M2006C3MII

6:05 PM

UNIT-II

Syllabus: Creational Patterns: Abstract Factory, Builder, Factory Method, Prototype, Singleton, Creational Patterns

Abstract Factory Patterns:

Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

- **Also Known As**

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

M2006C3MII

6:05 PM

Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.

- **Also Known As**

KIT

Implementation

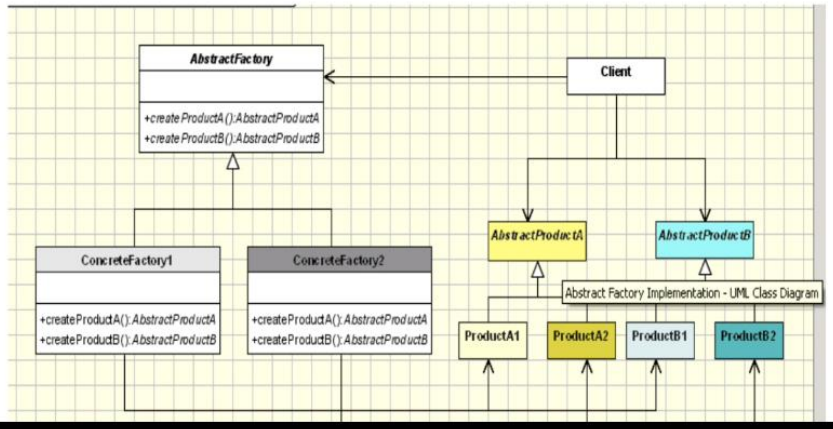
The pattern basically works as shown below, in the UML diagram:

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

M2006C3MII

6:05 PM

The pattern basically works as shown below, in the UML diagram:

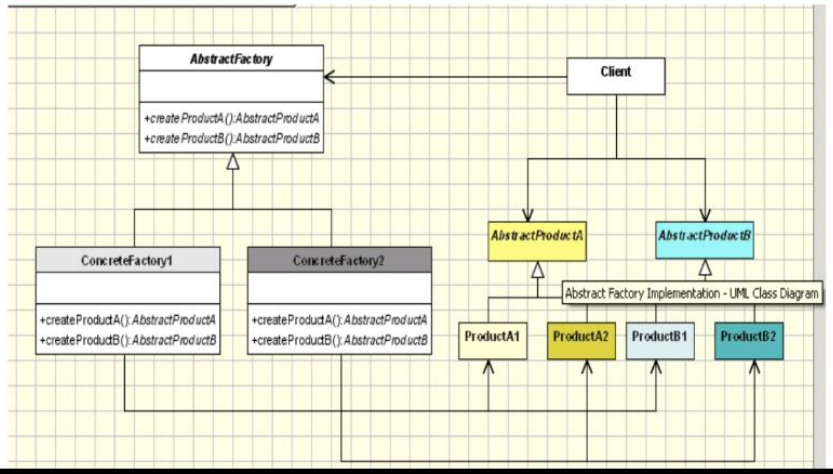


Type here to search

28°C ENG 18:03 03-08-2023

M2006C3MII

6:05 PM



Type here to search

28°C ENG 18:03 03-08-2023

M2006C3MII

6:05 PM

The classes that participate to the Abstract Factory pattern are:

- **AbstractFactory** - declares an interface for operations that create abstract products.
- **ConcreteFactory** - implements operations to create concrete products.
- **AbstractProduct** - declares an interface for a type of product objects.
- **Product** - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
- **Client** - uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

Applicability & Examples

We should use the Abstract Factory design pattern when:

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

M2006C3MII

6:05 PM

Applicability & Examples

We should use the Abstract Factory design pattern when:

- a system needs to be independent from the way the products it works with are created.
- a system is or should be configured to work with multiple families of products.
- a family of related products is designed to be used together and we need to enforce this constraint
- the creation of a library of products is needed, for which is relevant only the interface, not the implementation, too.

Implementation

Type here to search 28°C ENG 18:04 03-08-2023

M2006C3MII

6:05 PM

Implementation

We're going to create a *ColorInfo* and *ShapeInfo* interfaces and concrete classes implementing these interfaces. We create an abstract factory class *AbstractFactory* as next step. Factory classes *ShapeInfoFactory* and *ColorInfoFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

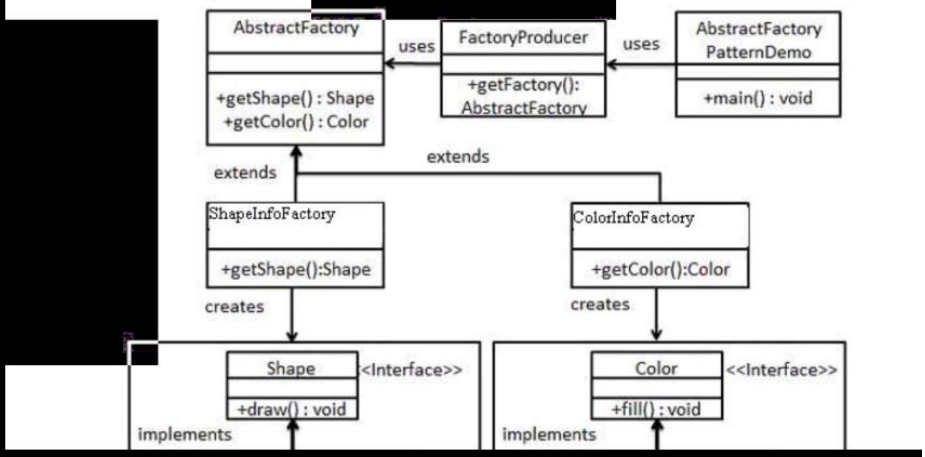
AbstractFactoryPatternDemo class uses *FactoryProducer* toget a *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.

Type here to search 28°C ENG 18:04 03-08-2023

M2006C3MII

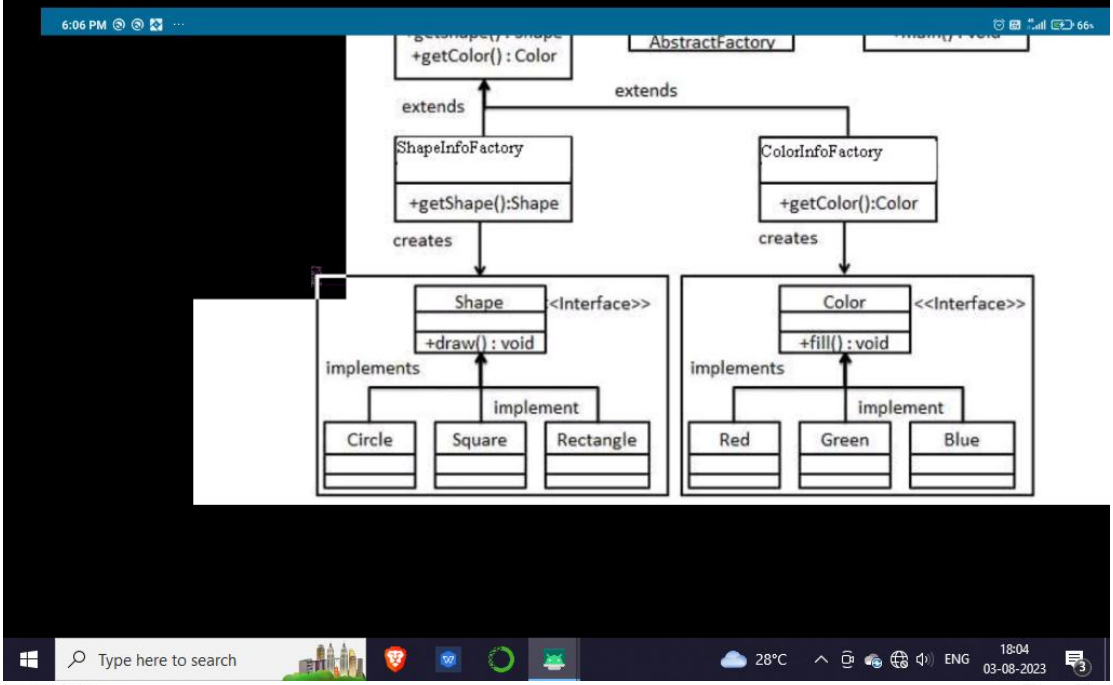
6:06 PM

Class Diagram:



Type here to search 28°C ENG 18:04 03-08-2023

M2006C3MII



M2006C3MII

Step 1

Create an interface for Shapes.

