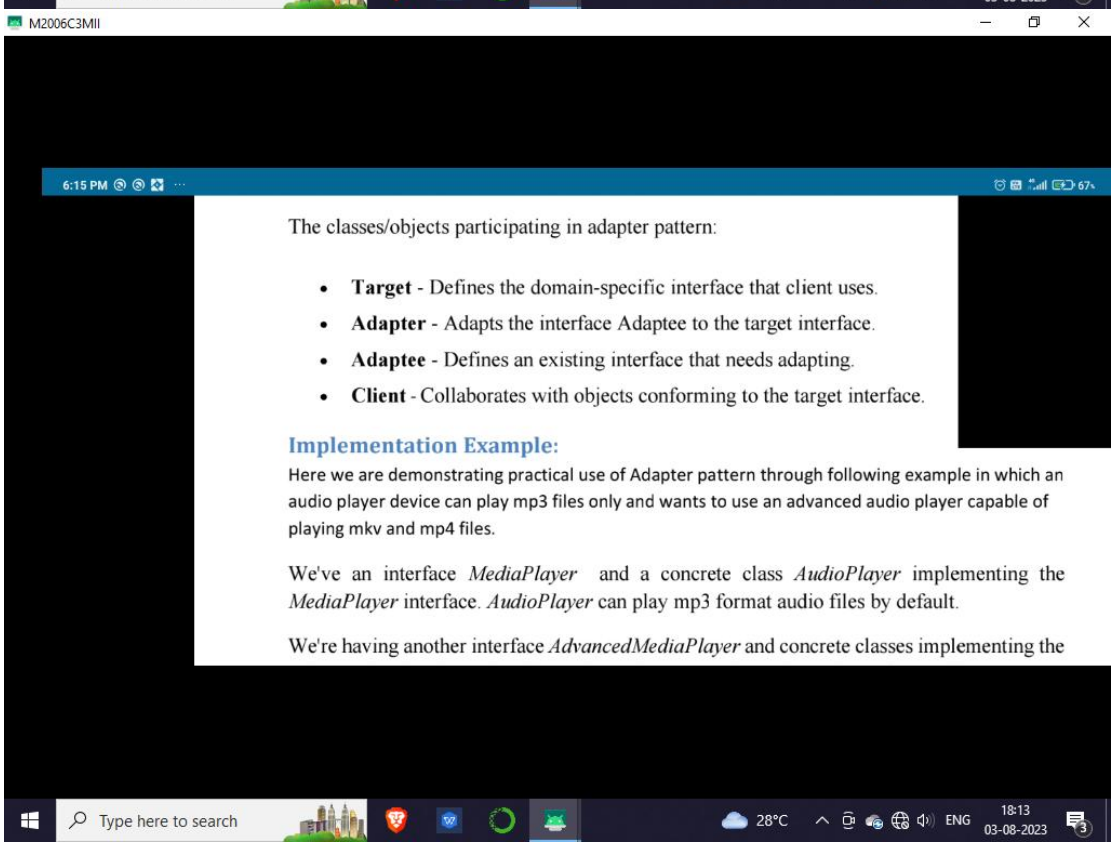
- Convert the interface of a class into another interface clients expect.
- Adapter classes' work together, that could not otherwise because of incompatible interfaces.



Type here to search 28°C ENG 18:13 03-08-2023



The classes/objects participating in adapter pattern:



The classes/objects participating in adapter pattern:

- **Target** - Defines the domain-specific interface that client uses.
- **Adapter** - Adapts the interface Adaptee to the target interface.
- **Adaptee** - Defines an existing interface that needs adapting.
- **Client** - Collaborates with objects conforming to the target interface.

Implementation Example:

Here we are demonstrating practical use of Adapter pattern through following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing mkv and mp4 files.

We've an interface *MediaPlayer* and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

We're having another interface *AdvancedMediaPlayer* and concrete classes implementing the

M2006C3MII

6:15 PM

Implementation Example:

Here we are demonstrating practical use of Adapter pattern through following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing mkv and mp4 files.

We've an interface *MediaPlayer* and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

We're having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play mkv and mp4 format files.

We want to make *AudioPlayer* to play other formats as well. To attain this, we've created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

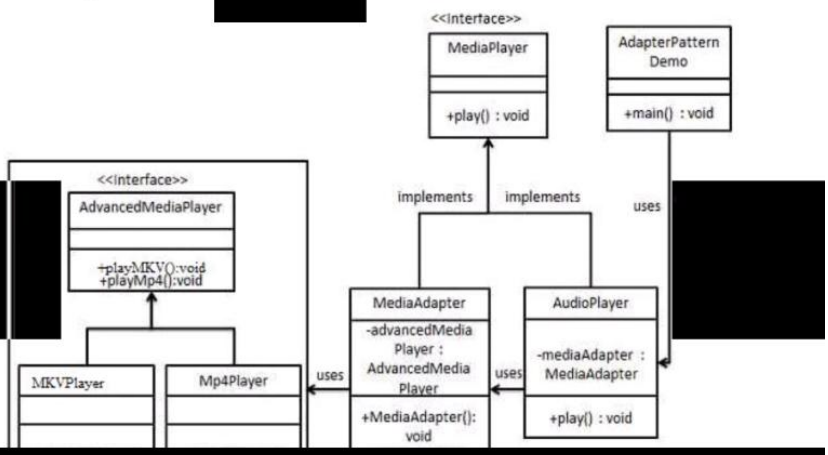
AudioPlayer uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, this demo class will use *AudioPlayer* class to play various formats.

Type here to search 28°C ENG 18:13 03-08-2023

M2006C3MII

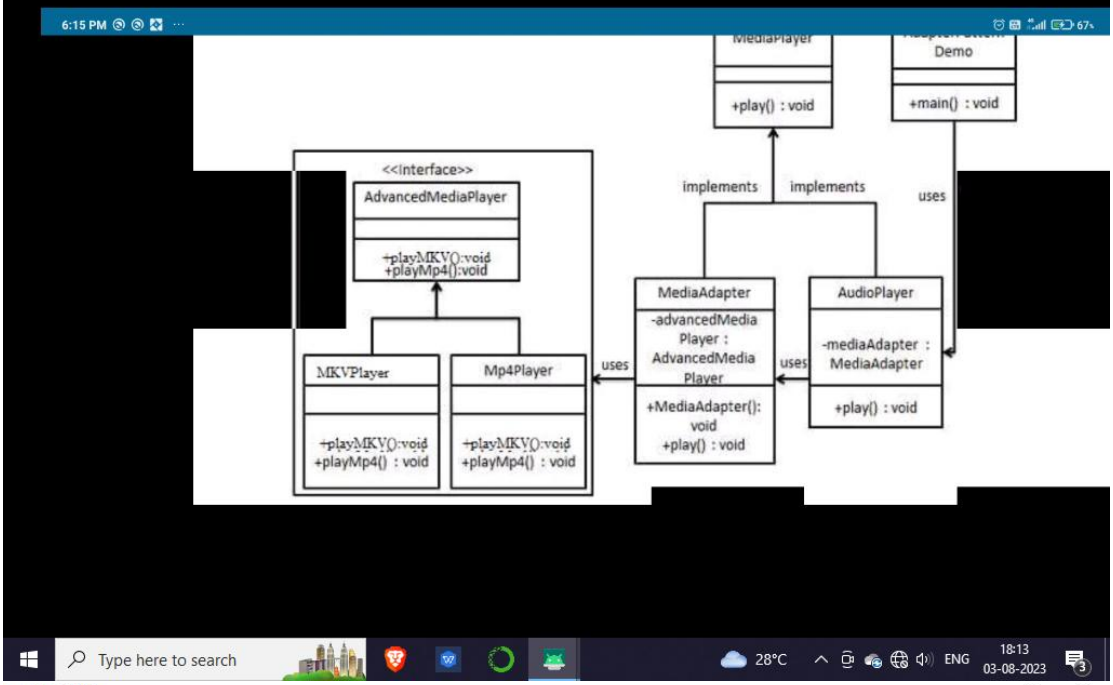
6:15 PM

Class Diagram:

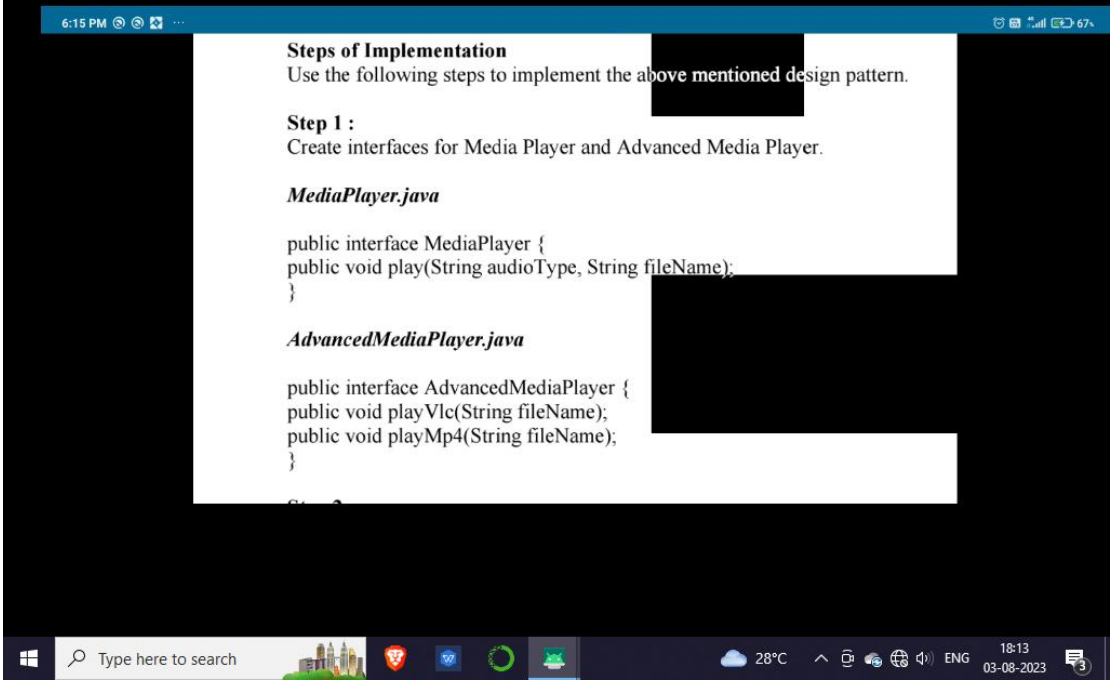


Type here to search 28°C ENG 18:13 03-08-2023

M2006C3MII



M2006C3MII



Steps of Implementation
Use the following steps to implement the above mentioned design pattern.

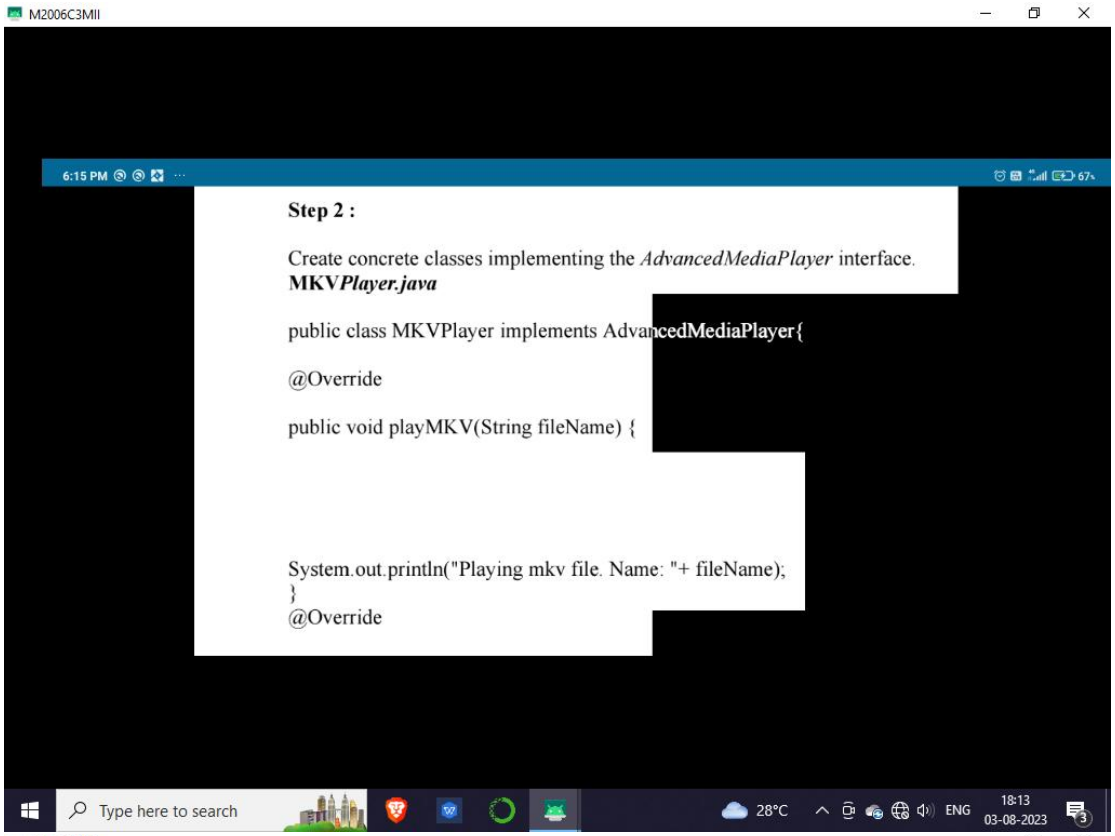
Step 1 :
Create interfaces for Media Player and Advanced Media Player.

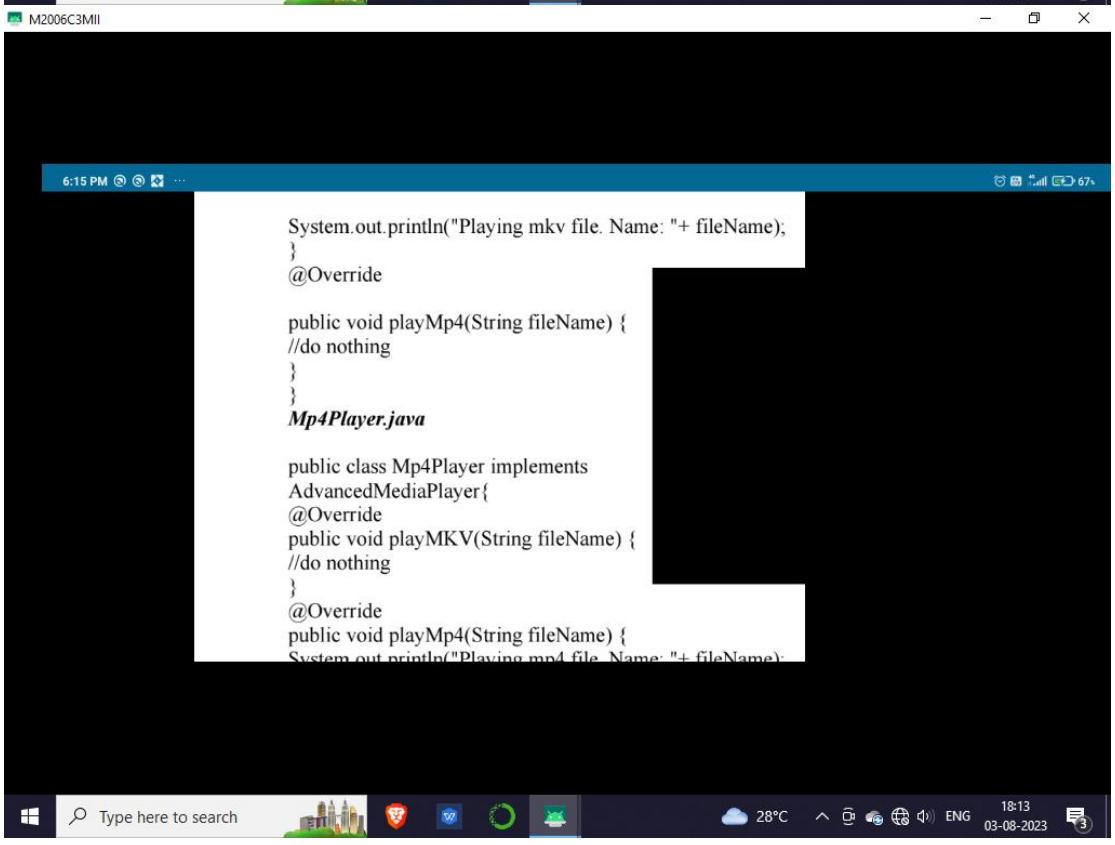
MediaPlayer.java

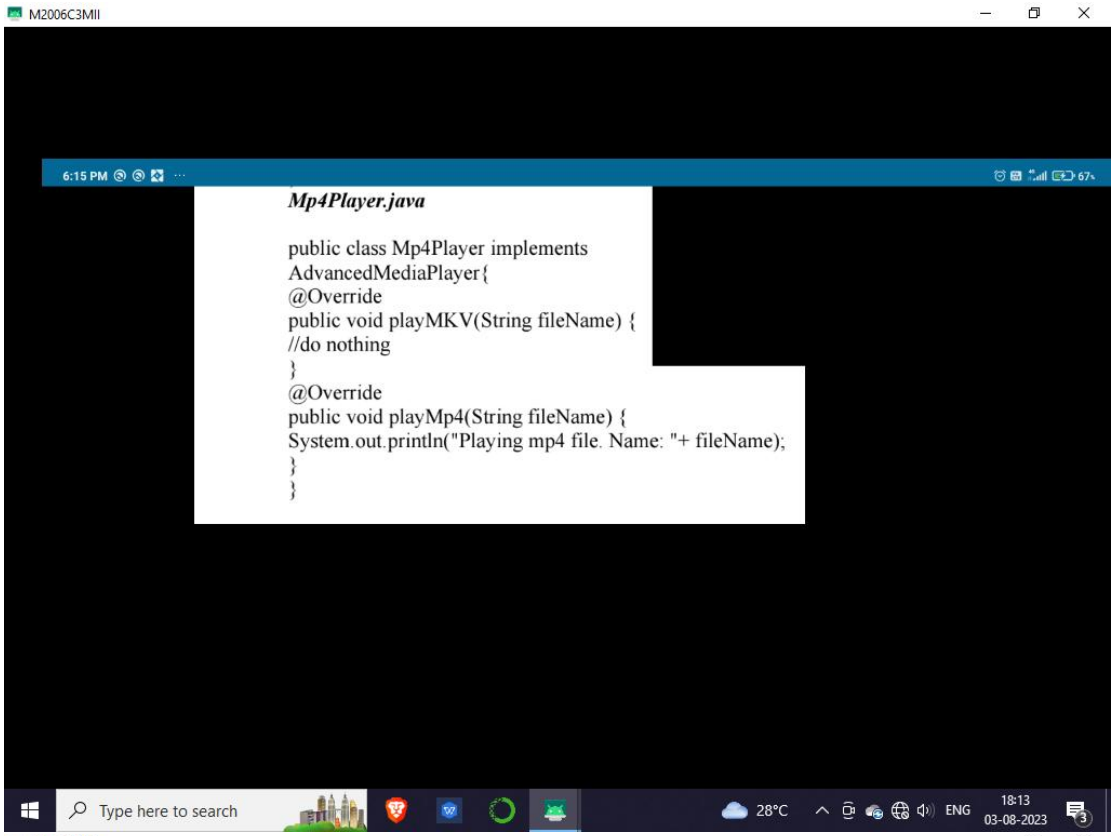
```
public interface MediaPlayer {
public void play(String audioType, String fileName);
}
```

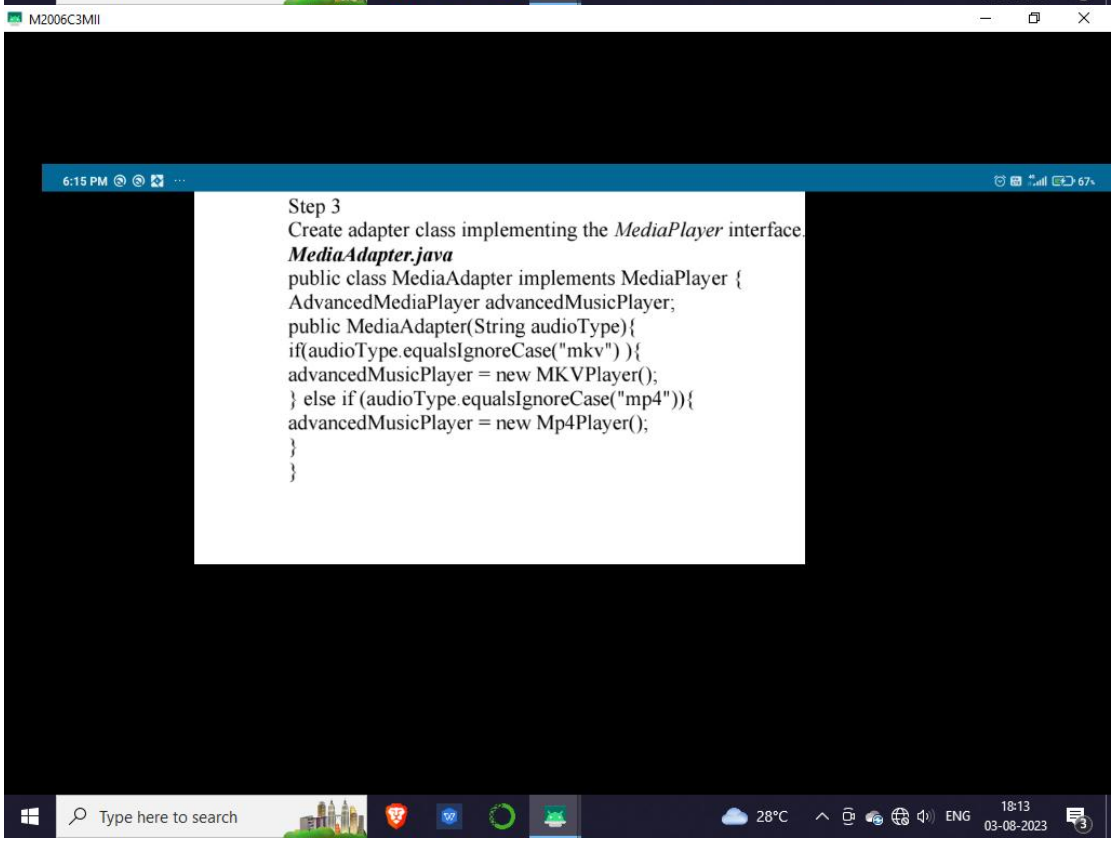
AdvancedMediaPlayer.java

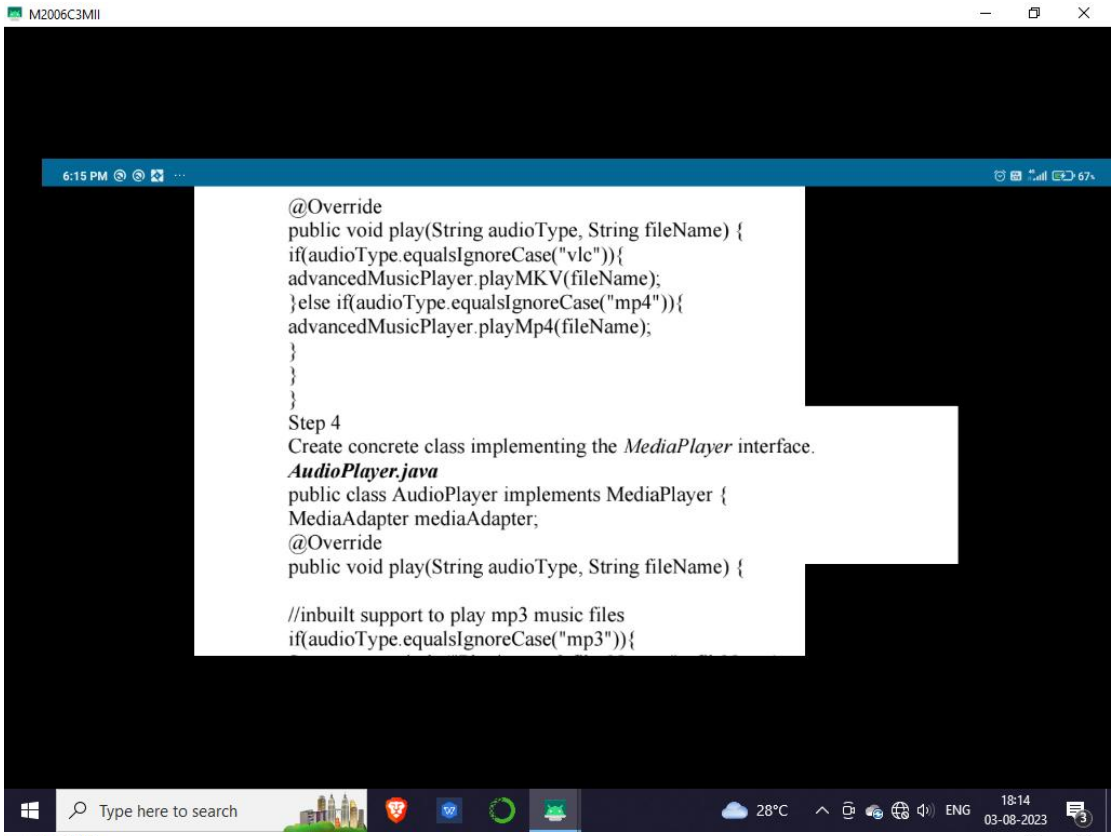
```
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName);
}
```

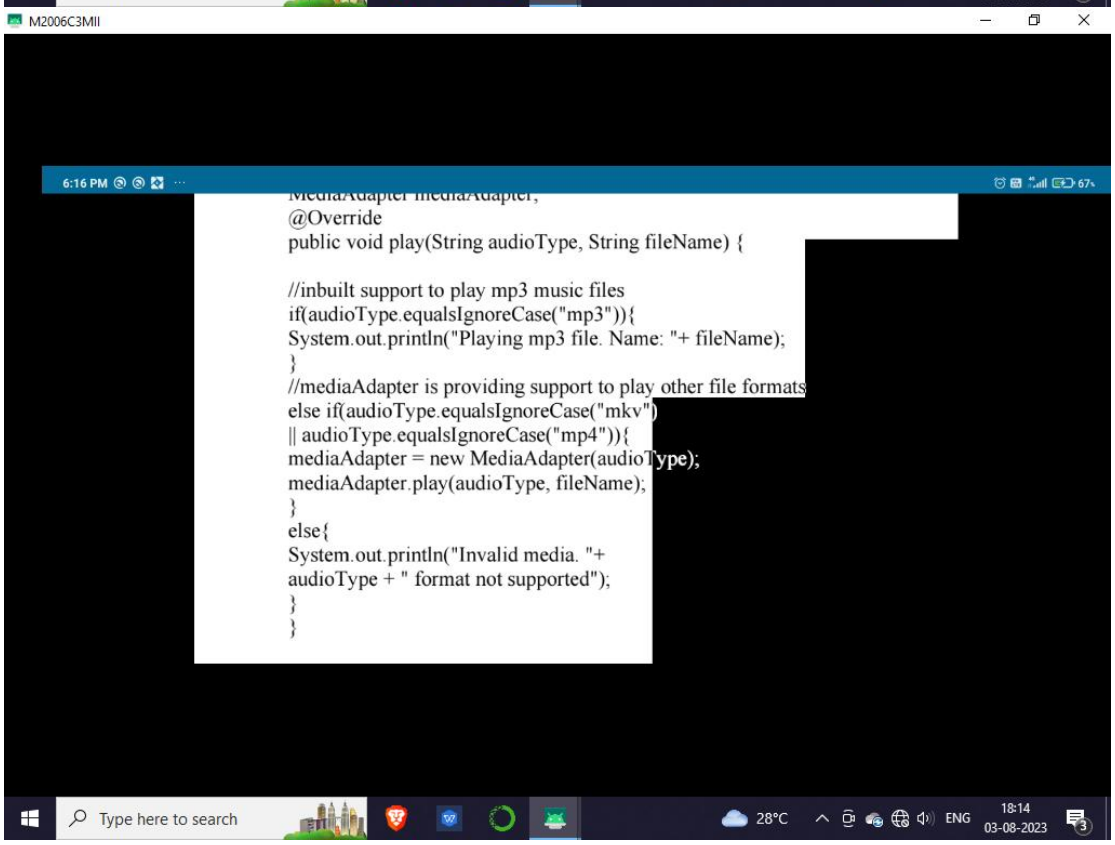

M2006C3MII
6:15 PM
Step 2 :
Create concrete classes implementing the AdvancedMediaPlayer interface.
MKVPlayer.java
public class MKVPlayer implements AdvancedMediaPlayer{
@Override
public void playMKV(String fileName) {
System.out.println("Playing mkv file. Name: "+ fileName);
}
@Override
Type here to search
28°C
ENG
18:13
03-08-2023

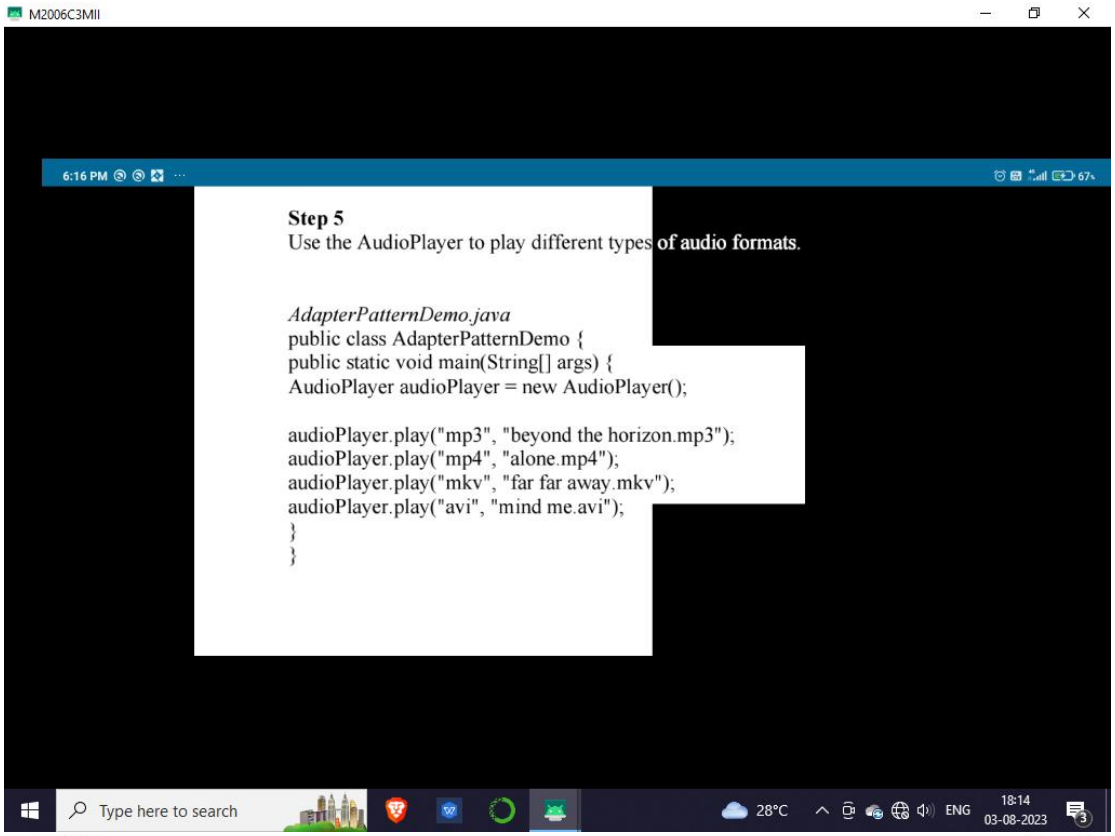

M2006C3MII
6:15 PM
System.out.println("Playing mkv file. Name: "+ fileName);
}
@Override
public void playMp4(String fileName) {
//do nothing
}
}
Mp4Player.java
public class Mp4Player implements
AdvancedMediaPlayer{
@Override
public void playMKV(String fileName) {
//do nothing
}
@Override
public void playMp4(String fileName) {
Type here to search
28°C
ENG
18:13
03-08-2023

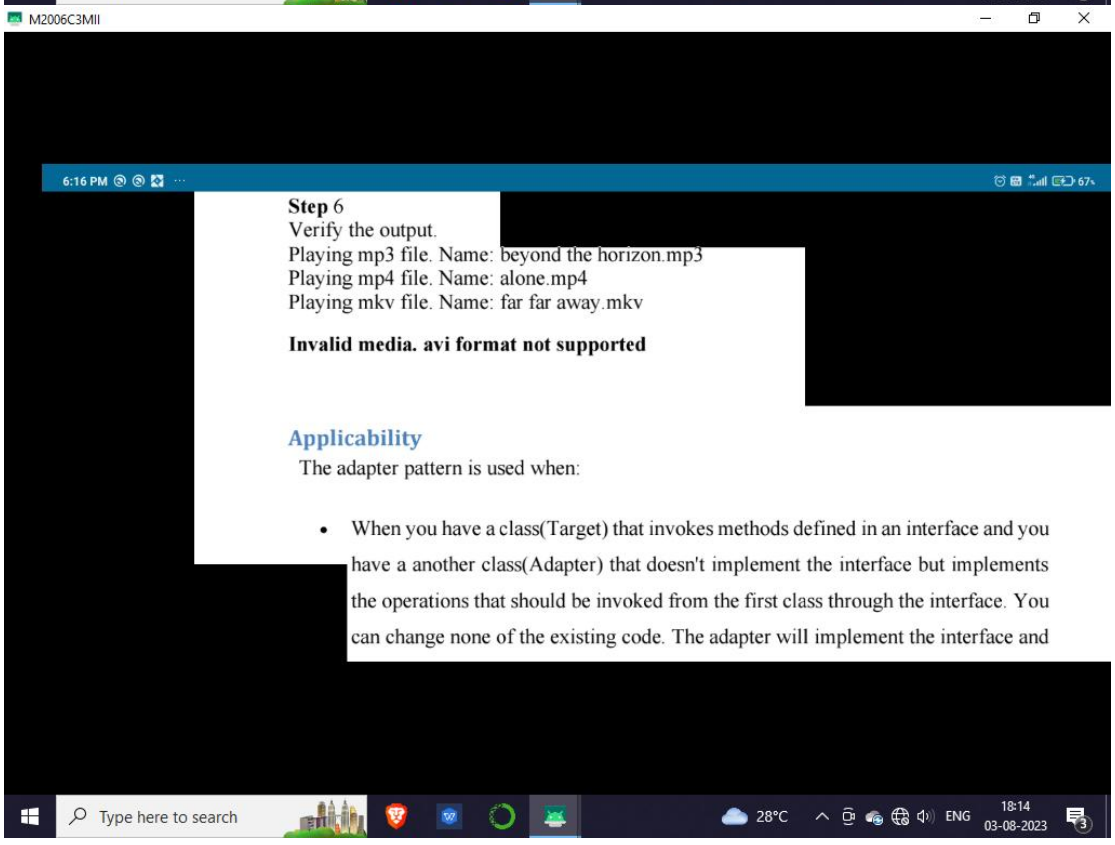

M2006C3MII
6:15 PM
Mp4Player.java
public class Mp4Player implements
AdvancedMediaPlayer{
@Override
public void playMKV(String fileName) {
//do nothing
}
@Override
public void playMp4(String fileName) {
System.out.println("Playing mp4 file. Name: "+ fileName);
}
}
Type here to search
28°C
ENG
18:13
03-08-2023

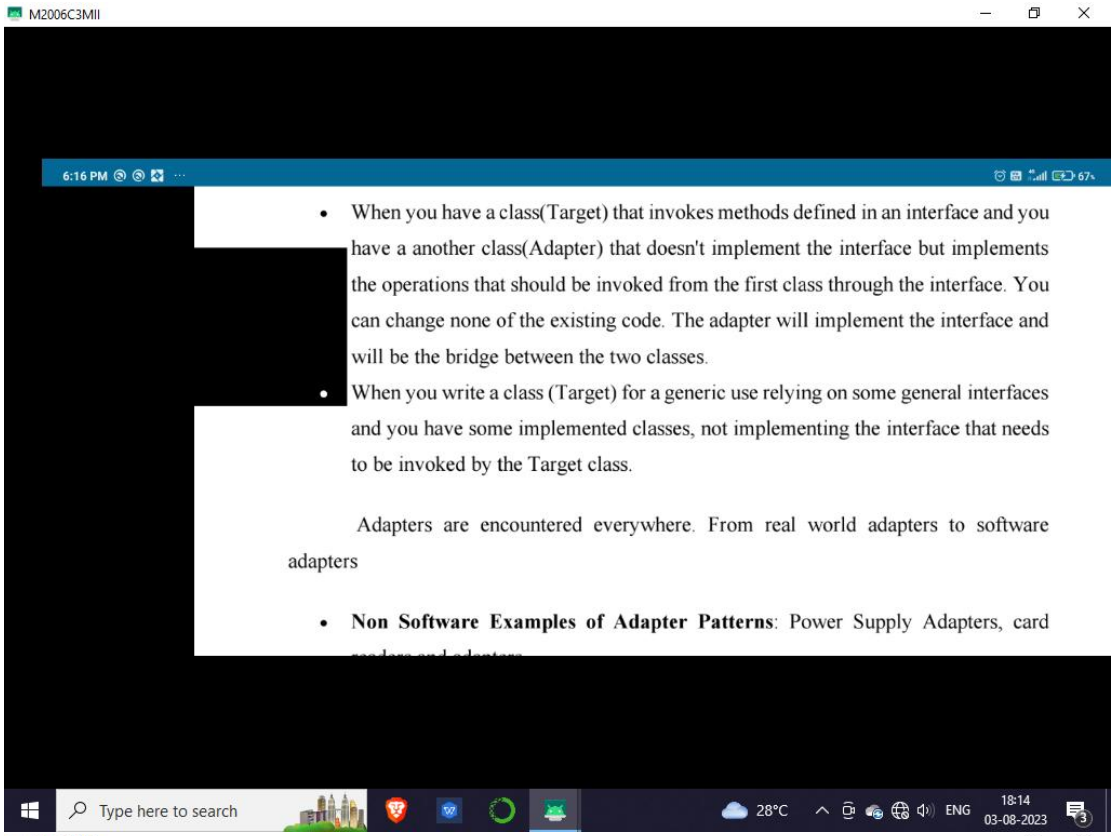

M2006C3MII
6:15 PM
Step 3
Create adapter class implementing the MediaPlayer interface.
MediaAdapter.java
public class MediaAdapter implements MediaPlayer {
AdvancedMediaPlayer advancedMusicPlayer;
public MediaAdapter(String audioType){
if(audioType.equalsIgnoreCase("mkv")){
advancedMusicPlayer = new MKVPlayer();
} else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();
}
}
Type here to search
28°C
ENG
18:13
03-08-2023

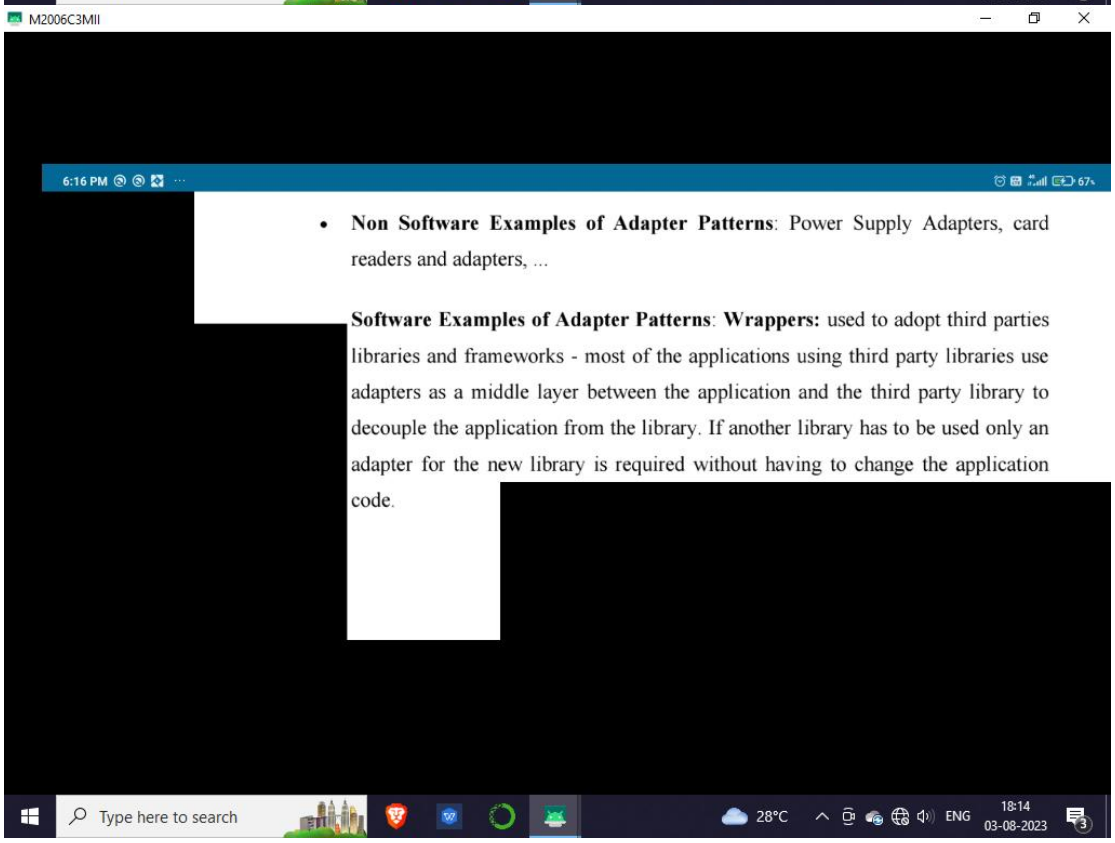

M2006C3MII
6:15 PM
@Override
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){
advancedMusicPlayer.playMKV(fileName);
}else if(audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer.playMp4(fileName);
}
}
}
Step 4
Create concrete class implementing the MediaPlayer interface.
AudioPlayer.java
public class AudioPlayer implements MediaPlayer {
MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
//inbuilt support to play mp3 music files
if(audioType.equalsIgnoreCase("mp3")){
Type here to search
28°C
ENG
18:14
03-08-2023


M2006C3MII
6:16 PM
@Override
public void play(String audioType, String fileName) {
//inbuilt support to play mp3 music files
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: "+ fileName);
}
//mediaAdapter is providing support to play other file formats
else if(audioType.equalsIgnoreCase("mkv")
|| audioType.equalsIgnoreCase("mp4")){
mediaAdapter = new MediaAdapter(audioType);
mediaAdapter.play(audioType, fileName);
}
else{
System.out.println("Invalid media. "+
audioType + " format not supported");
}
}
Type here to search
28°C
ENG
18:14
03-08-2023


M2006C3MII
6:16 PM
Step 5
Use the AudioPlayer to play different types of audio formats.
AdapterPatternDemo.java
public class AdapterPatternDemo {
public static void main(String[] args) {
AudioPlayer audioPlayer = new AudioPlayer();
audioPlayer.play("mp3", "beyond the horizon.mp3");
audioPlayer.play("mp4", "alone.mp4");
audioPlayer.play("mkv", "far far away.mkv");
audioPlayer.play("avi", "mind me.avi");
}
}
Type here to search
28°C
ENG
18:14
03-08-2023


M2006C3MII
6:16 PM
Step 6
Verify the output.
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing mkv file. Name: far far away.mkv
Invalid media. avi format not supported
Applicability
The adapter pattern is used when:
When you have a class(Target) that invokes methods defined in an interface and you have a another class(Adapter) that doesn't implement the interface but implements the operations that should be invoked from the first class through the interface. You can change none of the existing code. The adapter will implement the interface and
Type here to search
28°C
ENG
18:14
03-08-2023


M2006C3MII
6:16 PM
67%
When you have a class(Target) that invokes methods defined in an interface and you have a another class(Adapter) that doesn't implement the interface but implements the operations that should be invoked from the first class through the interface. You can change none of the existing code. The adapter will implement the interface and will be the bridge between the two classes.
When you write a class (Target) for a generic use relying on some general interfaces and you have some implemented classes, not implementing the interface that needs to be invoked by the Target class.
Adapters are encountered everywhere. From real world adapters to software adapters
Non Software Examples of Adapter Patterns: Power Supply Adapters, card
Type here to search
28°C
ENG
18:14
03-08-2023


M2006C3MII
6:16 PM
67%
Non Software Examples of Adapter Patterns: Power Supply Adapters, card readers and adapters, ...
Software Examples of Adapter Patterns: Wrappers: used to adopt third parties libraries and frameworks - most of the applications using third party libraries use adapters as a middle layer between the application and the third party library to decouple the application from the library. If another library has to be used only an adapter for the new library is required without having to change the application code.
Type here to search
28°C
ENG
18:14
03-08-2023

M2006C3MII

6:16 PM

Specific problems and implementation

Objects Adapters (Based on Delegation)

Objects Adapters are the classical example of the adapter pattern. It uses composition. The Adaptee delegates the calls to Adaptee (opossed to class adapters which extends the Adaptee). This behavior gives us a few advantages over the class adapters (however the class adapters can be implemented in languages allowing multiple inheritance).

The main advantage is that the Adapter adapts not only the Adpatee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well. The main disadvantage is

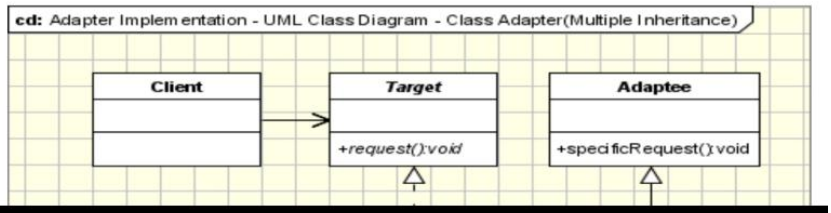
Type here to search 28°C ENG 18:14 03-08-2023

M2006C3MII

6:16 PM

The main advantage is that the Adapter adapts not only the Adpatee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well. The main disadvantage is that it requires to write all the code for delegating all the necessary requests to the Adaptee.