Shape.java

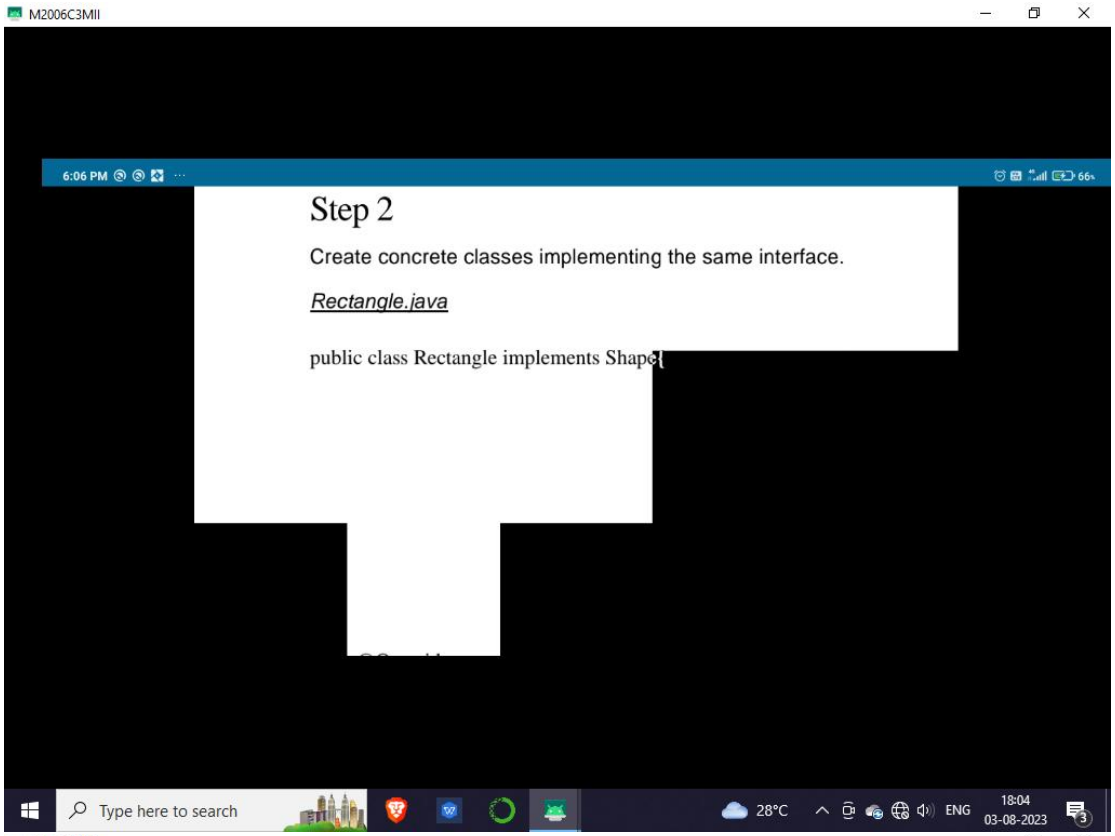
```
public interface Shape {
        void draw();
                }
```