Class Adapters - Based on (Multiple) Inheritance

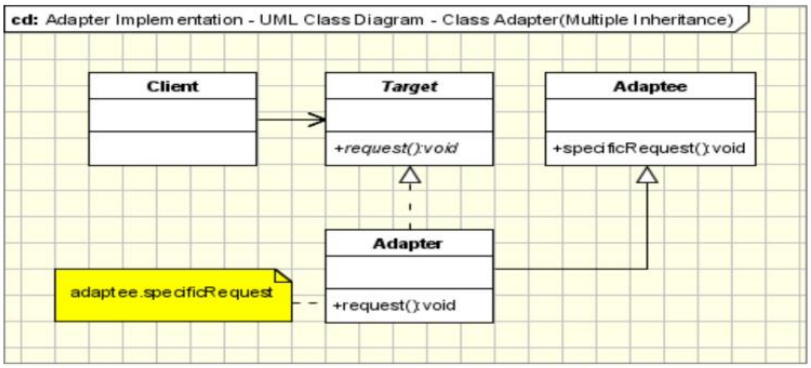


Type here to search 28°C ENG 18:14 03-08-2023

M2006C3MII

6:16 PM

Class Adapters - Based on (Multiple) Inheritance



Type here to search

28°C ENG 18:14 03-08-2023

M2006C3MII

6:16 PM

Class adapters can be implemented in languages supporting multiple inheritance (Java, C# or PHP does not support multiple inheritance). Thus, such adapters cannot be easy implemented in Java, C# or VB.NET. Class adapter uses inheritance instead of composition. It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.

T

Type here to search

28°C ENG 18:14 03-08-2023

M2006C3MII

6:16 PM

Bridge Pattern:

Definition: Bridge pattern provide a facility to progressively add functionality while separating out major differences using abstract classes.

Intent

- The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently.

Motivation

Sometimes an abstraction should have different implementations; consider an object that handles persistence of objects over different platforms using either relational databases or file system structures (files and folders). A simple implementation might choose to extend the object itself to implement the functionality for both file system and RDBMS. However

Type here to search 28°C ENG 18:14 03-08-2023

M2006C3MII

6:16 PM

Motivation

Sometimes an abstraction should have different implementations; consider an object that handles persistence of objects over different platforms using either relational databases or file system structures (files and folders). A simple implementation might choose to extend the object itself to implement the functionality for both file system and RDBMS. However this implementation would create a problem; Inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementation independently.

Implementation

As shown in the class diagram we've created an interface *DrawGraphics which* is acting as a bridge implementer and concrete classes *BlueCircle*, *YellowCircle* implementing the *DrawGraphics* interface. *Shape* is an abstract class and will use object of *DrawGraphics*.

Type here to search 28°C ENG 18:15 03-08-2023

M2006C3MII

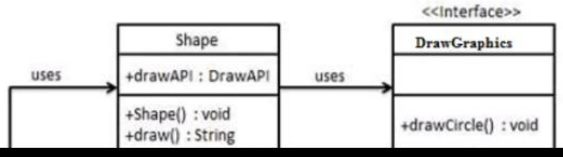
6:17 PM

implementation independently.

Implementation

As shown in the class diagram we've created an interface *DrawGraphics which* is acting as a bridge implementer and concrete classes *BlueCircle*, *YellowCircle* implementing the *DrawGraphics* interface. *Shape* is an abstract class and will use object of *DrawGraphics*. *BridgePatternDemo* class will use *Shape* class to draw different color circle.

Class Diagram:

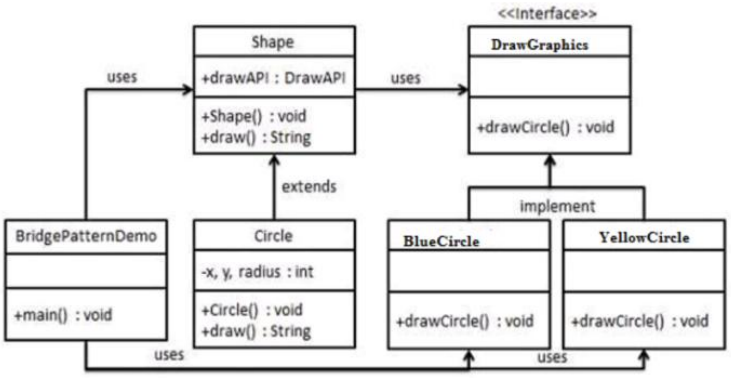


Type here to search 28°C ENG 18:15 03-08-2023

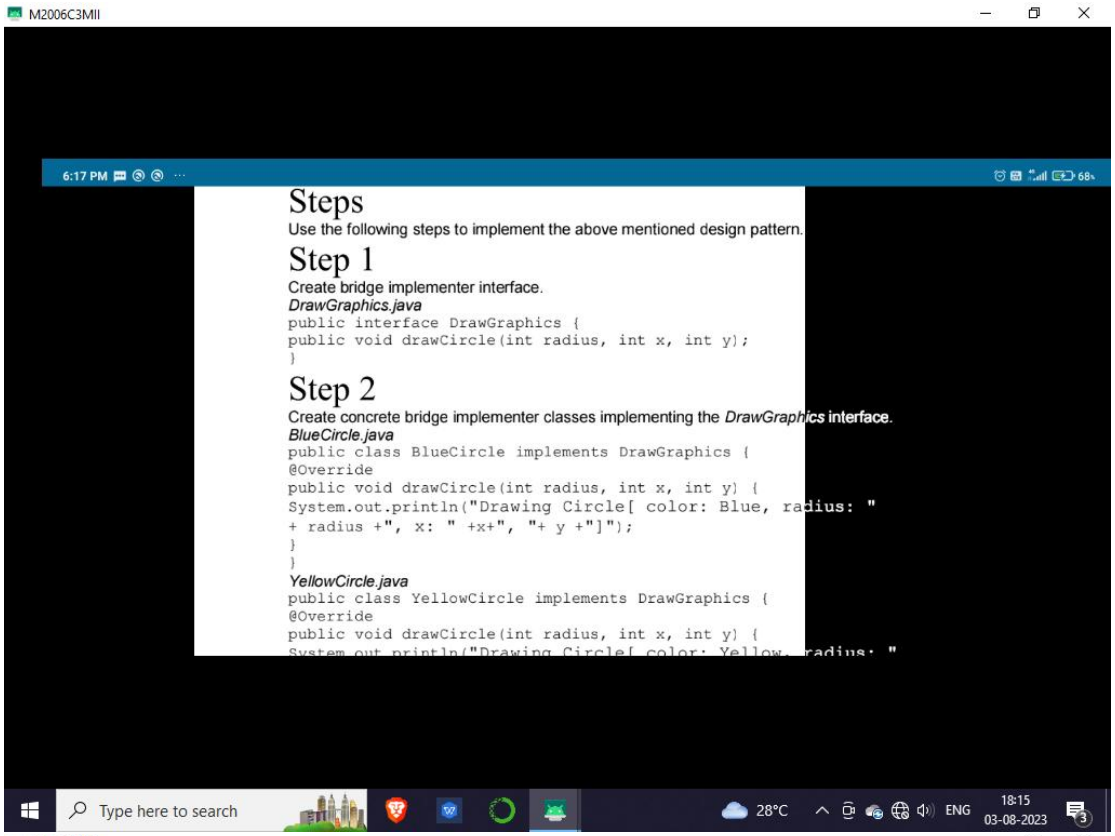
M2006C3MII

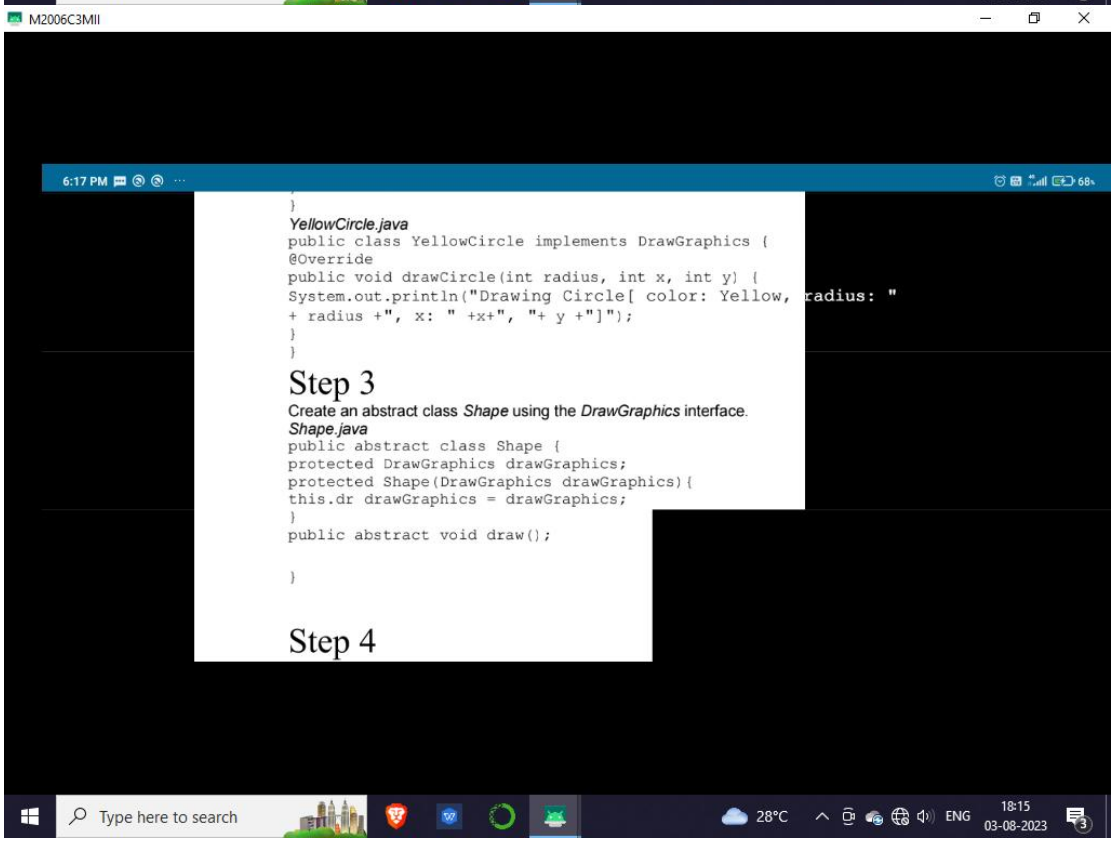
6:17 PM

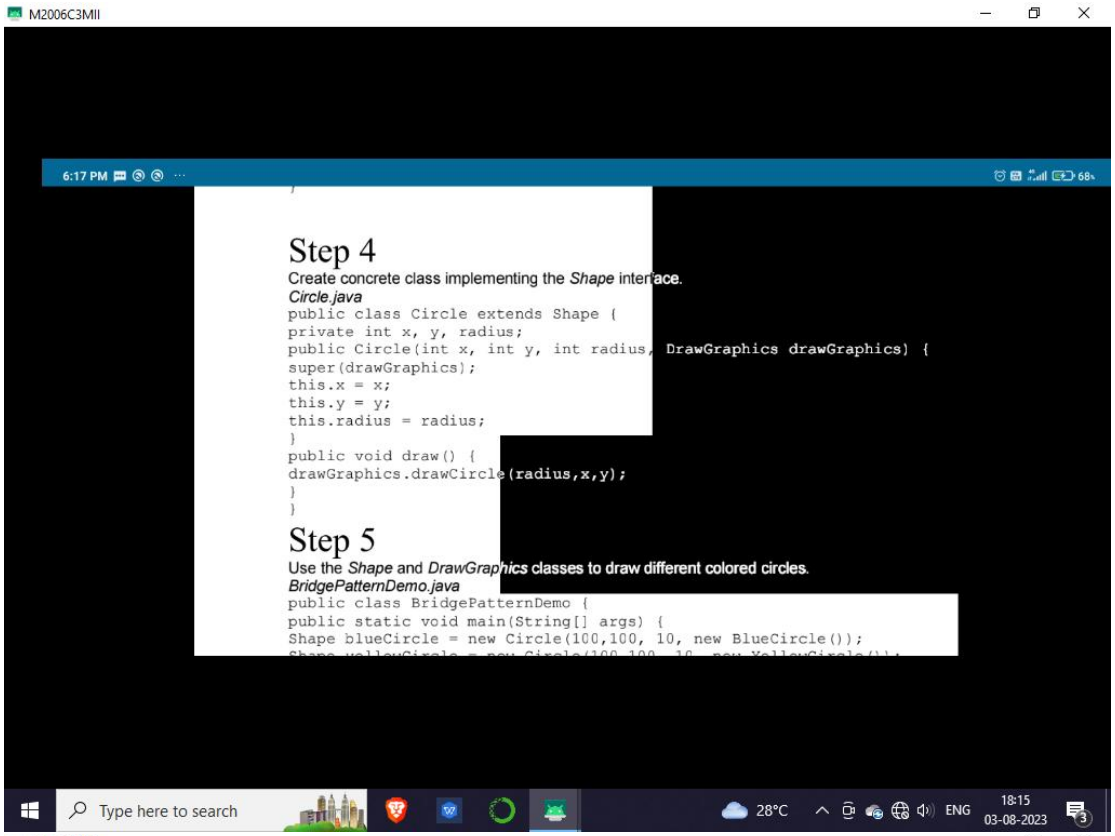
Class Diagram:

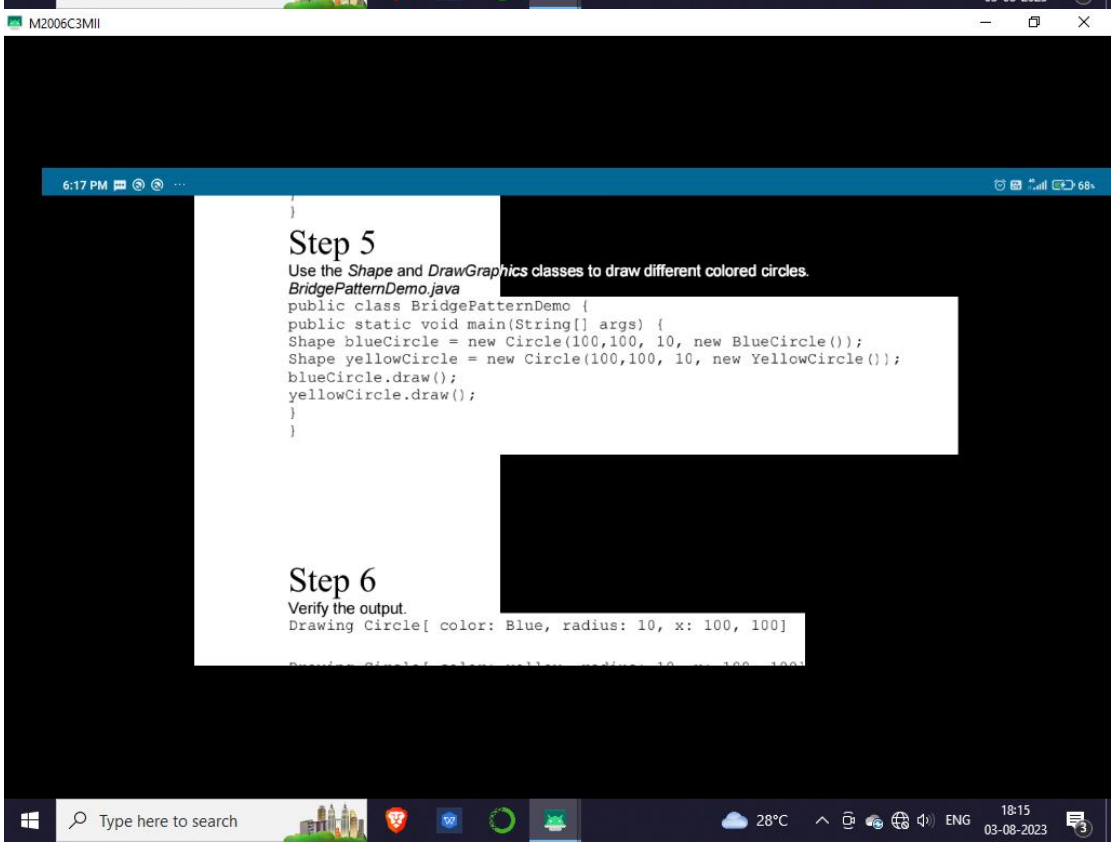


Type here to search 28°C ENG 18:15 03-08-2023


M2006C3MII
6:17 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create bridge implementer interface.
DrawGraphics.java
public interface DrawGraphics {
public void drawCircle(int radius, int x, int y);
}
Step 2
Create concrete bridge implementer classes implementing the DrawGraphics interface.
BlueCircle.java
public class BlueCircle implements DrawGraphics {
@Override
public void drawCircle(int radius, int x, int y) {
System.out.println("Drawing Circle[color: Blue, radius: "
+ radius +", x: " +x+", "+ y +"]");
}
}
YellowCircle.java
public class YellowCircle implements DrawGraphics {
@Override
public void drawCircle(int radius, int x, int y) {
Type here to search
28°C
ENG
18:15
03-08-2023


M2006C3MII
6:17 PM
}
YellowCircle.java
public class YellowCircle implements DrawGraphics {
@Override
public void drawCircle(int radius, int x, int y) {
System.out.println("Drawing Circle[color: Yellow, radius: "
+ radius +", x: " +x+", "+ y +"]");
}
}
Step 3
Create an abstract class Shape using the DrawGraphics interface.
Shape.java
public abstract class Shape {
protected DrawGraphics drawGraphics;
protected Shape(DrawGraphics drawGraphics){
this.dr drawGraphics = drawGraphics;
}
public abstract void draw();
}
Step 4
Type here to search
28°C
ENG
18:15
03-08-2023


M2006C3MII
6:17 PM
Step 4
Create concrete class implementing the Shape interface.
Circle.java
public class Circle extends Shape {
private int x, y, radius;
public Circle(int x, int y, int radius, DrawGraphics drawGraphics) {
super(drawGraphics);
this.x = x;
this.y = y;
this.radius = radius;
}
public void draw() {
drawGraphics.drawCircle(radius,x,y);
}
}
Step 5
Use the Shape and DrawGraphics classes to draw different colored circles.
BridgePatternDemo.java
public class BridgePatternDemo {
public static void main(String[] args) {
Shape blueCircle = new Circle(100,100, 10, new BlueCircle());
Type here to search
28°C
ENG
18:15
03-08-2023


M2006C3MII
6:17 PM
}
Step 5
Use the Shape and DrawGraphics classes to draw different colored circles.
BridgePatternDemo.java
public class BridgePatternDemo {
public static void main(String[] args) {
Shape blueCircle = new Circle(100,100, 10, new BlueCircle());
Shape yellowCircle = new Circle(100,100, 10, new YellowCircle());
blueCircle.draw();
yellowCircle.draw();
}
}
Step 6
Verify the output.
Drawing Circle[color: Blue, radius: 10, x: 100, 100]
Type here to search
28°C
ENG
18:15
03-08-2023

M2006C3MII

6:17 PM

Step 6

Verify the output.

```
Drawing Circle[ color: Blue, radius: 10, x: 100, 100]

Drawing Circle[ color: yellow, radius: 10, x: 100, 100]
```

Description

An Abstraction can be implemented by an abstraction implementation, and this implementation does not depend on any concrete implementers of the Implementer interface. Extending the abstraction does not affect the Implementer. Also extending the Implementer has no effect on the Abstraction.

Related Patterns

Type here to search

28°C ENG 18:15 03-08-2023

M2006C3MII

6:17 PM

Related Patterns

- **Abstract Factory Pattern** - An Abstract Factory pattern can be used create and configure a particular Bridge, for example a factory can choose the suitable concrete implementor at runtime.

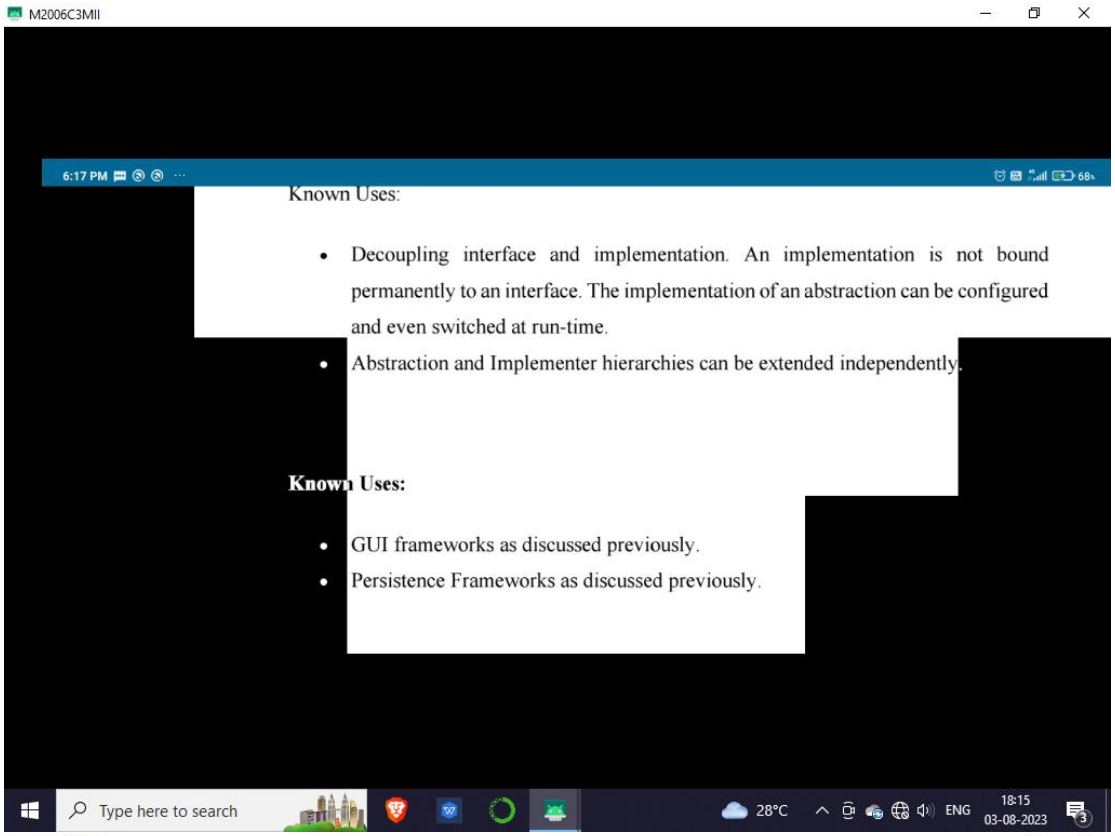
Consequences

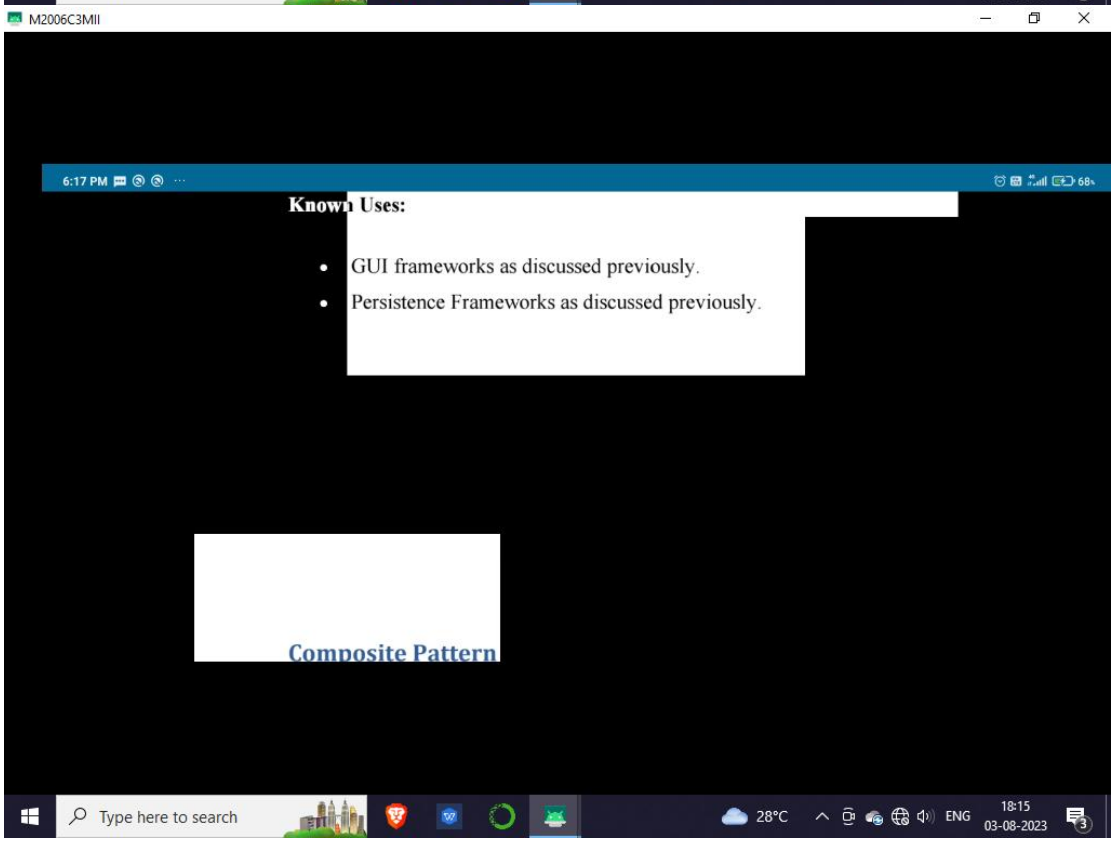
Known Uses:

- Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured and even switched at run-time.