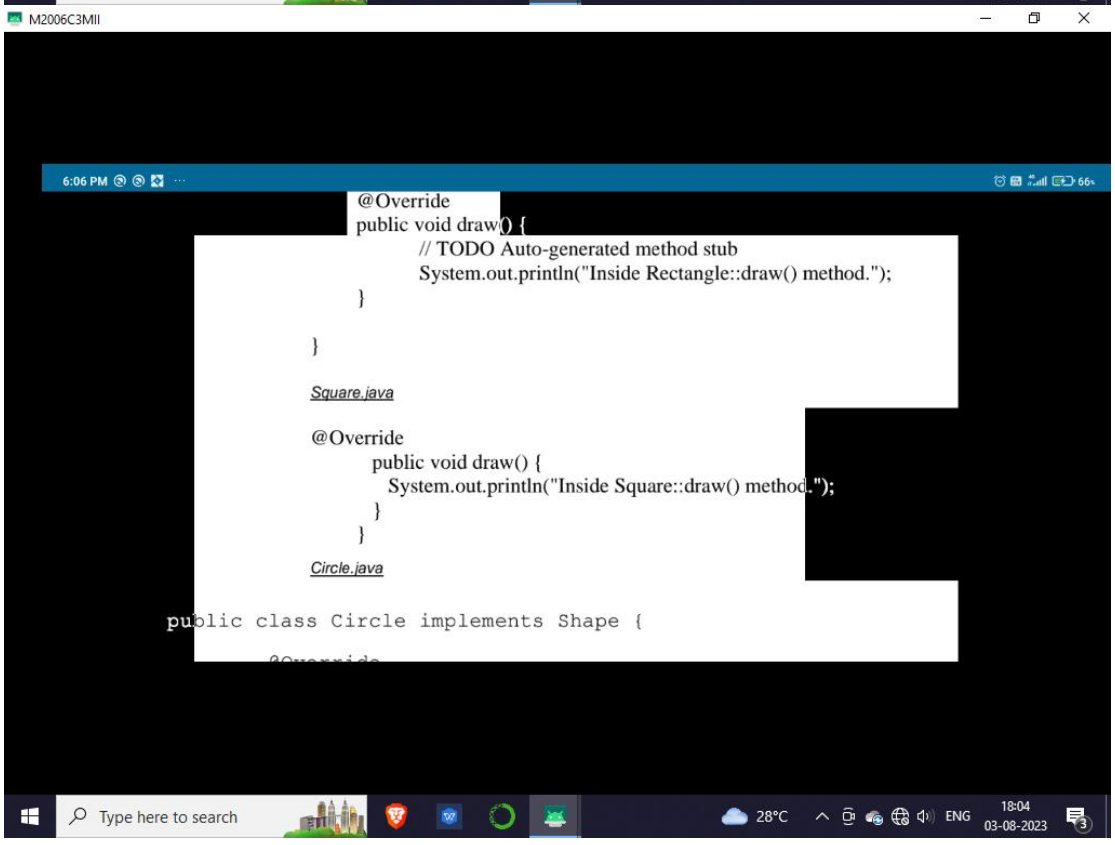
Step 2

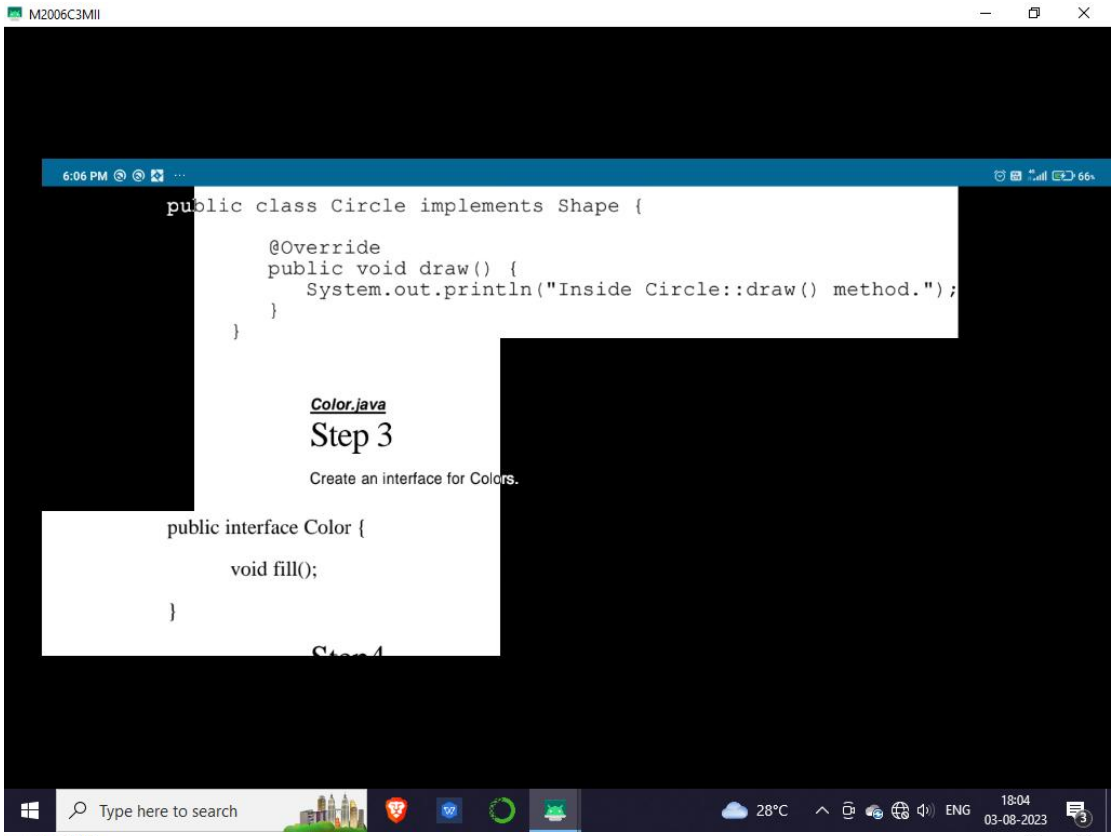
Create concrete classes implementing the same interface.

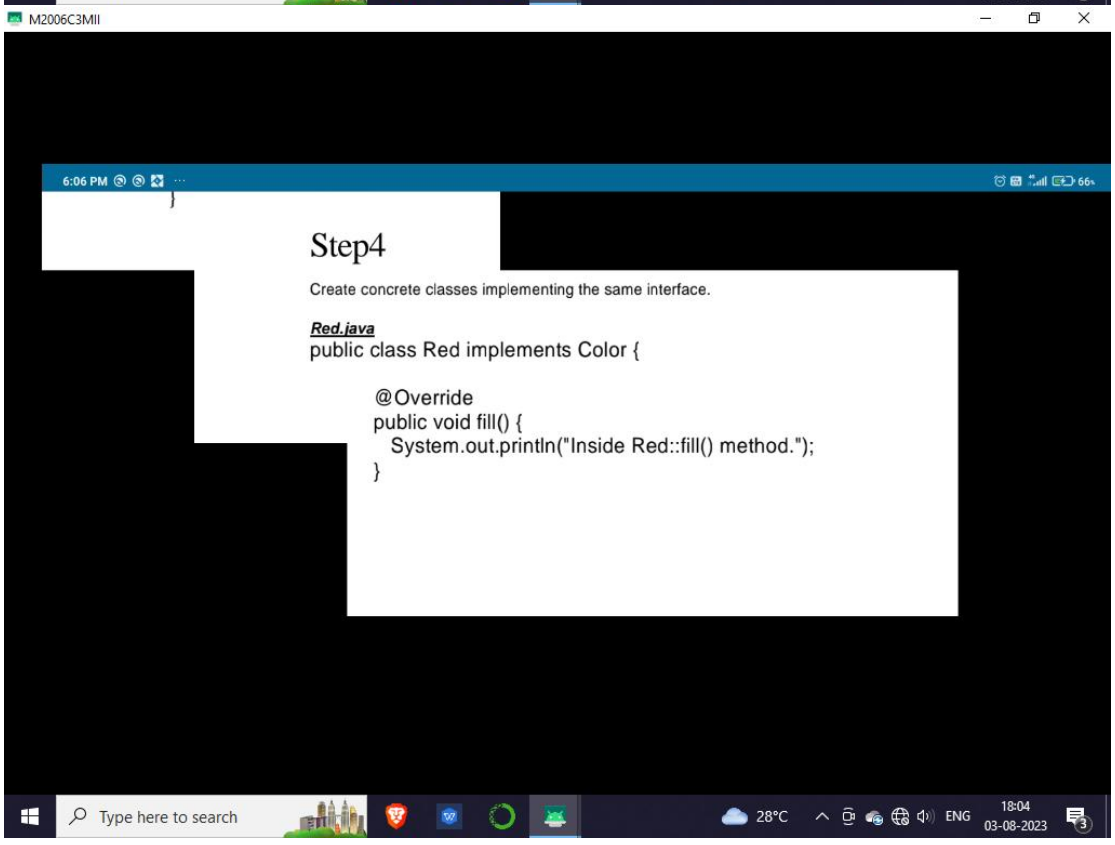
Rectangle.java

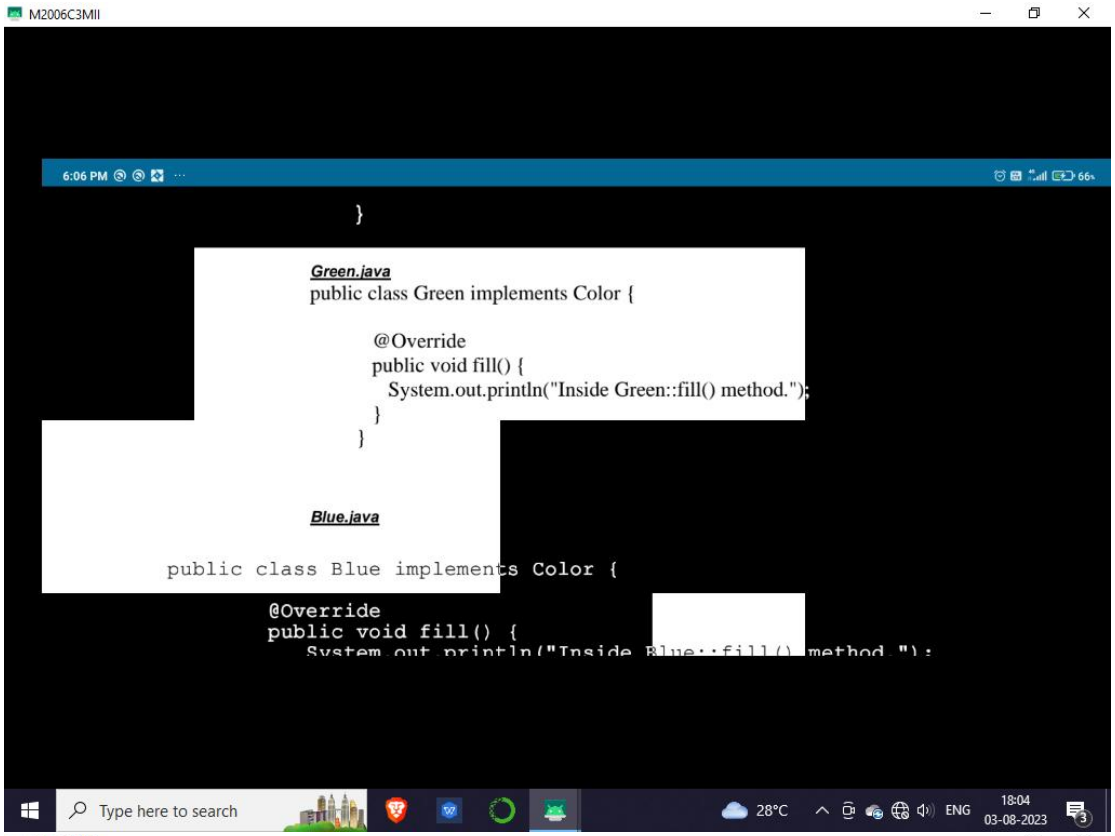
```
public class Rectangle implements Shape{
```

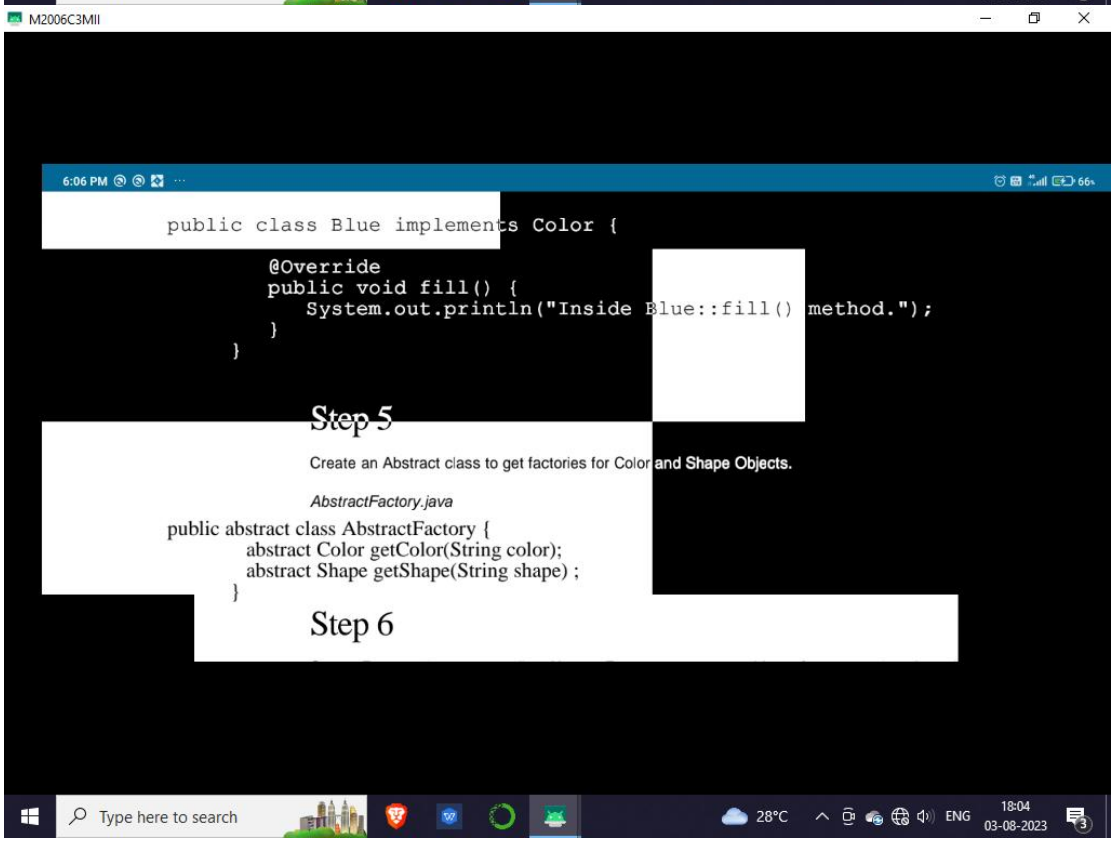

M2006C3MII
6:06 PM
Step 2
Create concrete classes implementing the same interface.
Rectangle.java
public class Rectangle implements Shape{
Type here to search
28°C
ENG
18:04
03-08-2023

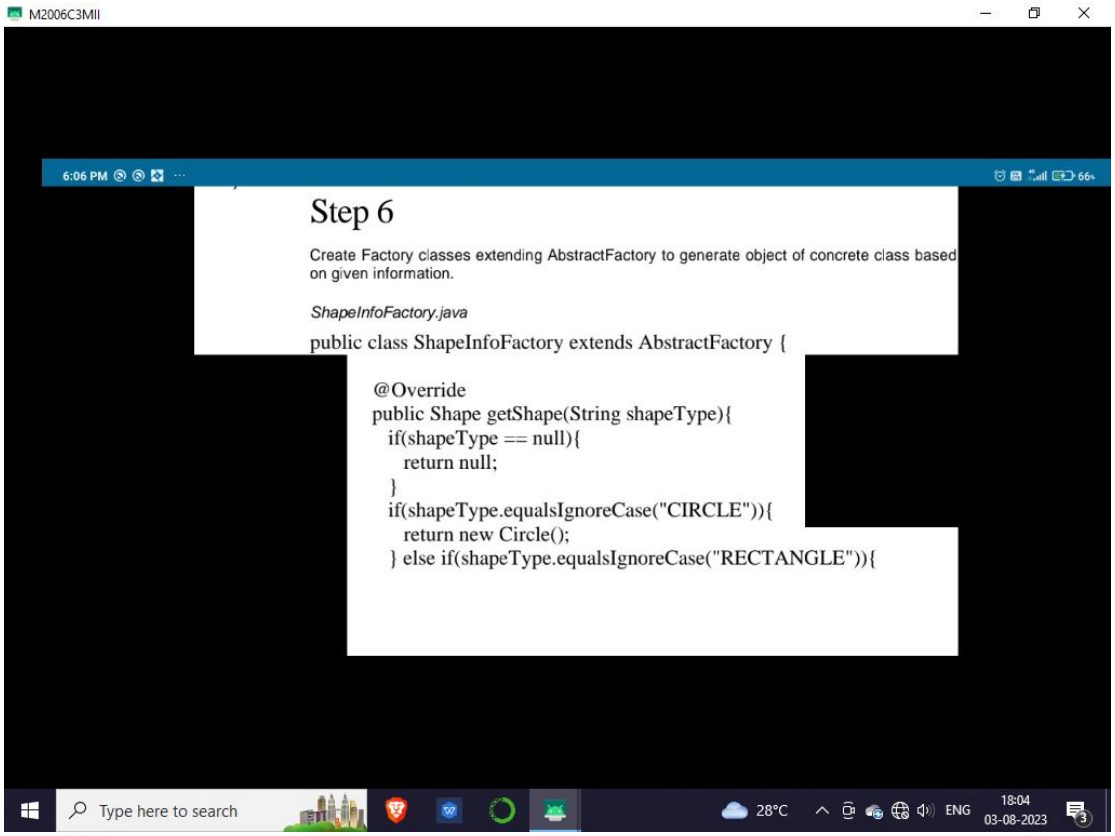

M2006C3MII
6:06 PM
@Override
public void draw() {
// TODO Auto-generated method stub
System.out.println("Inside Rectangle::draw() method.");
}
}
Square.java
@Override
public void draw() {
System.out.println("Inside Square::draw() method.");
}
}
Circle.java
public class Circle implements Shape {
Type here to search
28°C
ENG
18:04
03-08-2023