Type here to search

28°C ENG 18:15 03-08-2023


M2006C3MII
6:17 PM
Known Uses:
Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured and even switched at run-time.
Abstraction and Implementer hierarchies can be extended independently.
Known Uses:
GUI frameworks as discussed previously.
Persistence Frameworks as discussed previously.
Type here to search
28°C
ENG
18:15
03-08-2023


M2006C3MII
6:17 PM
Known Uses:
GUI frameworks as discussed previously.
Persistence Frameworks as discussed previously.
Composite Pattern
Type here to search
28°C
ENG
18:15
03-08-2023

M2006C3MII

6:17 PM

Composite Pattern

Definition:

Description: Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part - whole hierarchies. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class contains group of its own objects. This class provides ways to modify its group of same objects.

Motivation

There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system. A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files. Note that a folder object usually contains

Type here to search 28°C ENG 18:15 03-08-2023

M2006C3MII

6:17 PM

Motivation

There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. Consider for example a program that manipulates a file system. A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files. Note that a folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object.

Note also that since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, it would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface.

Intent

- The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Type here to search 28°C ENG 18:15 03-08-2023

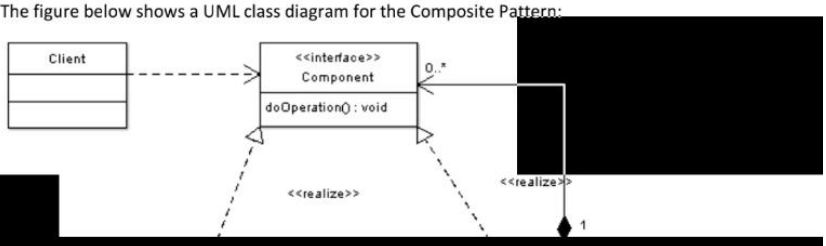
M2006C3MII

Intent

- The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Implementation

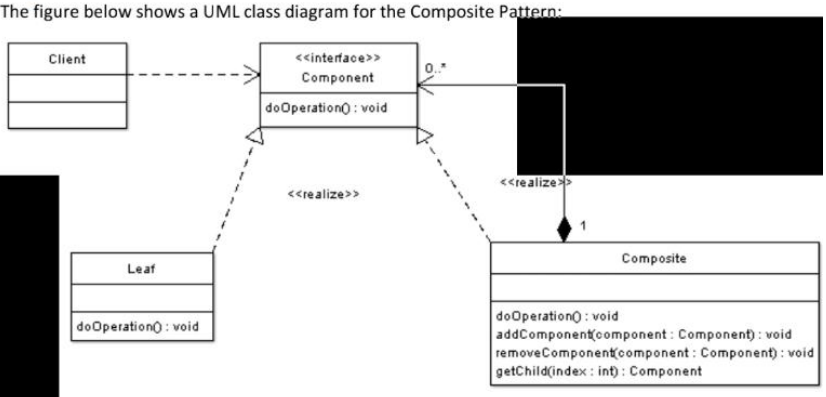
The figure below shows a UML class diagram for the Composite Pattern:



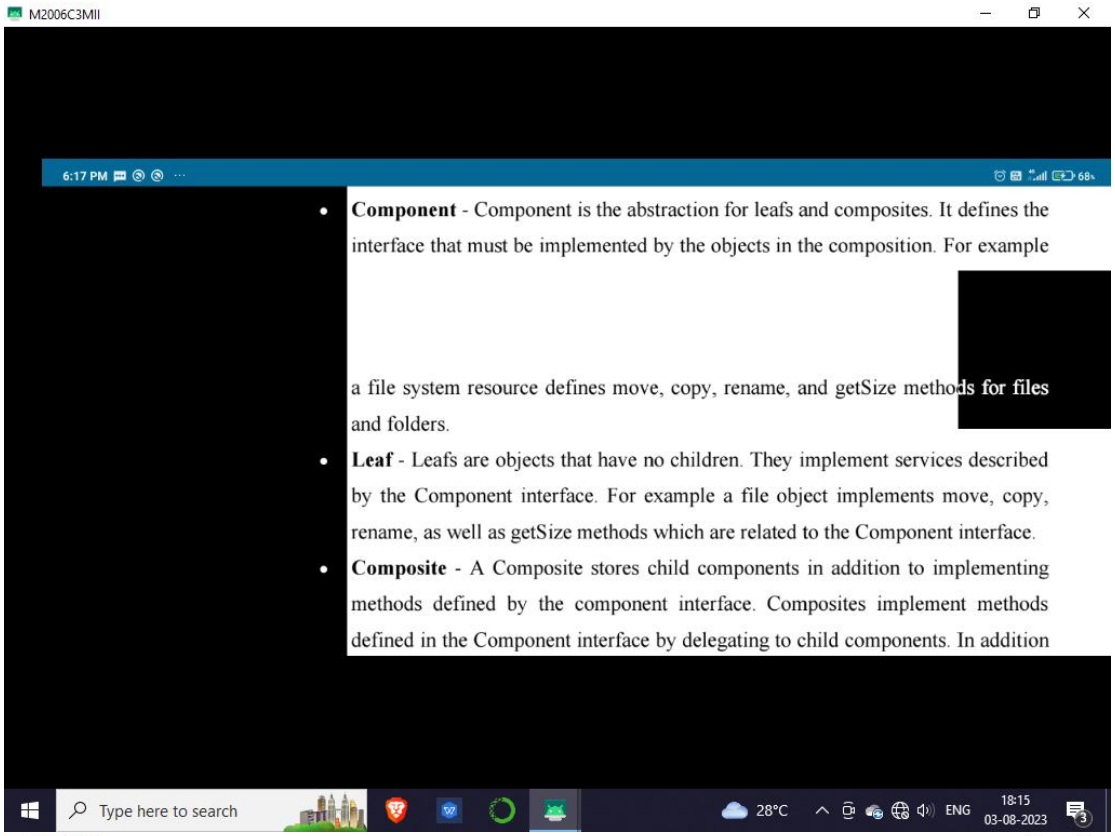
M2006C3MII

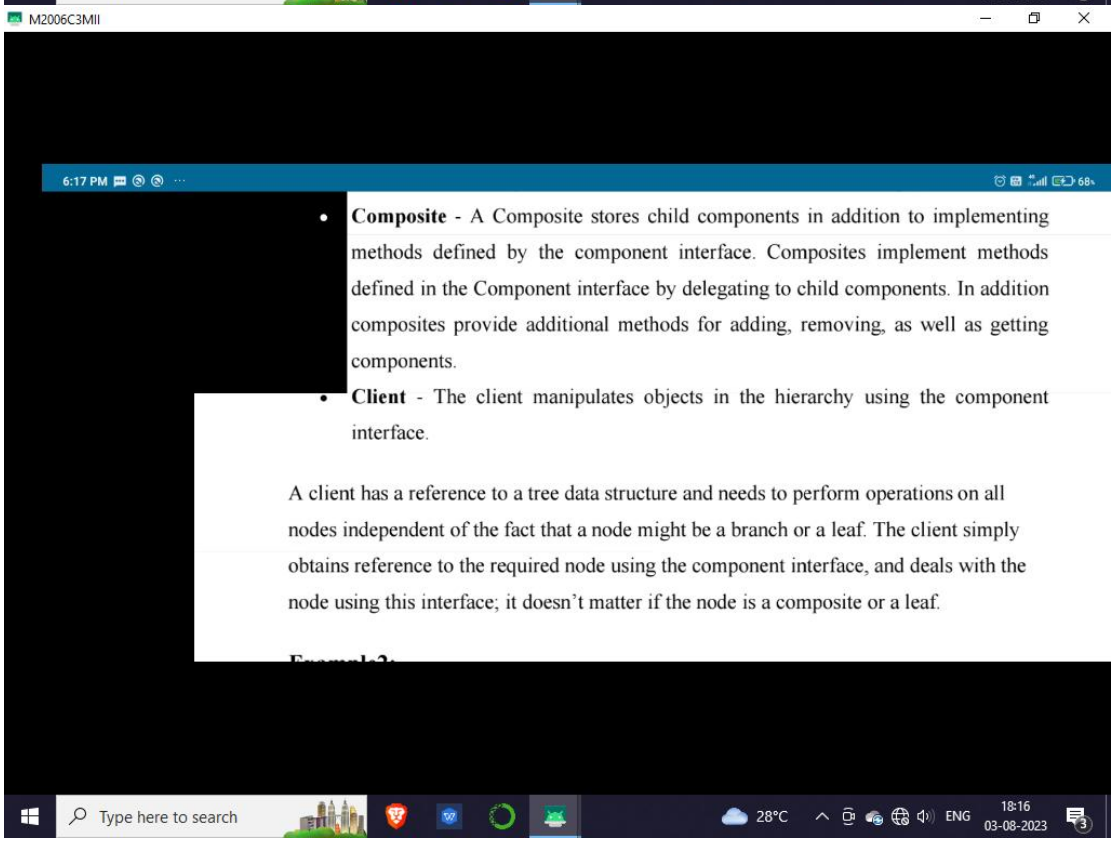
Implementation

The figure below shows a UML class diagram for the Composite Pattern:



- **Component** - Component is the abstraction for leafs and composites. It defines the


M2006C3MII
6:17 PM
Component - Component is the abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition. For example
a file system resource defines move, copy, rename, and getSize methods for files and folders.
Leaf - Leafs are objects that have no children. They implement services described by the Component interface. For example a file object implements move, copy, rename, as well as getSize methods which are related to the Component interface.
Composite - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components. In addition
Type here to search
28°C
ENG
18:15
03-08-2023


M2006C3MII
6:17 PM
Composite - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components. In addition composites provide additional methods for adding, removing, as well as getting components.
Client - The client manipulates objects in the hierarchy using the component interface.
A client has a reference to a tree data structure and needs to perform operations on all nodes independent of the fact that a node might be a branch or a leaf. The client simply obtains reference to the required node using the component interface, and deals with the node using this interface; it doesn't matter if the node is a composite or a leaf.
Type here to search
28°C
ENG
18:16
03-08-2023

M2006C3MII

6:18 PM

A client has a reference to a tree data structure and needs to perform operations on all nodes independent of the fact that a node might be a branch or a leaf. The client simply obtains reference to the required node using the component interface, and deals with the node using this interface; it doesn't matter if the node is a composite or a leaf.

Example2:

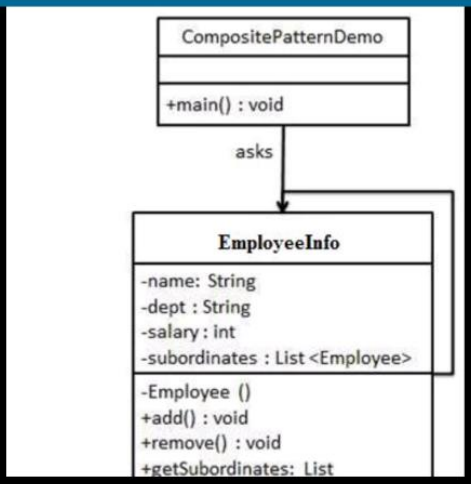
In this example we have a class *EmployeeInfo* which acts as composite pattern actor class. we will use *EmployeeInfo* class to add department level hierarchy and print all employees information.

Class Diagram:

Type here to search 28°C ENG 18:16 03-08-2023

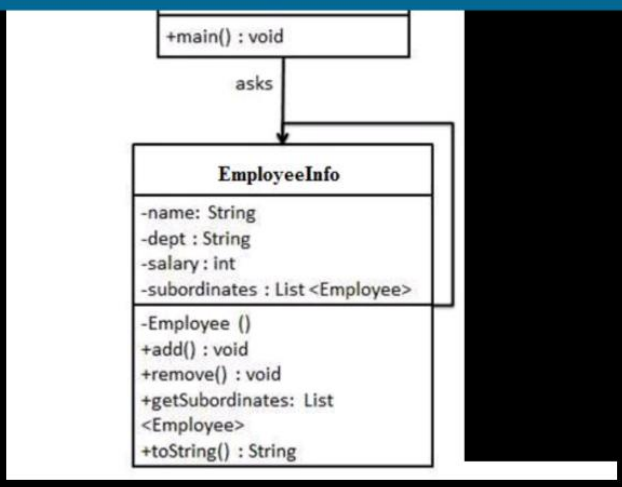
M2006C3MII

6:18 PM



Type here to search 28°C ENG 18:16 03-08-2023

M2006C3MII



M2006C3MII

Steps

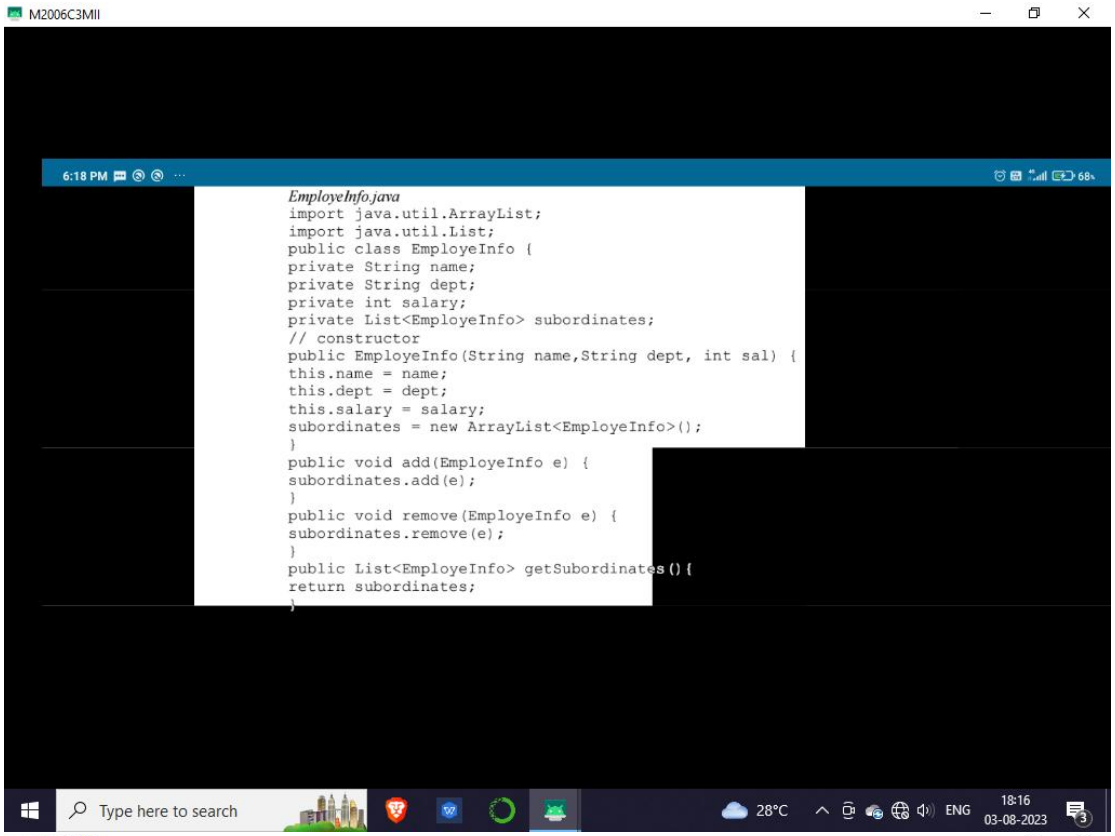
Use the following steps to implement the above mentioned design pattern.

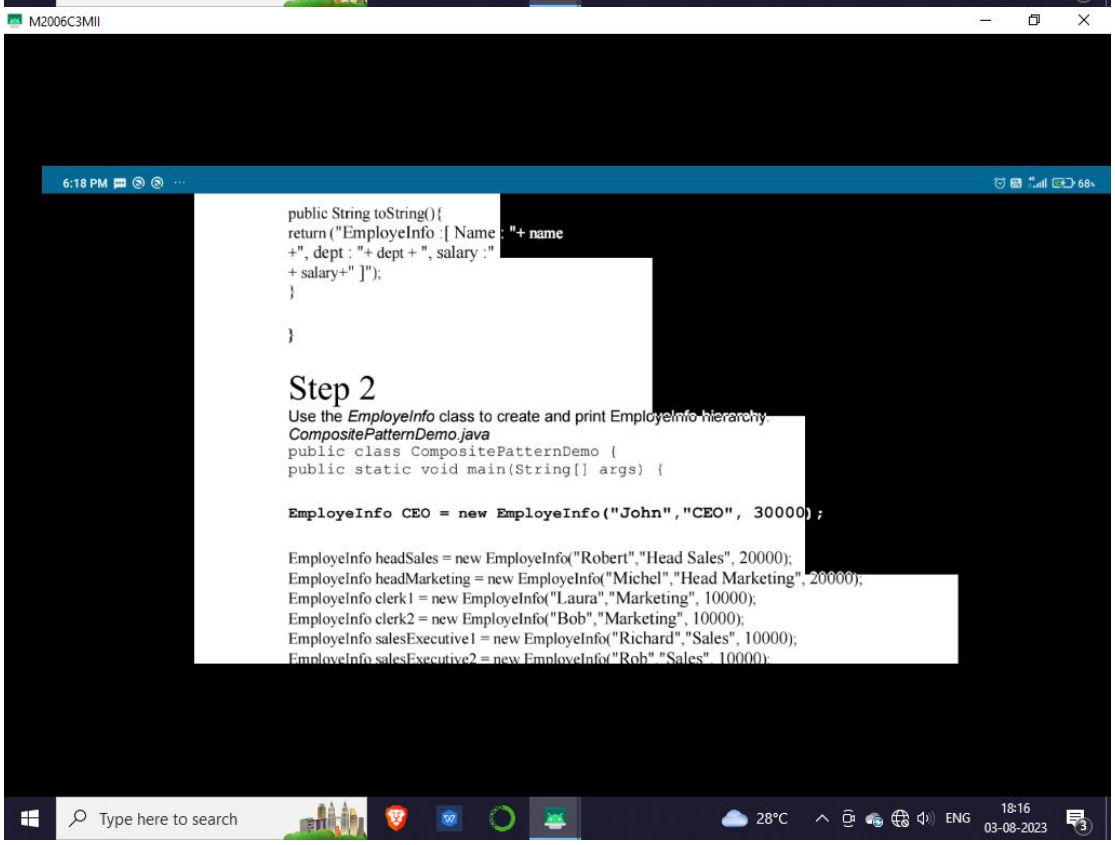
Step 1

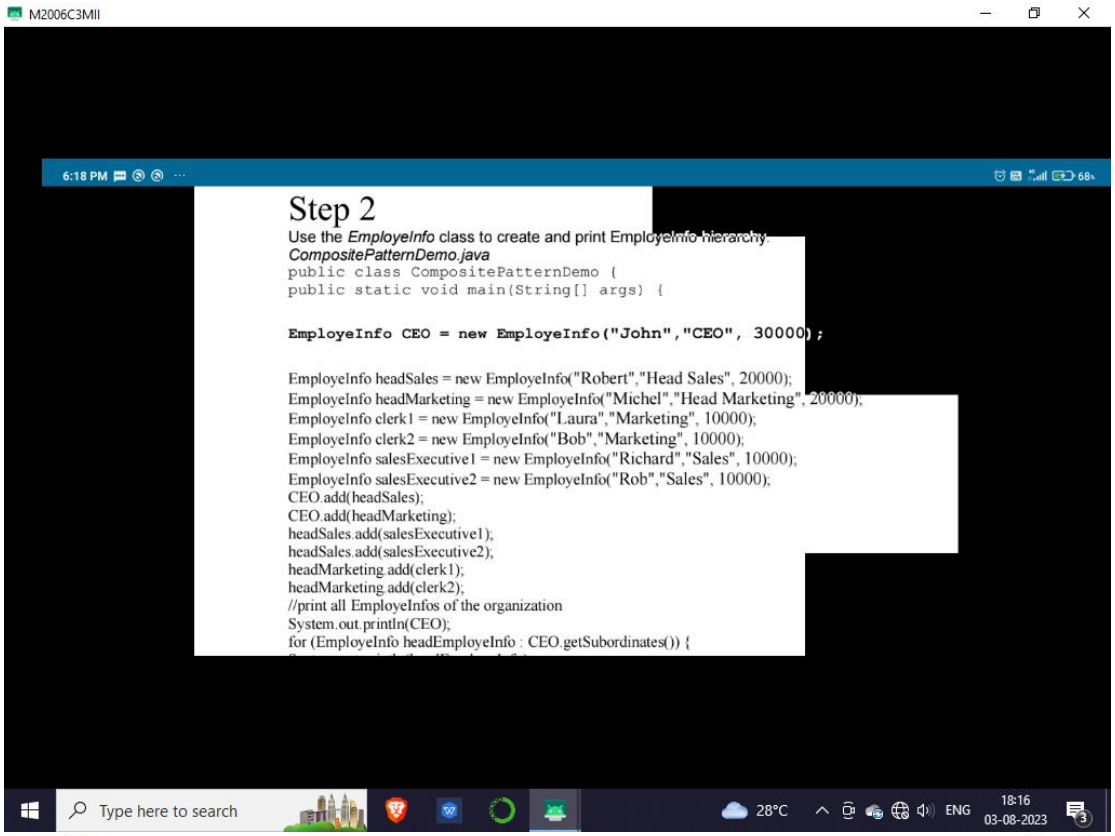
Create *Employee* class having list of *EmployeInfo* objects.