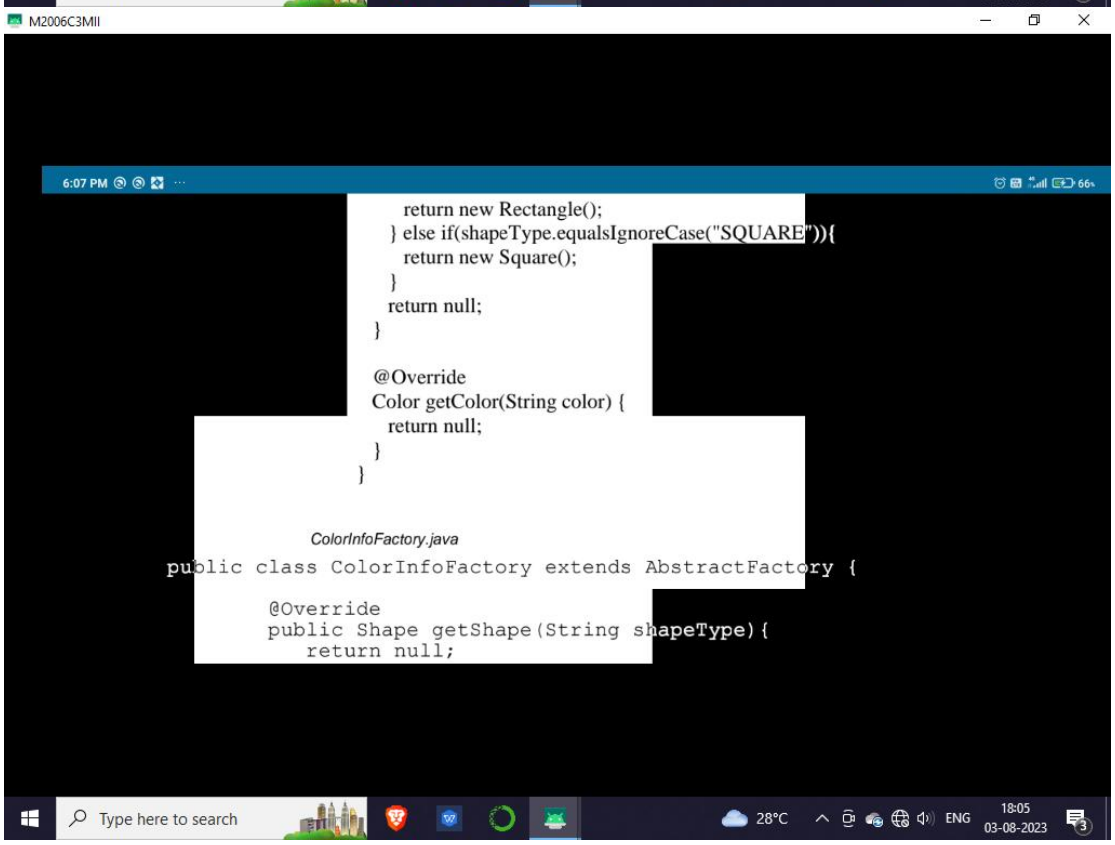

M2006C3MII
6:06 PM
public class Circle implements Shape {
@Override
public void draw() {
System.out.println("Inside Circle::draw() method.");
}
}
Color.java
Step 3
Create an interface for Colors.
public interface Color {
void fill();
}
Type here to search
28°C
ENG
18:04
03-08-2023

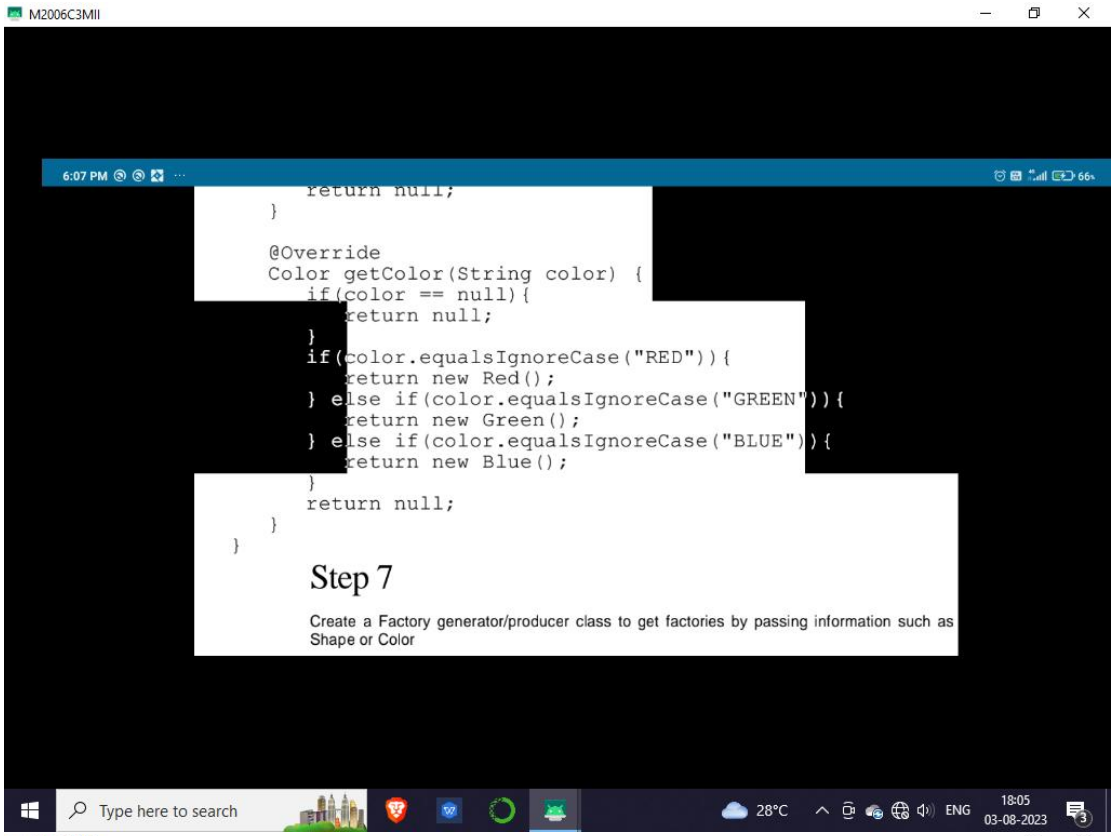

M2006C3MII
6:06 PM
}
Step4
Create concrete classes implementing the same interface.
Red.java
public class Red implements Color {
@Override
public void fill() {
System.out.println("Inside Red::fill() method.");
}
Type here to search
28°C
ENG
18:04
03-08-2023

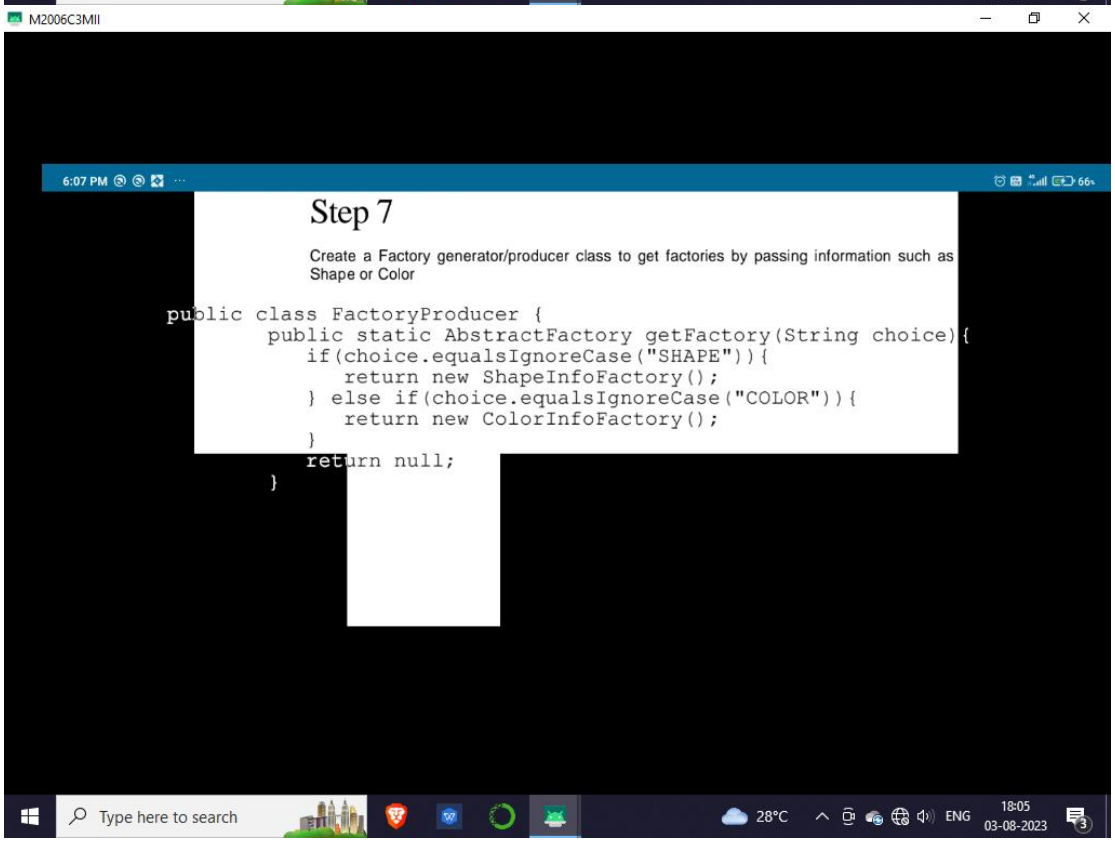

M2006C3MII
6:06 PM
}
Green.java
public class Green implements Color {
@Override
public void fill() {
System.out.println("Inside Green::fill() method.");
}
}
Blue.java
public class Blue implements Color {
@Override
public void fill() {
System.out.println("Inside Blue::fill() method.");
Type here to search
28°C
ENG
18:04
03-08-2023