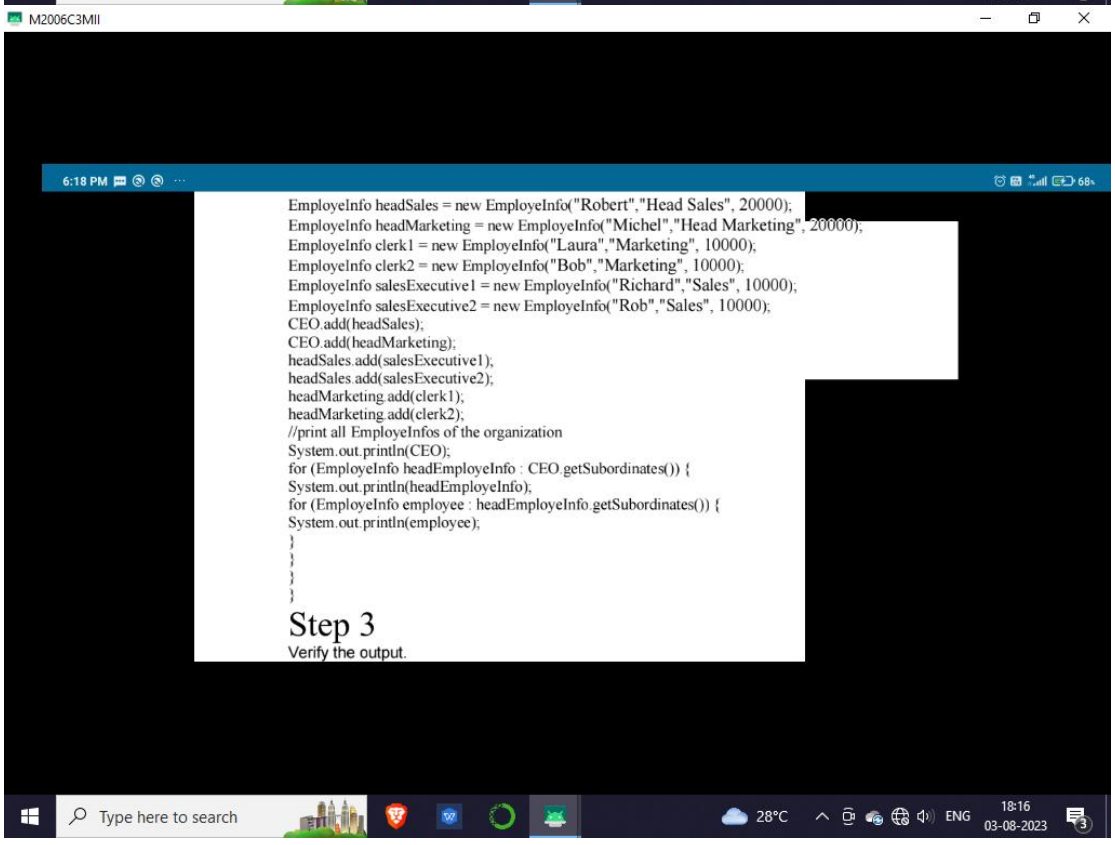
EmployeInfo.java

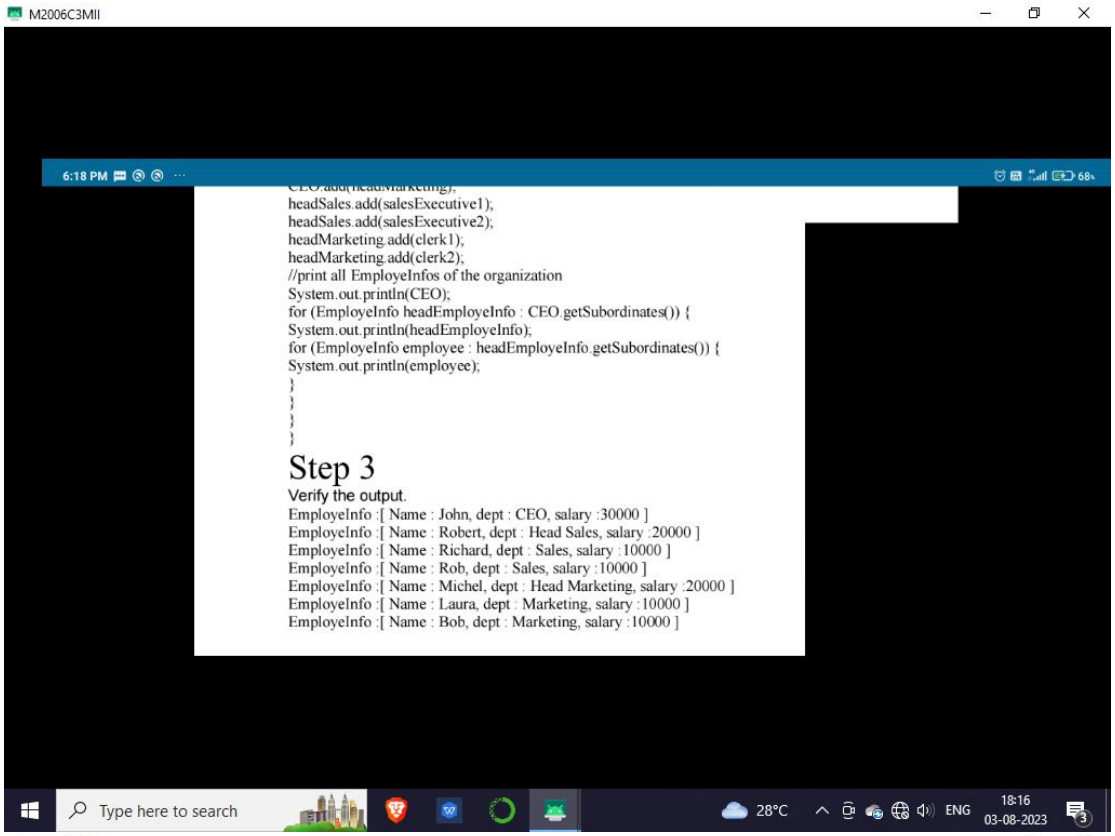
```
import java.util.ArrayList;
import java.util.List;
public class EmployeInfo {
private String name;
private String dept;
private int salary;
private List<EmployeInfo> subordinates;
// constructor
public EmployeInfo(String name,String dept, int sal) {
this.name = name;
this.dept = dept;
this.salary = salary;
subordinates = new ArrayList<EmployeInfo>();
}
public void add(EmployeInfo e) {
subordinates.add(e);
```

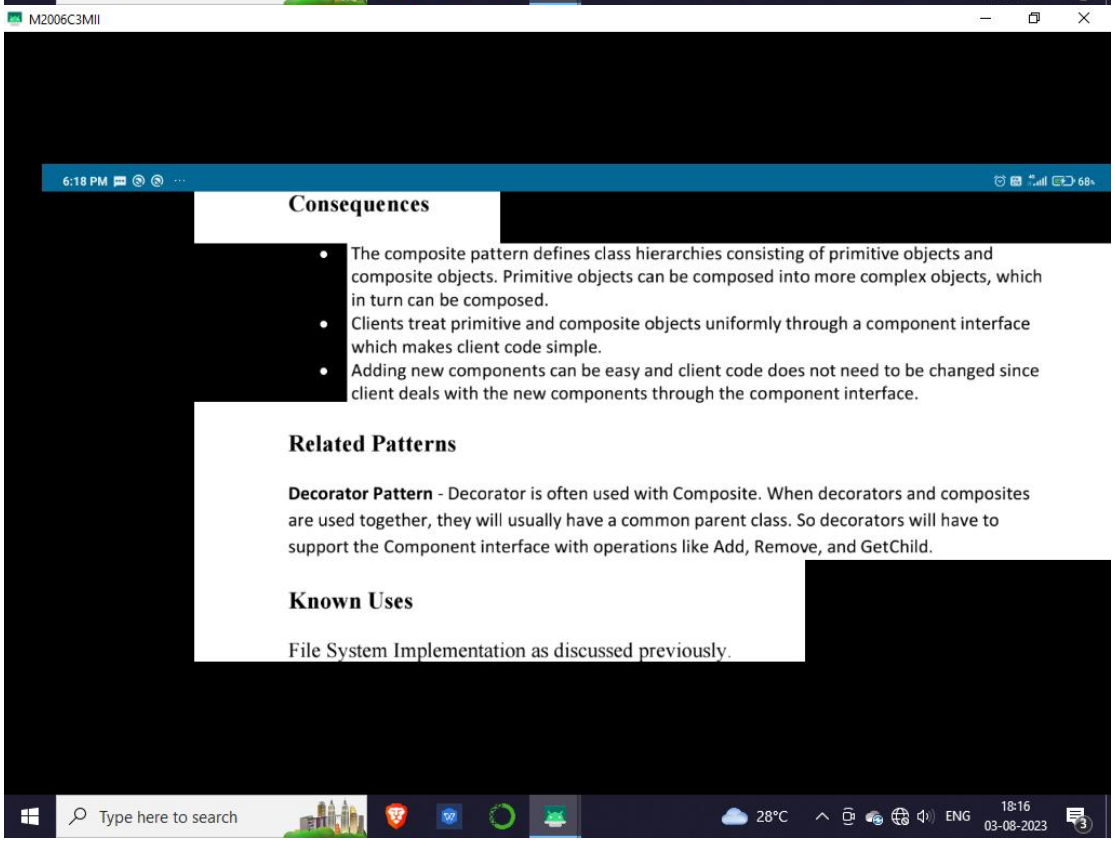

M2006C3MII
6:18 PM
EmployeeInfo.java
import java.util.ArrayList;
import java.util.List;
public class EmployeeInfo {
private String name;
private String dept;
private int salary;
private List<EmployeeInfo> subordinates;
// constructor
public EmployeeInfo(String name,String dept, int sal) {
this.name = name;
this.dept = dept;
this.salary = salary;
subordinates = new ArrayList<EmployeeInfo>();
}
public void add(EmployeeInfo e) {
subordinates.add(e);
}
public void remove(EmployeeInfo e) {
subordinates.remove(e);
}
public List<EmployeeInfo> getSubordinates () {
return subordinates;
}
Type here to search
28°C
ENG
18:16
03-08-2023


M2006C3MII
6:18 PM
public String toString(){
return ("EmployeInfo :[Name : "+ name
+", dept : "+ dept + ", salary :"
+ salary+"]");
}
}
Step 2
Use the EmployeInfo class to create and print EmployeInfo hierarchy.
CompositePatternDemo.java
public class CompositePatternDemo {
public static void main(String[] args) {
EmployeInfo CEO = new EmployeInfo("John","CEO", 30000);
EmployeInfo headSales = new EmployeInfo("Robert","Head Sales", 20000);
EmployeInfo headMarketing = new EmployeInfo("Michel","Head Marketing", 20000);
EmployeInfo clerk1 = new EmployeInfo("Laura","Marketing", 10000);
EmployeInfo clerk2 = new EmployeInfo("Bob","Marketing", 10000);
EmployeInfo salesExecutive1 = new EmployeInfo("Richard","Sales", 10000);
EmployeInfo salesExecutive2 = new EmployeInfo("Rob","Sales", 10000);
Type here to search
28°C
ENG
18:16
03-08-2023


M2006C3MII
6:18 PM
Step 2
Use the EmployeeInfo class to create and print EmployeeInfo hierarchy.
CompositePatternDemo.java
public class CompositePatternDemo {
public static void main(String[] args) {
EmployeeInfo CEO = new EmployeeInfo("John","CEO", 30000);
EmployeeInfo headSales = new EmployeeInfo("Robert","Head Sales", 20000);
EmployeeInfo headMarketing = new EmployeeInfo("Michel","Head Marketing", 20000);
EmployeeInfo clerk1 = new EmployeeInfo("Laura","Marketing", 10000);
EmployeeInfo clerk2 = new EmployeeInfo("Bob","Marketing", 10000);
EmployeeInfo salesExecutive1 = new EmployeeInfo("Richard","Sales", 10000);
EmployeeInfo salesExecutive2 = new EmployeeInfo("Rob","Sales", 10000);
CEO.add(headSales);
CEO.add(headMarketing);
headSales.add(salesExecutive1);
headSales.add(salesExecutive2);
headMarketing.add(clerk1);
headMarketing.add(clerk2);
//print all EmployeeInfos of the organization
System.out.println(CEO);
for (EmployeeInfo headEmployeeInfo : CEO.getSubordinates()) {
Type here to search
28°C
ENG
18:16
03-08-2023


M2006C3MII
6:18 PM
EmployeeInfo headSales = new EmployeeInfo("Robert","Head Sales", 20000);
EmployeeInfo headMarketing = new EmployeeInfo("Michel","Head Marketing", 20000);
EmployeeInfo clerk1 = new EmployeeInfo("Laura","Marketing", 10000);
EmployeeInfo clerk2 = new EmployeeInfo("Bob","Marketing", 10000);
EmployeeInfo salesExecutive1 = new EmployeeInfo("Richard","Sales", 10000);
EmployeeInfo salesExecutive2 = new EmployeeInfo("Rob","Sales", 10000);
CEO.add(headSales);
CEO.add(headMarketing);
headSales.add(salesExecutive1);
headSales.add(salesExecutive2);
headMarketing.add(clerk1);
headMarketing.add(clerk2);
//print all EmployeeInfos of the organization
System.out.println(CEO);
for (EmployeeInfo headEmployeeInfo : CEO.getSubordinates()) {
System.out.println(headEmployeeInfo);
for (EmployeeInfo employee : headEmployeeInfo.getSubordinates()) {
System.out.println(employee);
}
}
}
}
Step 3
Verify the output.
Type here to search
28°C
ENG
18:16
03-08-2023


M2006C3MII
6:18 PM
headSales.add(salesExecutive1);
headSales.add(salesExecutive2);
headMarketing.add(clerk1);
headMarketing.add(clerk2);
//print all EmployeInfos of the organization
System.out.println(CEO);
for (EmployeInfo headEmployeInfo : CEO.getSubordinates()) {
System.out.println(headEmployeInfo);
for (EmployeInfo employee : headEmployeInfo.getSubordinates()) {
System.out.println(employee);
}
}
}
}
Step 3
Verify the output.
EmployeInfo :[Name : John, dept : CEO, salary :30000]
EmployeInfo :[Name : Robert, dept : Head Sales, salary :20000]
EmployeInfo :[Name : Richard, dept : Sales, salary :10000]
EmployeInfo :[Name : Rob, dept : Sales, salary :10000]
EmployeInfo :[Name : Michel, dept : Head Marketing, salary :20000]
EmployeInfo :[Name : Laura, dept : Marketing, salary :10000]
EmployeInfo :[Name : Bob, dept : Marketing, salary :10000]
Type here to search
28°C
ENG
18:16
03-08-2023


M2006C3MII
6:18 PM
Consequences
The composite pattern defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed.
Clients treat primitive and composite objects uniformly through a component interface which makes client code simple.
Adding new components can be easy and client code does not need to be changed since client deals with the new components through the component interface.
Related Patterns
Decorator Pattern - Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
Known Uses
File System Implementation as discussed previously.
Type here to search
28°C
ENG
18:16
03-08-2023

M2006C3MII

6:18 PM

Related Patterns

Decorator Pattern - Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Known Uses

File System Implementation as discussed previously.

Graphics Editors as discussed previously.

Type here to search

28°C ENG 18:17 03-08-2023

M2006C3MII

6:19 PM

Decorator Pattern

Motivation

Extending an objects functionality can be done statically (at compile time) by using inheritance however it might be necessary to extend an objects functionality dynamically (at runtime) as an object is used.

Consider the typical example of a graphical window. To extend the functionality of the graphical window for example by adding a frame to the window, would require extending the window class to create a FramedWindow class. To create a framed window it is necessary to create an object of the FramedWindow class. However it would be impossible to start with a plain window and to extend its functionality at runtime to become a framed window.

Type here to search

28°C ENG 18:17 03-08-2023

M2006C3MII

6:19 PM

Intent

- The intent of this pattern is to add additional responsibilities dynamically to an object.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable. *BlueShapeDecorator* is concrete class implementing *ShapeDecorator*. *DecoratorPatternDemo*, our demo class will use *BlueShapeDecorator* to decorate *Shape* objects
The figure below shows a UML class diagram for the Decorator Pattern:

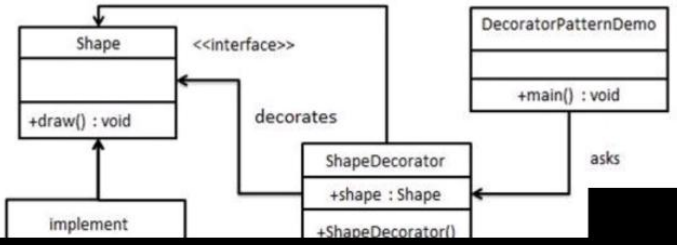
Type here to search 28°C ENG 18:17 03-08-2023

M2006C3MII

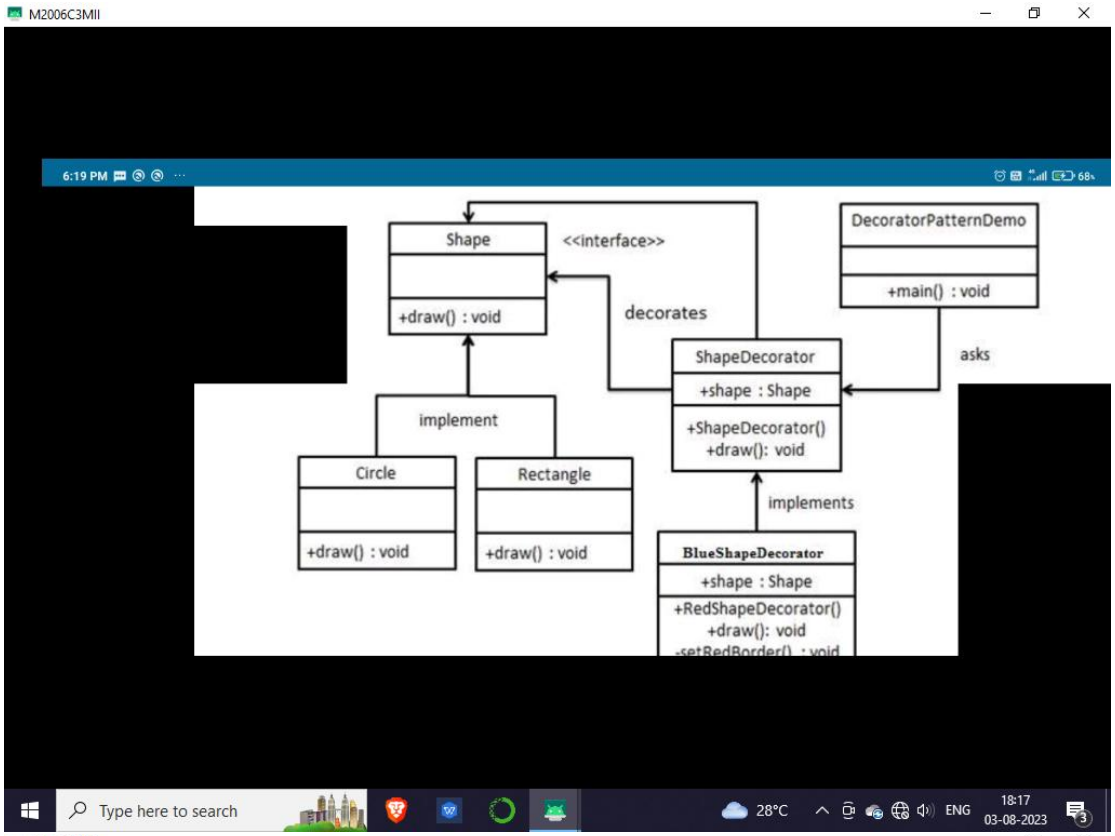
6:19 PM

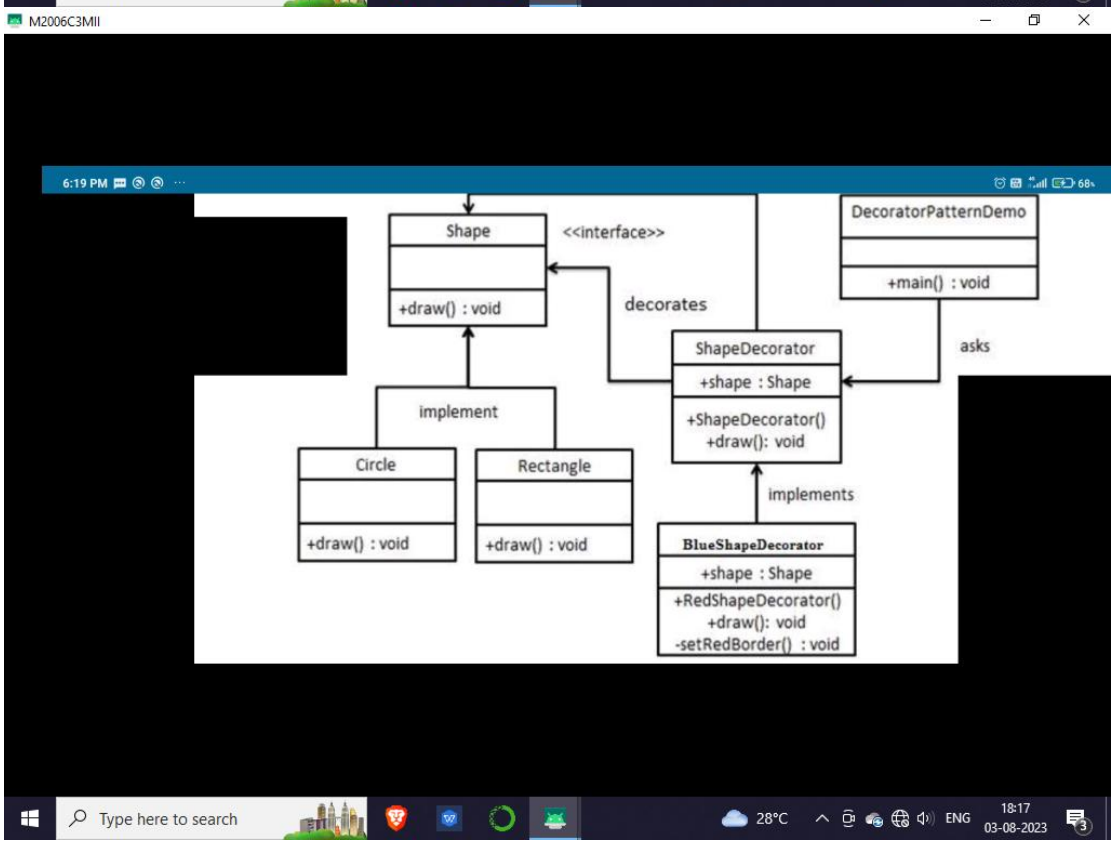
Implementation

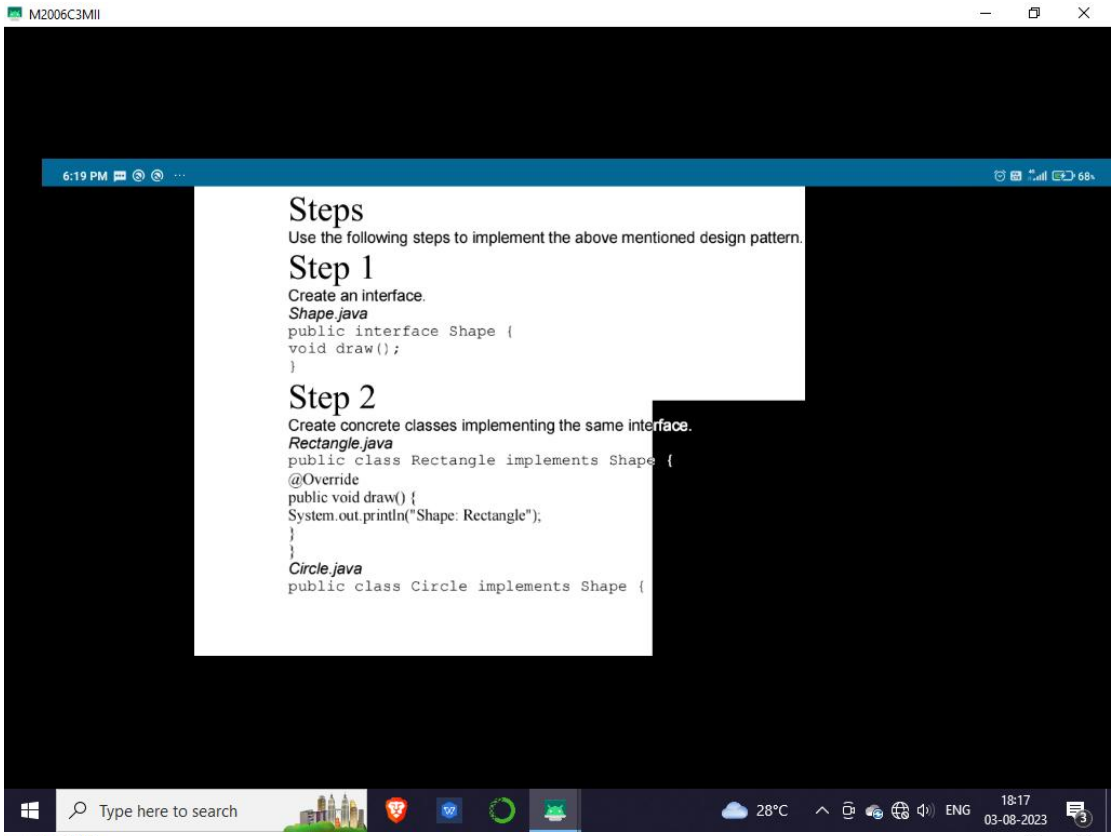
We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. We then create an abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable. *BlueShapeDecorator* is concrete class implementing *ShapeDecorator*. *DecoratorPatternDemo*, our demo class will use *BlueShapeDecorator* to decorate *Shape* objects
The figure below shows a UML class diagram for the Decorator Pattern:

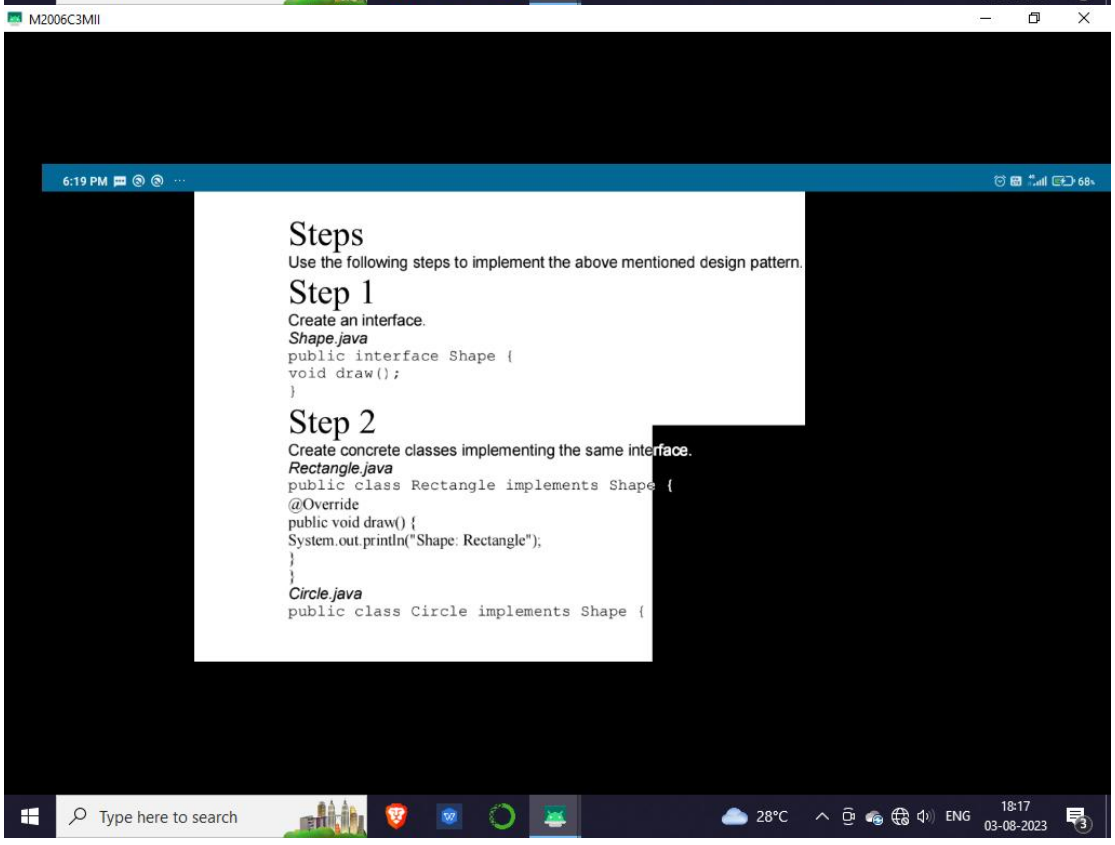


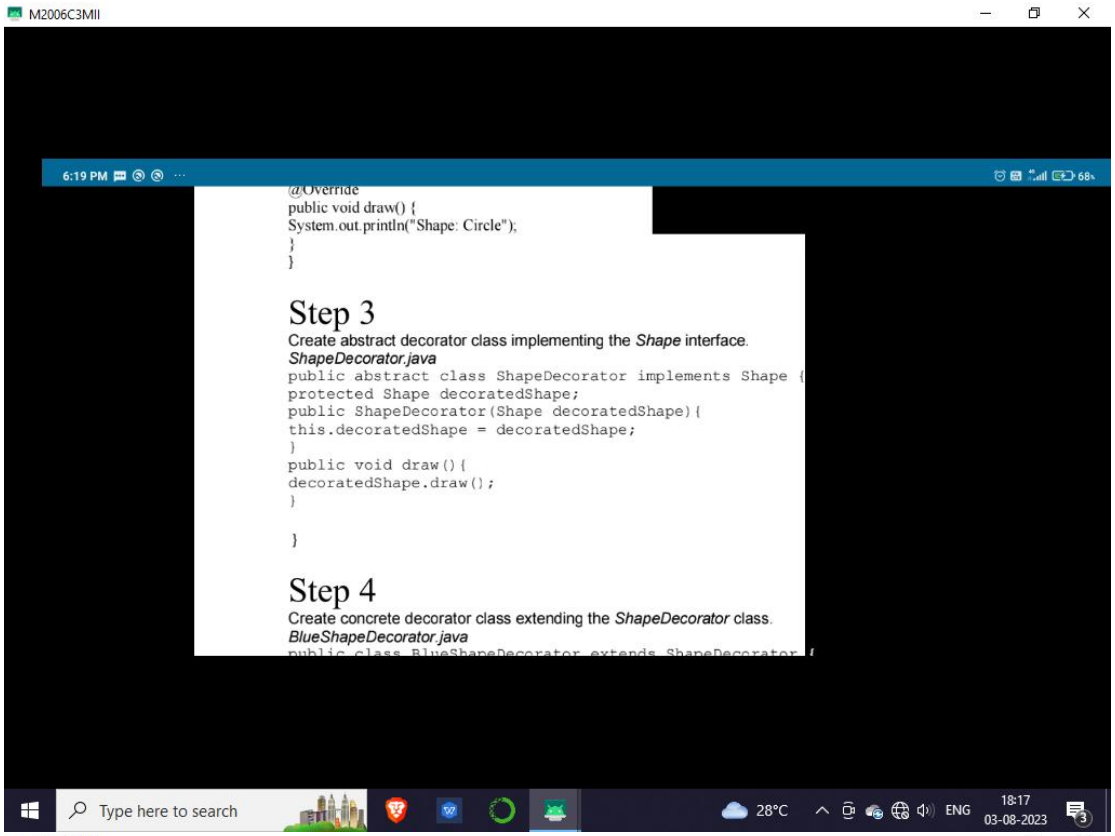
Type here to search 28°C ENG 18:17 03-08-2023

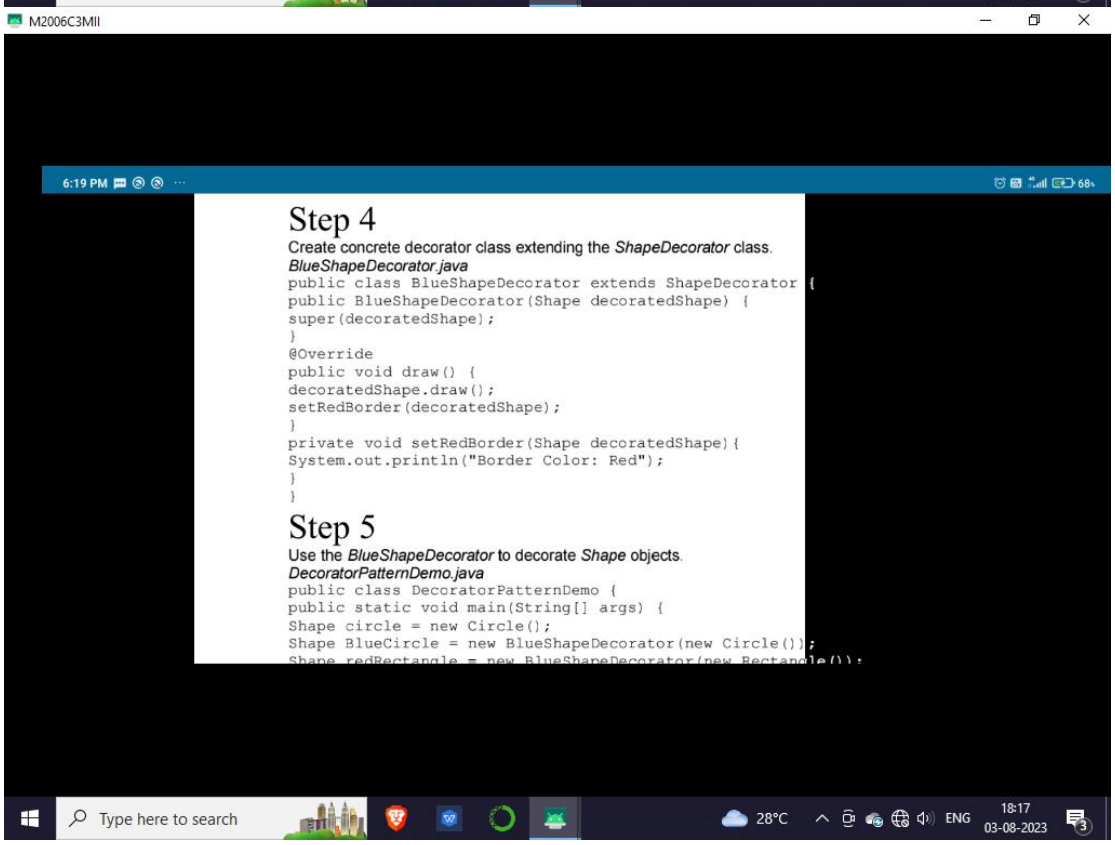

M2006C3MII
6:19 PM
Shape
<<interface>>
+draw() : void
decorates
DecoratorPatternDemo
+main() : void
asks
ShapeDecorator
+shape : Shape
+ShapeDecorator()
+draw(): void
implement
Circle
+draw() : void
Rectangle
+draw() : void
implements
BlueShapeDecorator
+shape : Shape
+RedShapeDecorator()
+draw(): void
Type here to search
28°C
ENG
18:17
03-08-2023

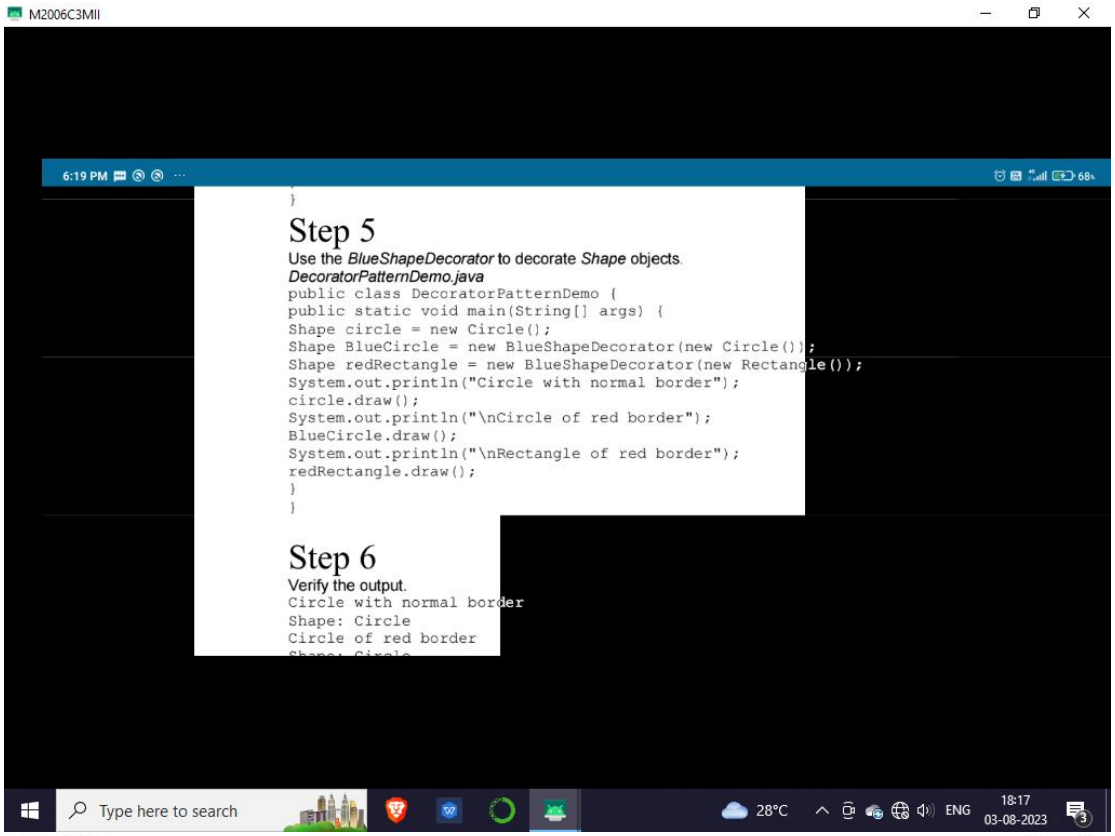

M2006C3MII
6:19 PM
Shape
<<interface>>
+draw() : void
decorates
DecoratorPatternDemo
+main() : void
asks
ShapeDecorator
+shape : Shape
+ShapeDecorator()
+draw(): void
implement
Circle
+draw() : void
Rectangle
+draw() : void
implements
BlueShapeDecorator
+shape : Shape
+RedShapeDecorator()
+draw(): void
-setRedBorder() : void
Type here to search
28°C
ENG
18:17
03-08-2023

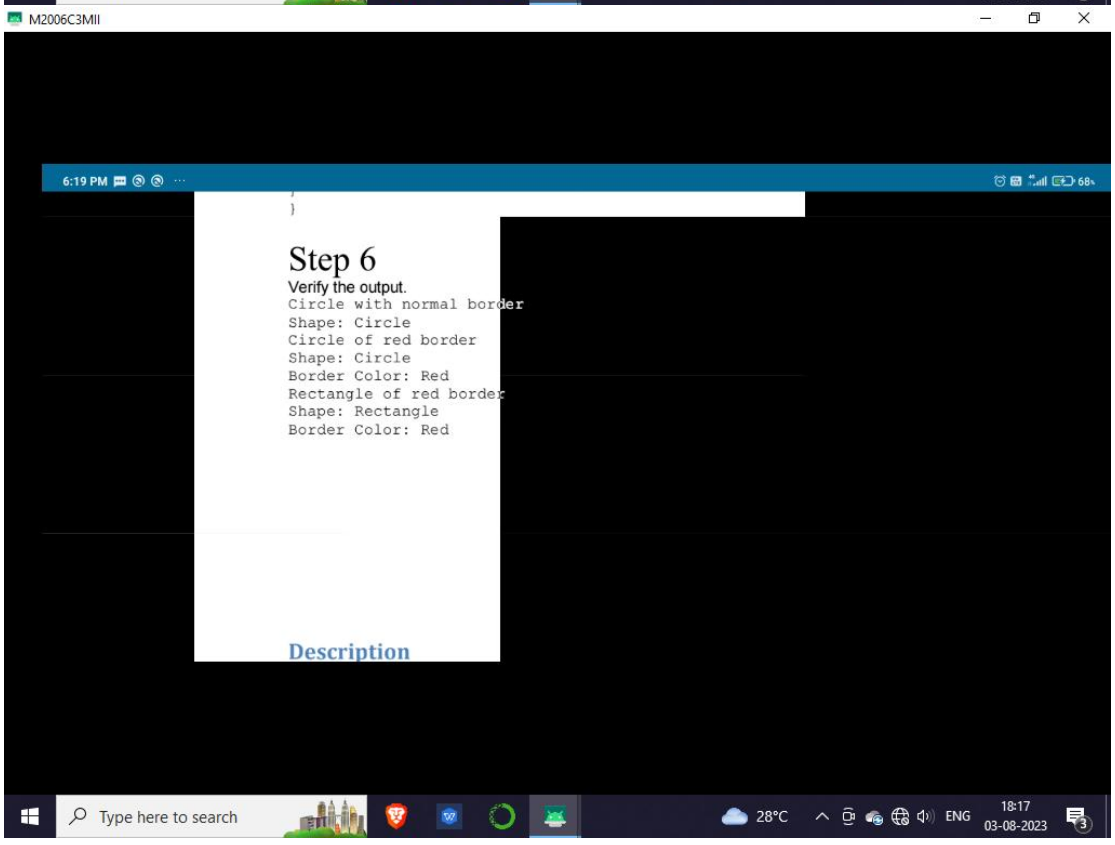

M2006C3MII
6:19 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create an interface.
Shape.java
public interface Shape {
void draw();
}
Step 2
Create concrete classes implementing the same interface.
Rectangle.java
public class Rectangle implements Shape {
@Override
public void draw() {
System.out.println("Shape: Rectangle");
}
}
Circle.java
public class Circle implements Shape {
Type here to search
28°C
ENG
18:17
03-08-2023

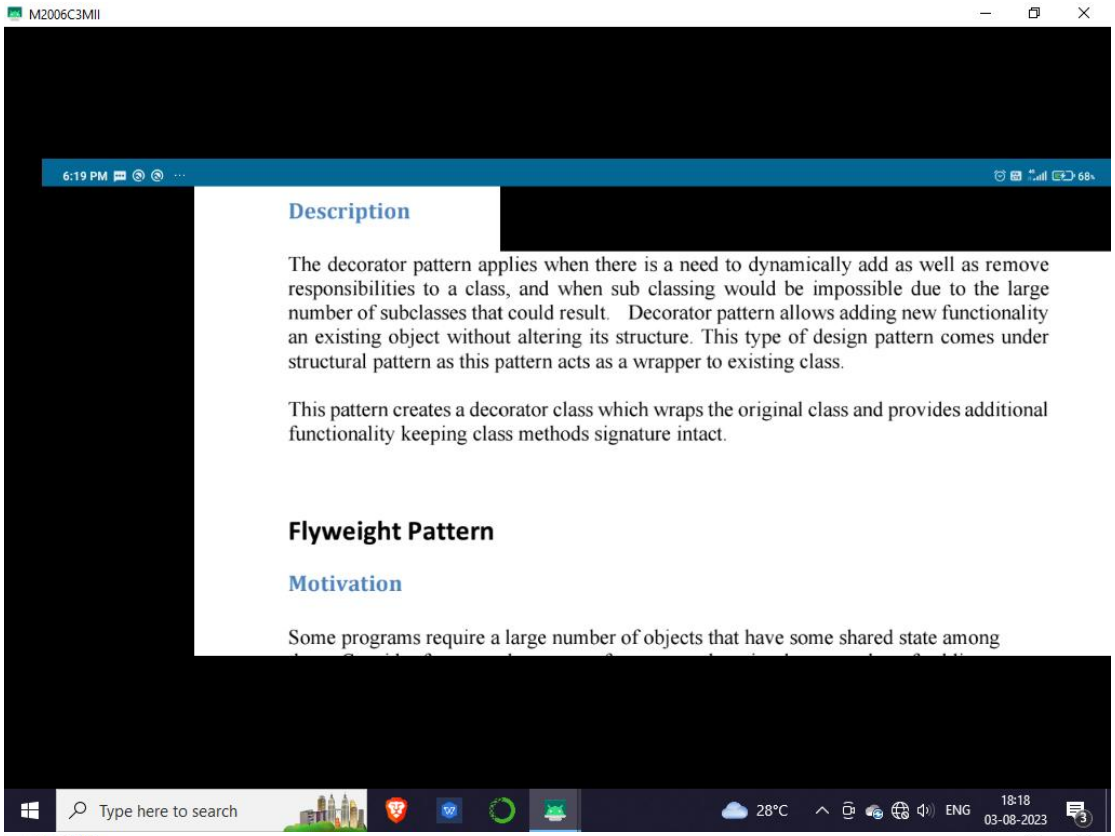

M2006C3MII
6:19 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create an interface.
Shape.java
public interface Shape {
void draw();
}
Step 2
Create concrete classes implementing the same interface.
Rectangle.java
public class Rectangle implements Shape {
@Override
public void draw() {
System.out.println("Shape: Rectangle");
}
}
Circle.java
public class Circle implements Shape {
Type here to search
28°C
ENG
18:17
03-08-2023