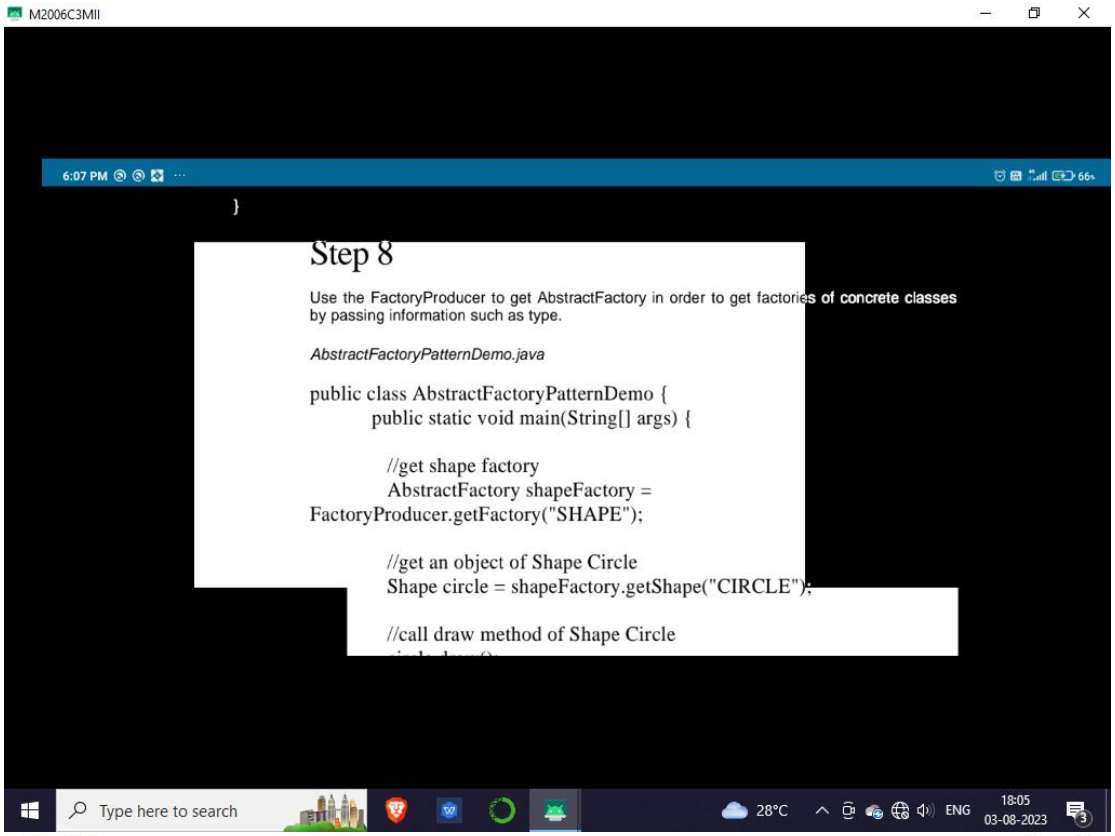

M2006C3MII
6:06 PM
public class Blue implements Color {
@Override
public void fill() {
System.out.println("Inside Blue::fill() method.");
}
}
Step 5
Create an Abstract class to get factories for Color and Shape Objects.
AbstractFactory.java
public abstract class AbstractFactory {
abstract Color getColor(String color);
abstract Shape getShape(String shape) ;
}
Step 6
Type here to search
28°C
ENG
18:04
03-08-2023

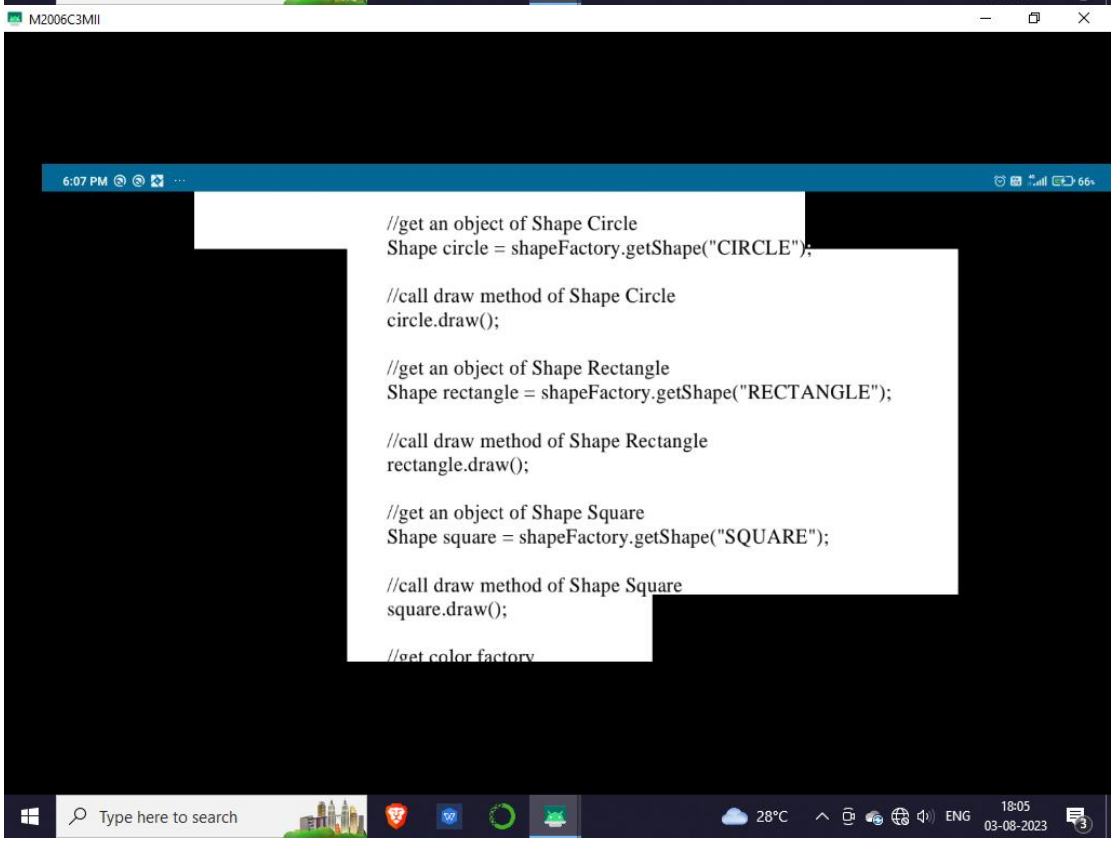

M2006C3MII
6:06 PM
Step 6
Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.
ShapeInfoFactory.java
public class ShapeInfoFactory extends AbstractFactory {
@Override
public Shape getShape(String shapeType){
if(shapeType == null){
return null;
}
if(shapeType.equalsIgnoreCase("CIRCLE")){
return new Circle();
} else if(shapeType.equalsIgnoreCase("RECTANGLE")){
Type here to search
28°C
ENG
18:04
03-08-2023