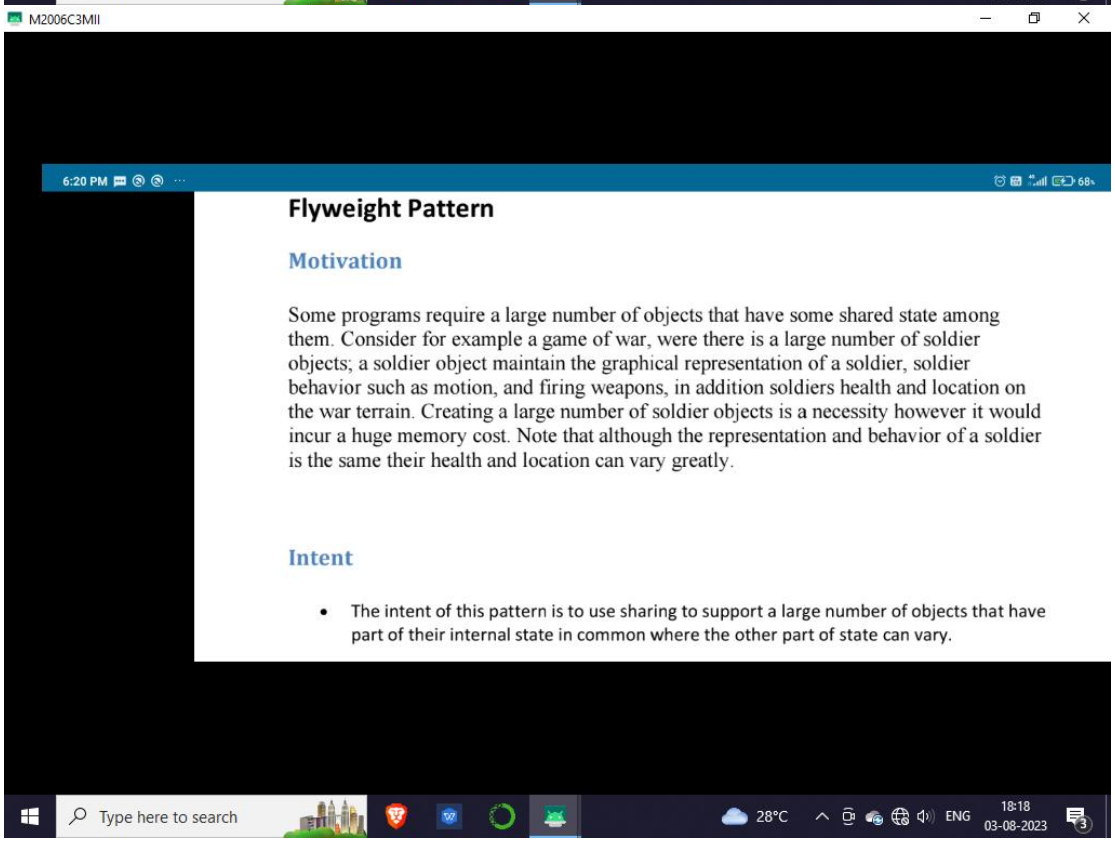

M2006C3MII
6:19 PM
@Override
public void draw() {
System.out.println("Shape: Circle");
}
}
Step 3
Create abstract decorator class implementing the Shape interface.
ShapeDecorator.java
public abstract class ShapeDecorator implements Shape {
protected Shape decoratedShape;
public ShapeDecorator(Shape decoratedShape){
this.decoratedShape = decoratedShape;
}
public void draw(){
decoratedShape.draw();
}
}
Step 4
Create concrete decorator class extending the ShapeDecorator class.
BlueShapeDecorator.java
Type here to search
28°C
ENG
18:17
03-08-2023


M2006C3MII
6:19 PM
Step 4
Create concrete decorator class extending the ShapeDecorator class.
BlueShapeDecorator.java
public class BlueShapeDecorator extends ShapeDecorator {
public BlueShapeDecorator(Shape decoratedShape) {
super(decoratedShape);
}
@Override
public void draw() {
decoratedShape.draw();
setRedBorder(decoratedShape);
}
private void setRedBorder(Shape decoratedShape){
System.out.println("Border Color: Red");
}
}
Step 5
Use the BlueShapeDecorator to decorate Shape objects.
DecoratorPatternDemo.java
public class DecoratorPatternDemo {
public static void main(String[] args) {
Shape circle = new Circle();
Shape BlueCircle = new BlueShapeDecorator(new Circle());
Type here to search
28°C
ENG
18:17
03-08-2023


M2006C3MII
6:19 PM
}
Step 5
Use the BlueShapeDecorator to decorate Shape objects.
DecoratorPatternDemo.java
public class DecoratorPatternDemo {
public static void main(String[] args) {
Shape circle = new Circle();
Shape BlueCircle = new BlueShapeDecorator(new Circle());
Shape redRectangle = new BlueShapeDecorator(new Rectangle());
System.out.println("Circle with normal border");
circle.draw();
System.out.println("\nCircle of red border");
BlueCircle.draw();
System.out.println("\nRectangle of red border");
redRectangle.draw();
}
}
Step 6
Verify the output.
Circle with normal border
Shape: Circle
Circle of red border
Type here to search
28°C
ENG
18:17
03-08-2023


M2006C3MII
6:19 PM
}
Step 6
Verify the output.
Circle with normal border
Shape: Circle
Circle of red border
Shape: Circle
Border Color: Red
Rectangle of red border
Shape: Rectangle
Border Color: Red
Description
Type here to search
28°C
ENG
18:17
03-08-2023


M2006C3MII
6:19 PM
Description
The decorator pattern applies when there is a need to dynamically add as well as remove responsibilities to a class, and when sub classing would be impossible due to the large number of subclasses that could result. Decorator pattern allows adding new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.
This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
Flyweight Pattern
Motivation
Some programs require a large number of objects that have some shared state among
Type here to search
28°C
ENG
18:18
03-08-2023


M2006C3MII
6:20 PM
Flyweight Pattern
Motivation
Some programs require a large number of objects that have some shared state among them. Consider for example a game of war, were there is a large number of soldier objects; a soldier object maintain the graphical representation of a soldier, soldier behavior such as motion, and firing weapons, in addition soldiers health and location on the war terrain. Creating a large number of soldier objects is a necessity however it would incur a huge memory cost. Note that although the representation and behavior of a soldier is the same their health and location can vary greatly.
Intent
• The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.
Type here to search
28°C
ENG
18:18
03-08-2023

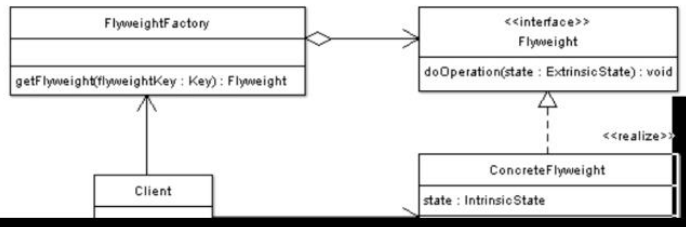
M2006C3MII

Intent

- The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

Implementation

The figure below shows a UML class diagram for the Flyweight Pattern:

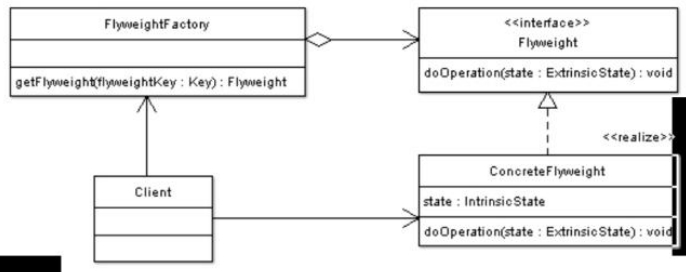


Type here to search 28°C ENG 18:18 03-08-2023

M2006C3MII

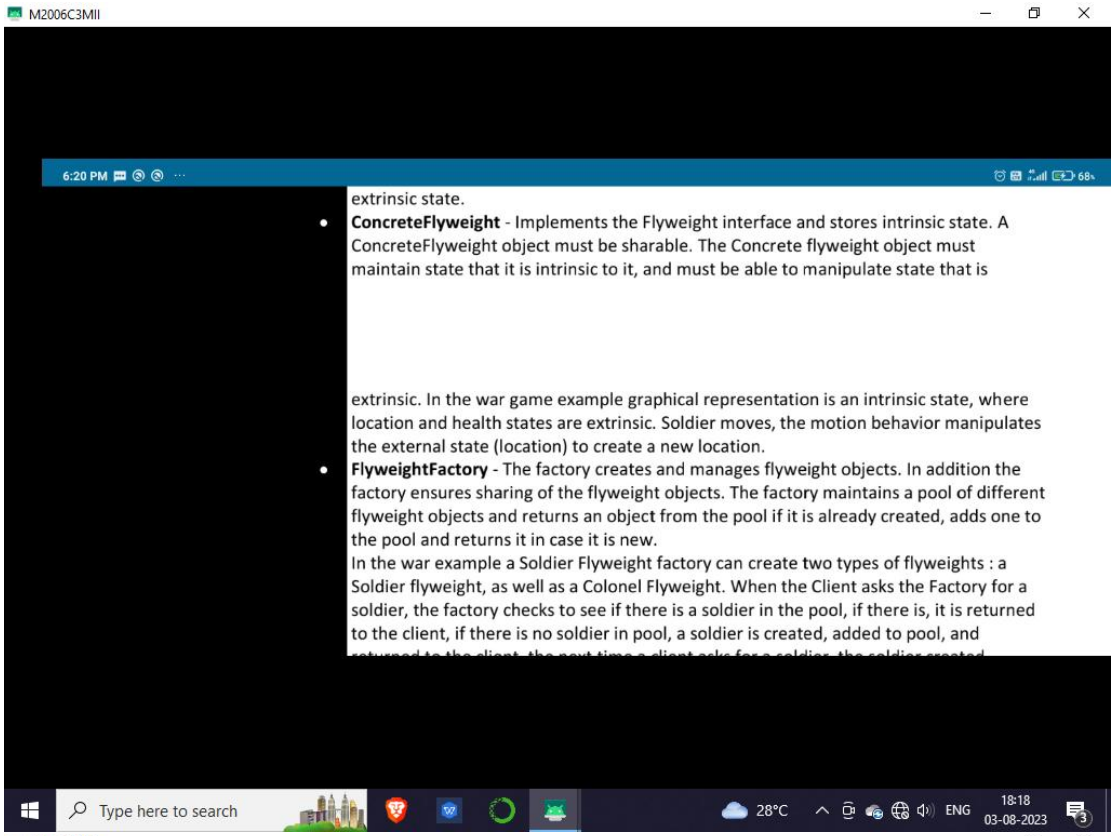
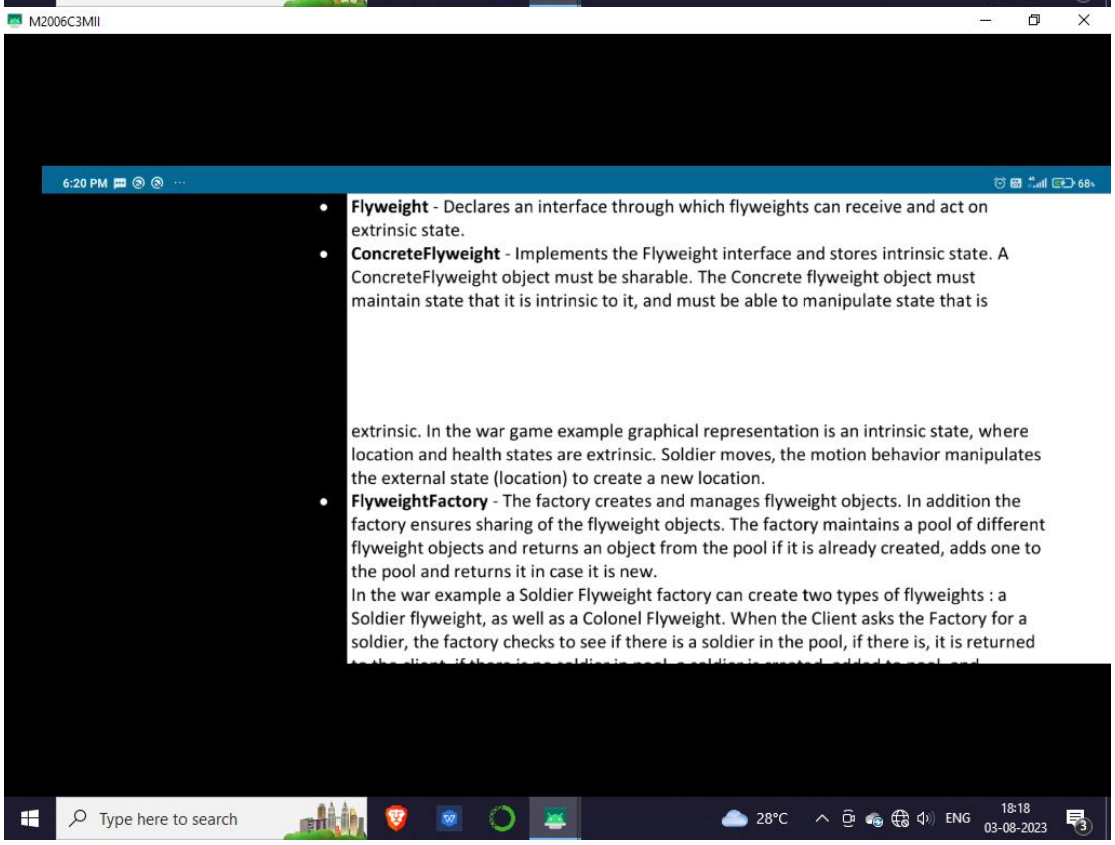
Implementation

The figure below shows a UML class diagram for the Flyweight Pattern:



- **Flyweight** - Declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight** - Implements the Flyweight interface and stores intrinsic state. A ConcreteFlyweight object must be sharable. The Concrete flyweight object must

Type here to search 28°C ENG 18:18 03-08-2023


M2006C3MII
6:20 PM
extrinsic state.
• **ConcreteFlyweight** - Implements the Flyweight interface and stores intrinsic state. A ConcreteFlyweight object must be sharable. The Concrete flyweight object must maintain state that it is intrinsic to it, and must be able to manipulate state that is
extrinsic. In the war game example graphical representation is an intrinsic state, where location and health states are extrinsic. Soldier moves, the motion behavior manipulates the external state (location) to create a new location.
• **FlyweightFactory** - The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.
In the war example a Soldier Flyweight factory can create two types of flyweights : a Soldier flyweight, as well as a Colonel Flyweight. When the Client asks the Factory for a soldier, the factory checks to see if there is a soldier in the pool, if there is, it is returned to the client, if there is no soldier in pool, a soldier is created, added to pool, and
Type here to search
28°C
ENG
18:18
03-08-2023

M2006C3MII
6:20 PM
• **Flyweight** - Declares an interface through which flyweights can receive and act on extrinsic state.
• **ConcreteFlyweight** - Implements the Flyweight interface and stores intrinsic state. A ConcreteFlyweight object must be sharable. The Concrete flyweight object must maintain state that it is intrinsic to it, and must be able to manipulate state that is
extrinsic. In the war game example graphical representation is an intrinsic state, where location and health states are extrinsic. Soldier moves, the motion behavior manipulates the external state (location) to create a new location.
• **FlyweightFactory** - The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.
In the war example a Soldier Flyweight factory can create two types of flyweights : a Soldier flyweight, as well as a Colonel Flyweight. When the Client asks the Factory for a soldier, the factory checks to see if there is a soldier in the pool, if there is, it is returned
Type here to search
28°C
ENG
18:18
03-08-2023

M2006C3MII

6:20 PM

- **FlyweightFactory** - The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.
  In the war example a Soldier Flyweight factory can create two types of flyweights : a Soldier flyweight, as well as a Colonel Flyweight. When the Client asks the Factory for a soldier, the factory checks to see if there is a soldier in the pool, if there is, it is returned to the client, if there is no soldier in pool, a soldier is created, added to pool, and returned to the client, the next time a client asks for a soldier, the soldier created previously is returned, no new soldier is created.
- **Client** - A client maintains references to flyweights in addition to computing and maintaining extrinsic state

A client needs a flyweight object; it calls the factory to get the flyweight object. The factory checks a pool of flyweights to determine if a flyweight object of the requested type is in the pool, if there is, the reference to that object is returned. If there is no object of the required type, the factory creates a flyweight of the requested type, adds it to the pool, and returns a reference to the flyweight. The flyweight maintains intrinsic state (state that is shared among the large number of objects that we have created the flyweight for) and provides methods to manipulate external state (State that vary from object to

Type here to search

28°C ENG 18:18 03-08-2023

M2006C3MII

6:20 PM

A client needs a flyweight object; it calls the factory to get the flyweight object. The factory checks a pool of flyweights to determine if a flyweight object of the requested type is in the pool, if there is, the reference to that object is returned. If there is no object of the required type, the factory creates a flyweight of the requested type, adds it to the pool, and returns a reference to the flyweight. The flyweight maintains intrinsic state (state that is shared among the large number of objects that we have created the flyweight for) and provides methods to manipulate external state (State that vary from object to object and is not common among the objects we have created the flyweight for).

Applicability & Examples:

In this example we're going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface. A factory class *ShapeFactoryInfo* is defined as a next step.
ShapeFactoryInfo Info have a *HashMap* of *Circle* having key as color of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactoryInfo*. *ShapeFactoryInfo* checks the circle object in its*HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use and returned to client.
FlyWeightPatternDemo, our demo class will use *ShapeFactoryInfo* to get a *Shape* object. It will pass information (*red / green / blue/ black / white*) to *ShapeFactoryInfo* to get the circle of desired color it needs.

Type here to search

28°C ENG 18:18 03-08-2023

M2006C3MII

6:20 PM

Applicability & Examples:

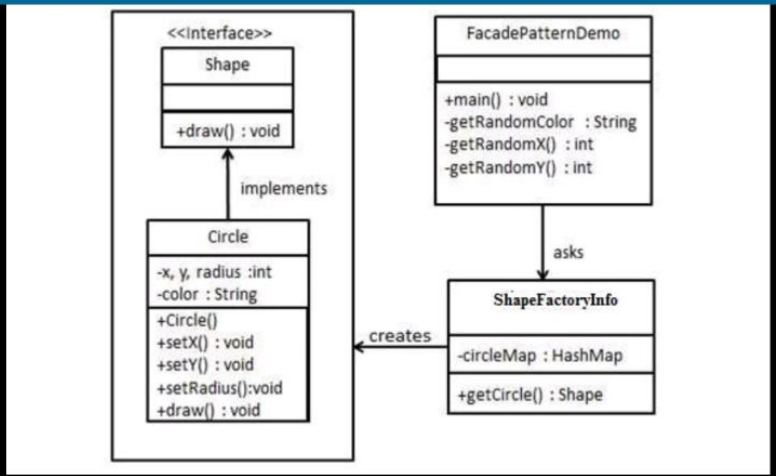
In this example we're going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface. A factory class *ShapeFactoryInfo* is defined as a next step.
ShapeFactoryInfo Info have a *HashMap* of *Circle* having key as color of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactoryInfo*. *ShapeFactoryInfo* checks the circle object in its*HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use and returned to client.
FlyWeightPatternDemo, our demo class will use *ShapeFactoryInfo* to get a *Shape* object. It will pass information (*red / green / blue/ black / white*) to *ShapeFactoryInfo* to get the circle of desired color it needs.

Class Diagram:

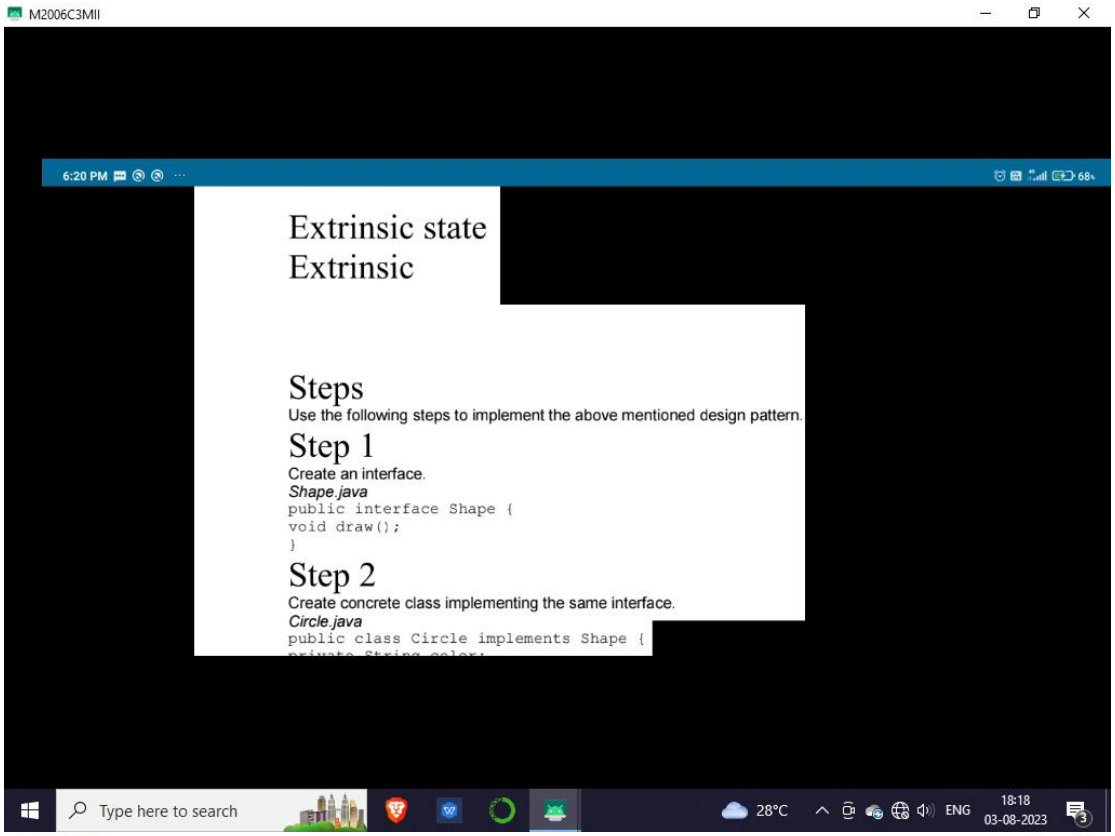
Type here to search 28°C ENG 18:18 03-08-2023

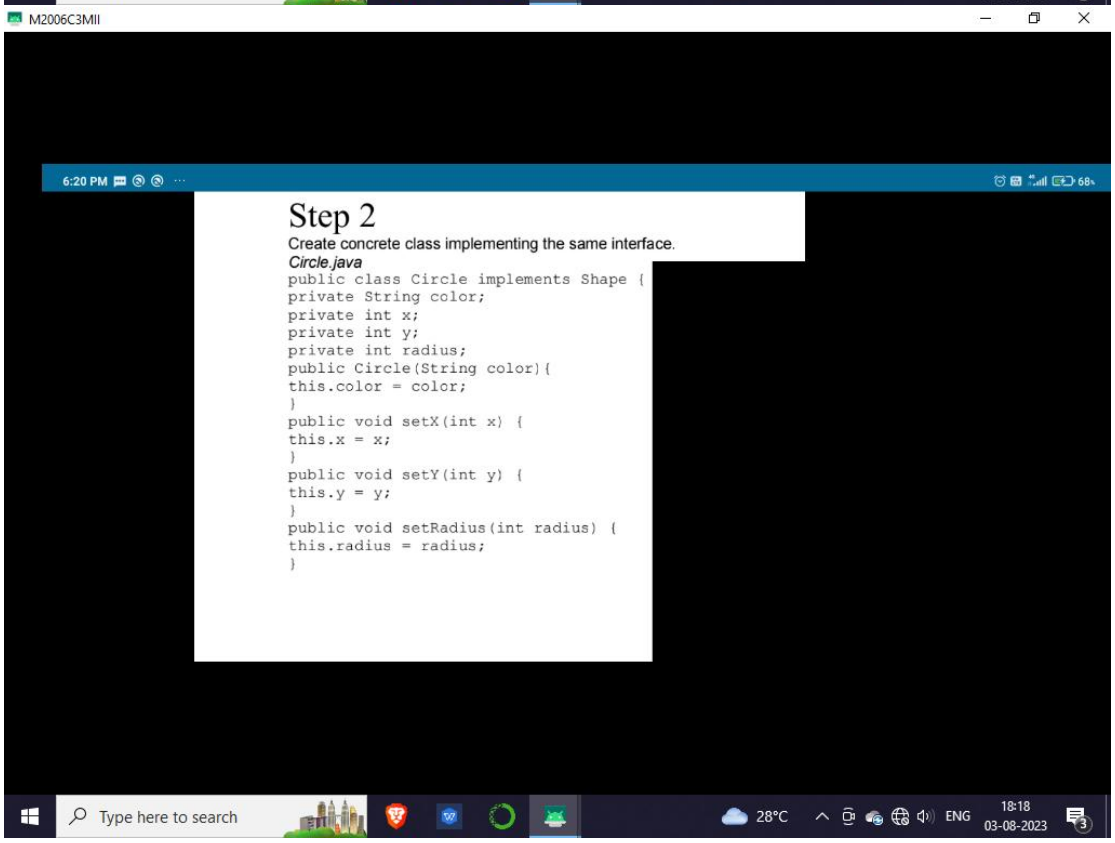
M2006C3MII

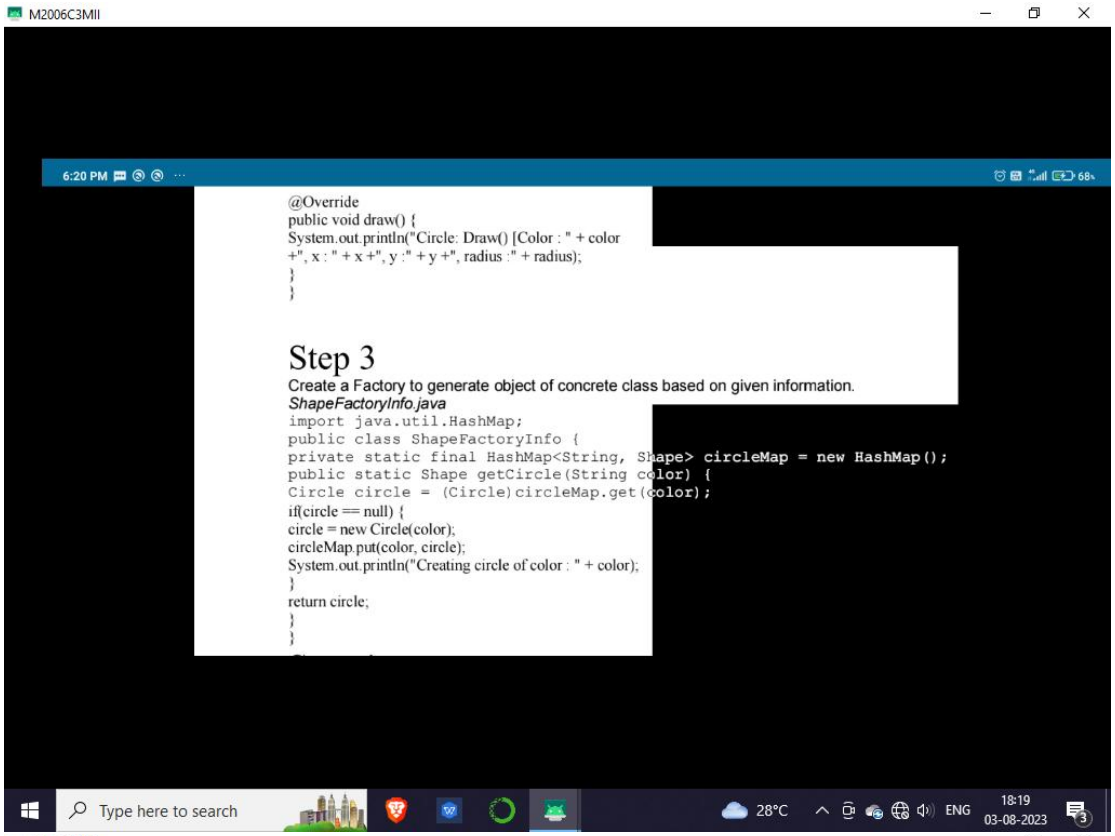
6:20 PM

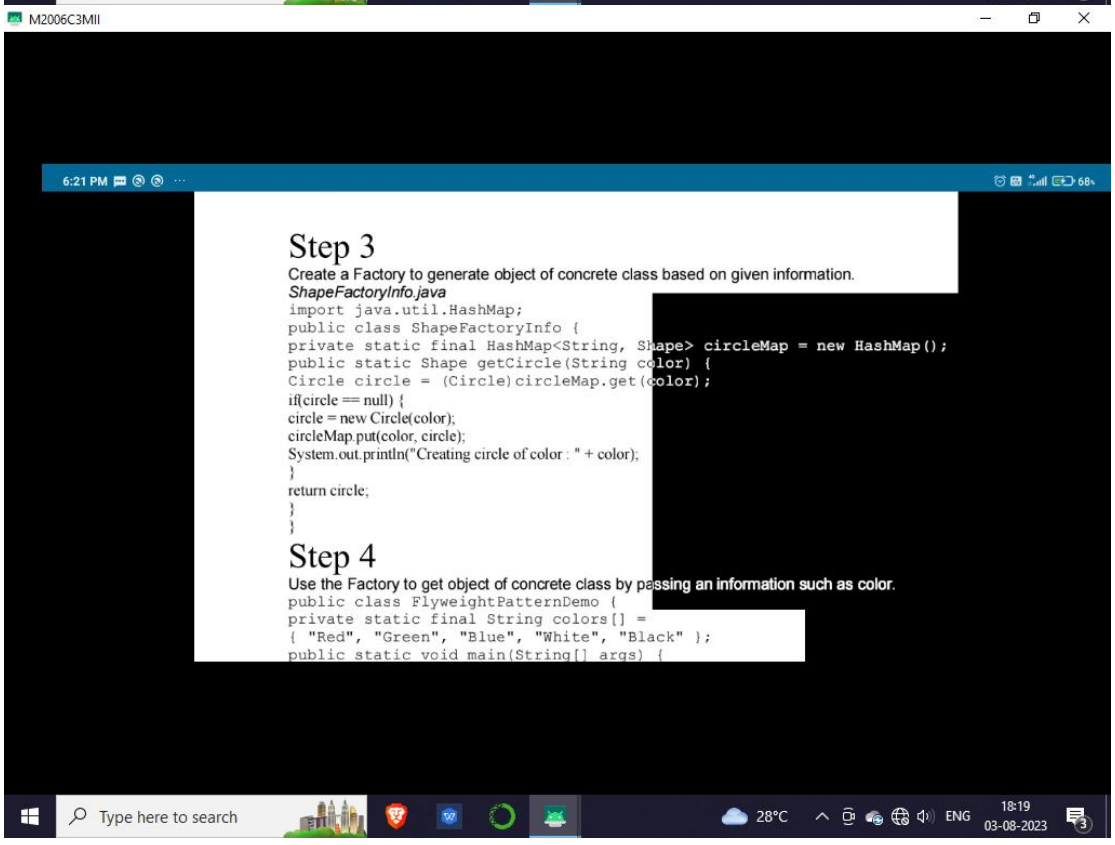


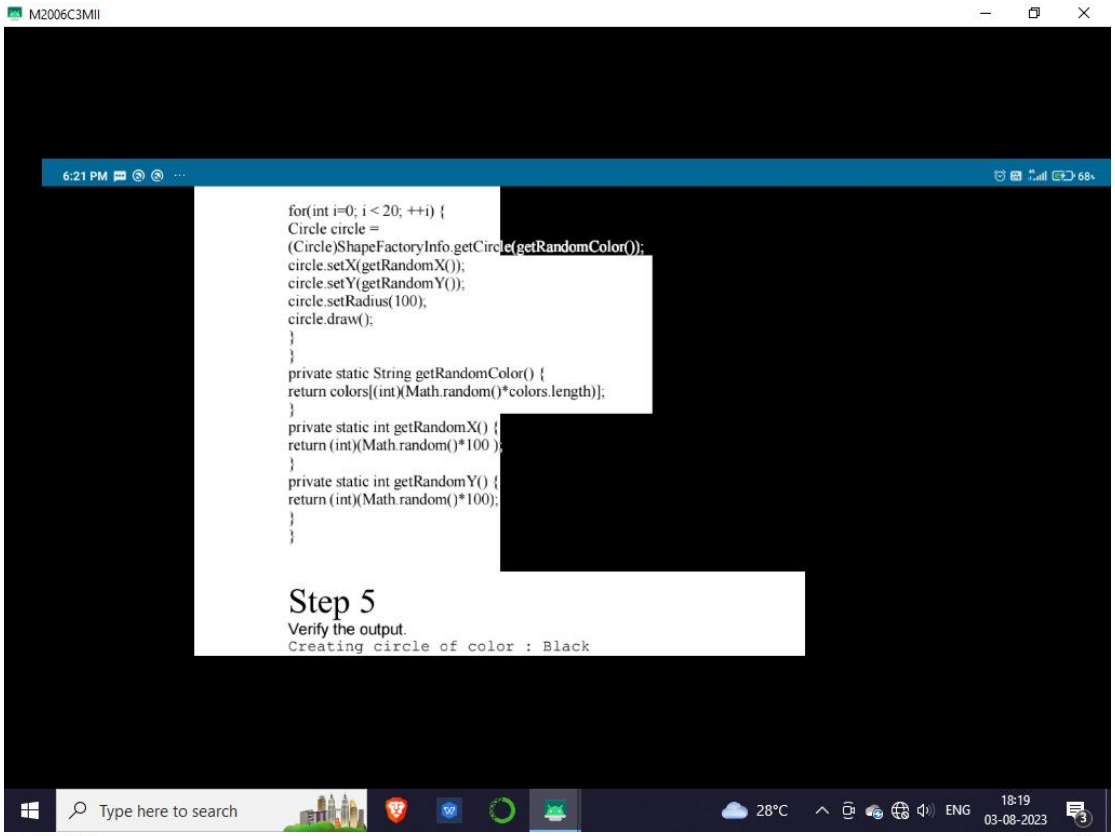
Type here to search 28°C ENG 18:18 03-08-2023

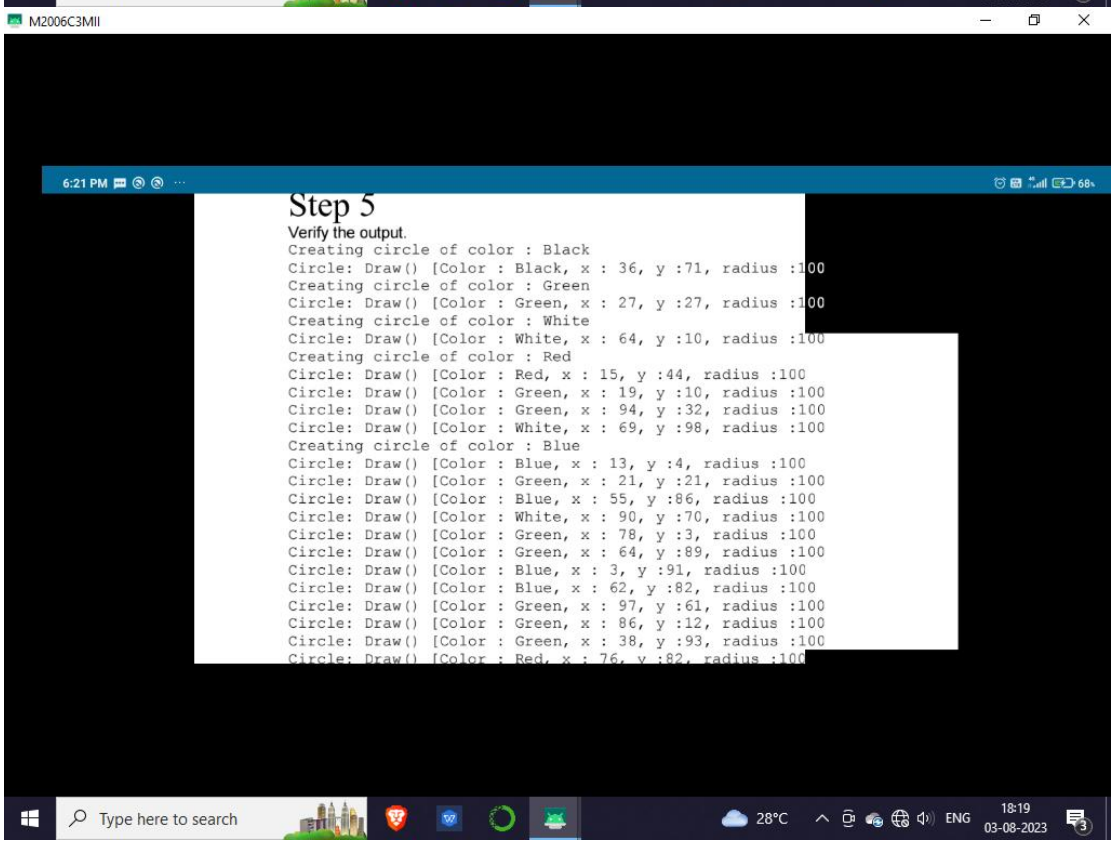

M2006C3MII
6:20 PM
Extrinsic state
Extrinsic
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create an interface.
Shape.java
public interface Shape {
void draw();
}
Step 2
Create concrete class implementing the same interface.
Circle.java
public class Circle implements Shape {
Type here to search
28°C
ENG
18:18
03-08-2023

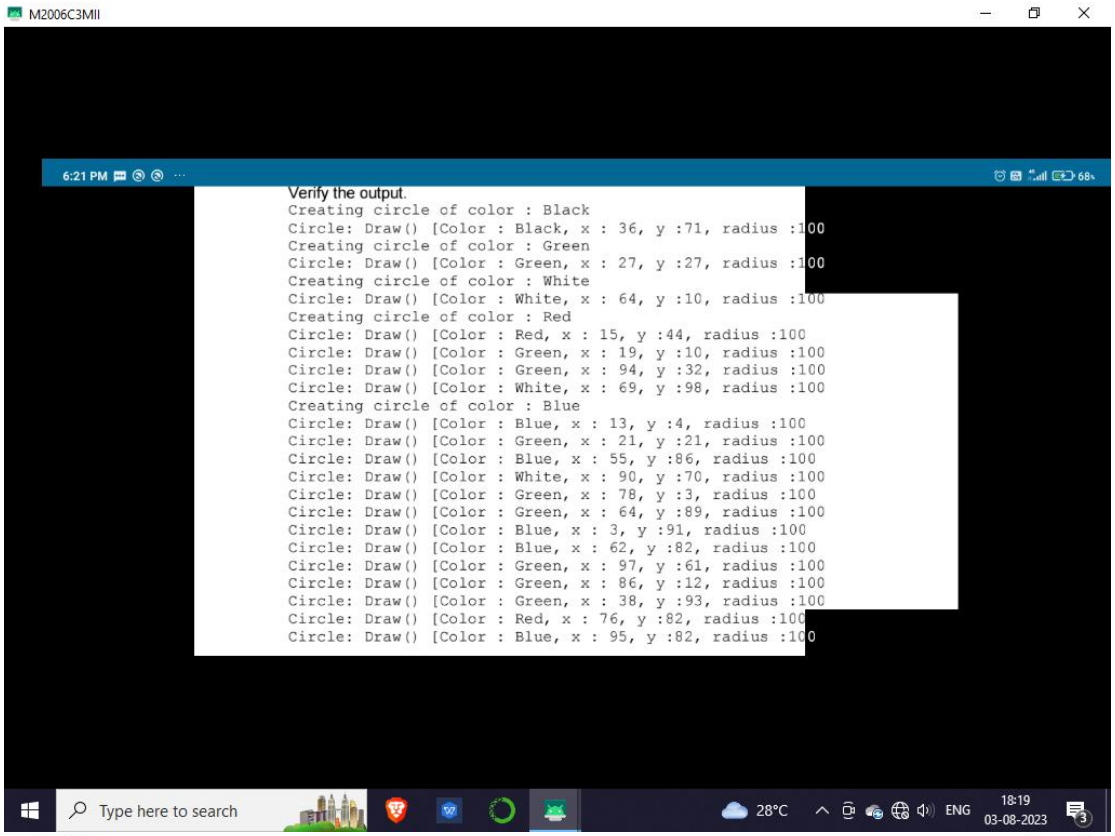

M2006C3MII
6:20 PM
Step 2
Create concrete class implementing the same interface.
Circle.java
public class Circle implements Shape {
private String color;
private int x;
private int y;
private int radius;
public Circle(String color){
this.color = color;
}
public void setX(int x) {
this.x = x;
}
public void setY(int y) {
this.y = y;
}
public void setRadius(int radius) {
this.radius = radius;
}
Type here to search
28°C
ENG
18:18
03-08-2023

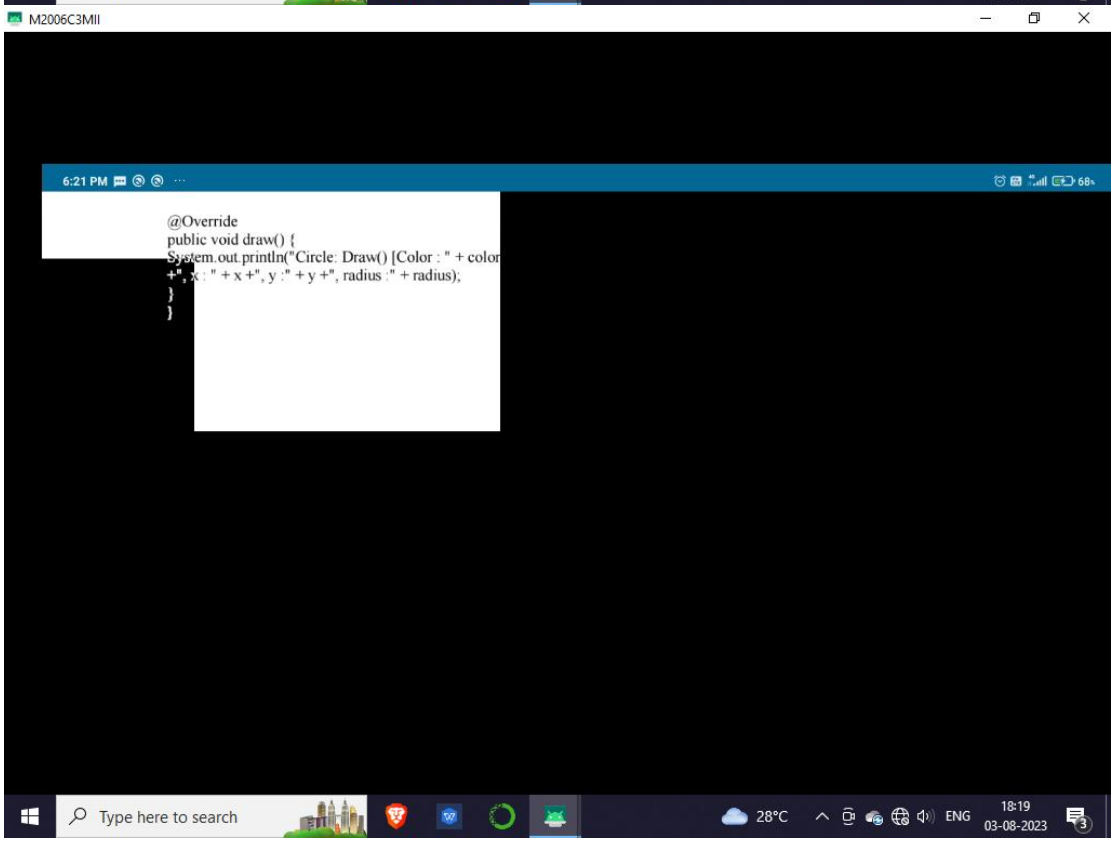

M2006C3MII
6:20 PM
@Override
public void draw() {
System.out.println("Circle: Draw() [Color : " + color
+", x : " + x +", y :" + y +", radius :" + radius);
}
}
Step 3
Create a Factory to generate object of concrete class based on given information.
ShapeFactoryInfo.java
import java.util.HashMap;
public class ShapeFactoryInfo {
private static final HashMap<String, Shape> circleMap = new HashMap();
public static Shape getCircle(String color) {
Circle circle = (Circle)circleMap.get(color);
if(circle == null) {
circle = new Circle(color);
circleMap.put(color, circle);
System.out.println("Creating circle of color : " + color);
}
return circle;
}
}
Type here to search
28°C
ENG
18:19
03-08-2023


M2006C3MII
6:21 PM
Step 3
Create a Factory to generate object of concrete class based on given information.
ShapeFactoryInfo.java
import java.util.HashMap;
public class ShapeFactoryInfo {
private static final HashMap<String, Shape> circleMap = new HashMap();
public static Shape getCircle(String color) {
Circle circle = (Circle)circleMap.get(color);
if(circle == null) {
circle = new Circle(color);
circleMap.put(color, circle);
System.out.println("Creating circle of color : " + color);
}
return circle;
}
}
Step 4
Use the Factory to get object of concrete class by passing an information such as color.
public class FlyweightPatternDemo {
private static final String colors[] =
{ "Red", "Green", "Blue", "White", "Black" };
public static void main(String[] args) {
Type here to search
28°C
ENG
18:19
03-08-2023


M2006C3MII
6:21 PM
for(int i=0; i < 20; ++i) {
Circle circle =
(Circle)ShapeFactoryInfo.getCircle(getRandomColor());
circle.setX(getRandomX());
circle.setY(getRandomY());
circle.setRadius(100);
circle.draw();
}
}
private static String getRandomColor() {
return colors[(int)(Math.random()*colors.length)];
}
private static int getRandomX() {
return (int)(Math.random()*100);
}
private static int getRandomY() {
return (int)(Math.random()*100);
}
}
Step 5
Verify the output.
Creating circle of color : Black
Type here to search
28°C
ENG
18:19
03-08-2023


M2006C3MII
6:21 PM
Step 5
Verify the output.
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Type here to search
28°C
ENG
18:19
03-08-2023


M2006C3MII
6:21 PM
Verify the output.
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
Type here to search
28°C
ENG
18:19
03-08-2023


M2006C3MII
6:21 PM
@Override
public void draw() {
System.out.println("Circle: Draw() [Color : " + color
+", x : " + x +", y :" + y +", radius :" + radius);
}
}
Type here to search
28°C
ENG
18:19
03-08-2023