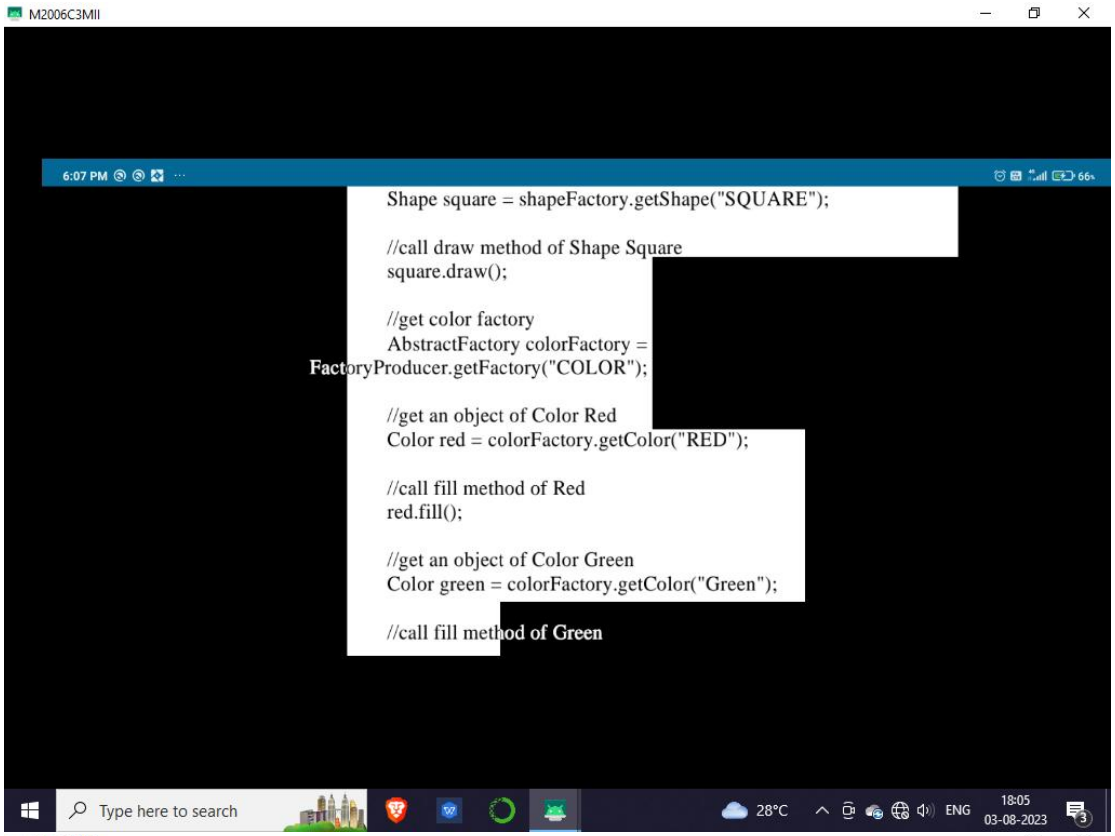

M2006C3MII
6:07 PM
return new Rectangle();
} else if(shapeType.equalsIgnoreCase("SQUARE")){
return new Square();
}
return null;
}
@Override
Color getColor(String color) {
return null;
}
}
ColorInfoFactory.java
public class ColorInfoFactory extends AbstractFactory {
@Override
public Shape getShape(String shapeType){
return null;
Type here to search
28°C
ENG
18:05
03-08-2023

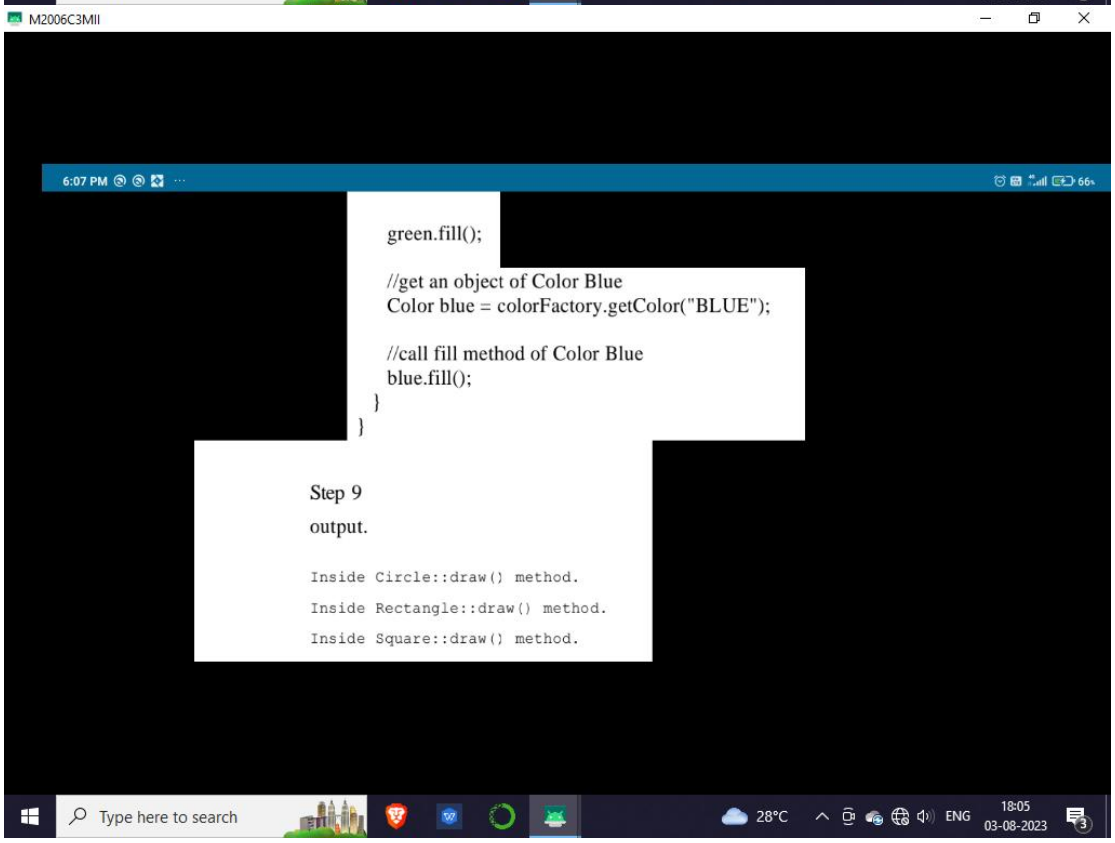

M2006C3MII
6:07 PM
return null;
}
@Override
Color getColor(String color) {
if(color == null){
return null;
}
if(color.equalsIgnoreCase("RED")){
return new Red();
} else if(color.equalsIgnoreCase("GREEN")){
return new Green();
} else if(color.equalsIgnoreCase("BLUE")){
return new Blue();
}
return null;
}
}
Step 7
Create a Factory generator/producer class to get factories by passing information such as Shape or Color
Type here to search
28°C
ENG
18:05
03-08-2023

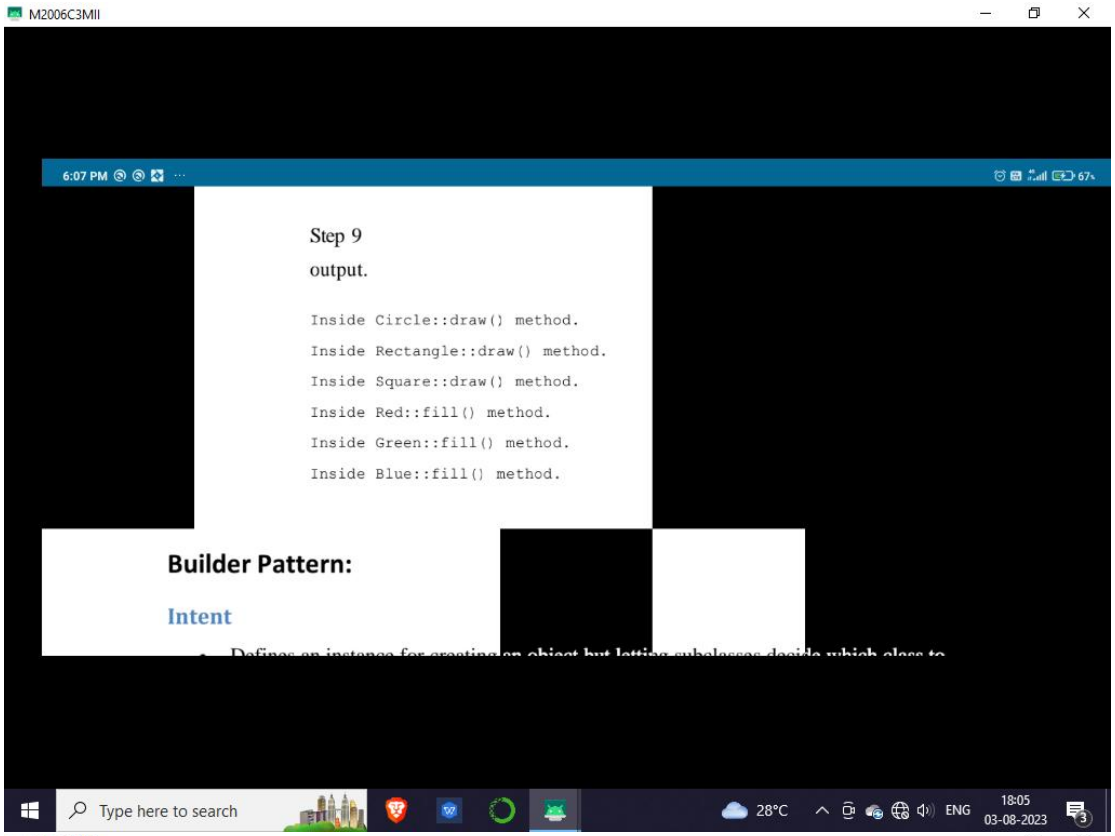

M2006C3MII
6:07 PM
Step 7
Create a Factory generator/producer class to get factories by passing information such as Shape or Color
public class FactoryProducer {
public static AbstractFactory getFactory(String choice){
if(choice.equalsIgnoreCase("SHAPE")){
return new ShapeInfoFactory();
} else if(choice.equalsIgnoreCase("COLOR")){
return new ColorInfoFactory();
}
return null;
}
Type here to search
28°C
ENG
18:05
03-08-2023

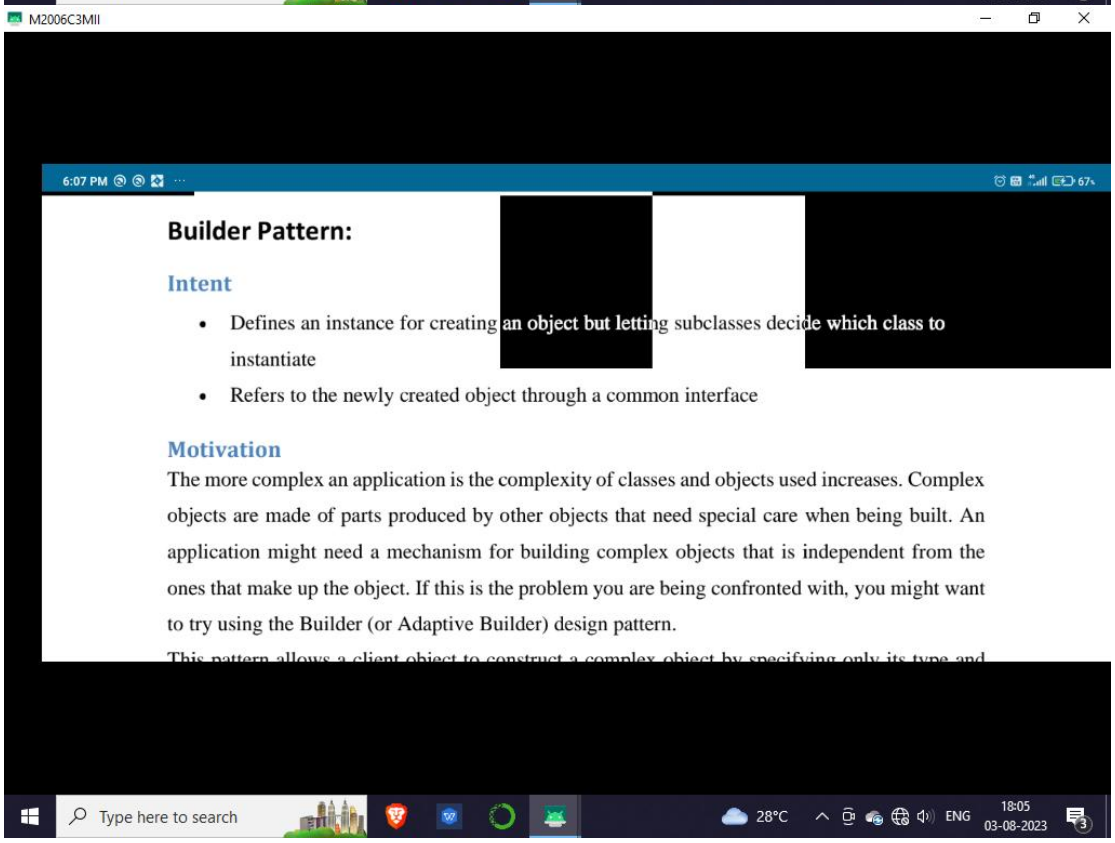

M2006C3MII
6:07 PM
}
Step 8
Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing information such as type.
AbstractFactoryPatternDemo.java
public class AbstractFactoryPatternDemo {
public static void main(String[] args) {
//get shape factory
AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");
//get an object of Shape Circle
Shape circle = shapeFactory.getShape("CIRCLE");
//call draw method of Shape Circle
Type here to search
28°C
ENG
18:05
03-08-2023


M2006C3MII
6:07 PM
//get an object of Shape Circle
Shape circle = shapeFactory.getShape("CIRCLE");
//call draw method of Shape Circle
circle.draw();
//get an object of Shape Rectangle
Shape rectangle = shapeFactory.getShape("RECTANGLE");
//call draw method of Shape Rectangle
rectangle.draw();
//get an object of Shape Square
Shape square = shapeFactory.getShape("SQUARE");
//call draw method of Shape Square
square.draw();
//get color factory
Type here to search
28°C
ENG
18:05
03-08-2023


M2006C3MII
6:07 PM
Shape square = shapeFactory.getShape("SQUARE");
//call draw method of Shape Square
square.draw();
//get color factory
AbstractFactory colorFactory =
FactoryProducer.getFactory("COLOR");
//get an object of Color Red
Color red = colorFactory.getColor("RED");
//call fill method of Red
red.fill();
//get an object of Color Green
Color green = colorFactory.getColor("Green");
//call fill method of Green
Type here to search
28°C
ENG
18:05
03-08-2023


M2006C3MII
6:07 PM
green.fill();
//get an object of Color Blue
Color blue = colorFactory.getColor("BLUE");
//call fill method of Color Blue
blue.fill();
}
}
Step 9
output.
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Type here to search
28°C
ENG
18:05
03-08-2023


M2006C3MII
6:07 PM
Step 9
output.
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.
Builder Pattern:
Intent
Type here to search
28°C
ENG
18:05
03-08-2023


M2006C3MII
6:07 PM
Builder Pattern:
Intent
Defines an instance for creating an object but letting subclasses decide which class to instantiate
Refers to the newly created object through a common interface
Motivation
The more complex an application is the complexity of classes and objects used increases. Complex objects are made of parts produced by other objects that need special care when being built. An application might need a mechanism for building complex objects that is independent from the ones that make up the object. If this is the problem you are being confronted with, you might want to try using the Builder (or Adaptive Builder) design pattern.
Type here to search
28°C
ENG
18:05
03-08-2023

M2006C3MII

6:07 PM

Motivation

The more complex an application is the complexity of classes and objects used increases. Complex objects are made of parts produced by other objects that need special care when being built. An application might need a mechanism for building complex objects that is independent from the ones that make up the object. If this is the problem you are being confronted with, you might want to try using the Builder (or Adaptive Builder) design pattern.

This pattern allows a client object to construct a complex object by specifying only its type and content, being shielded from the details related to the objects representation. This way the construction process can be used to create different representations. The logic of this process is

Type here to search 28°C ENG 18:05 03-08-2023

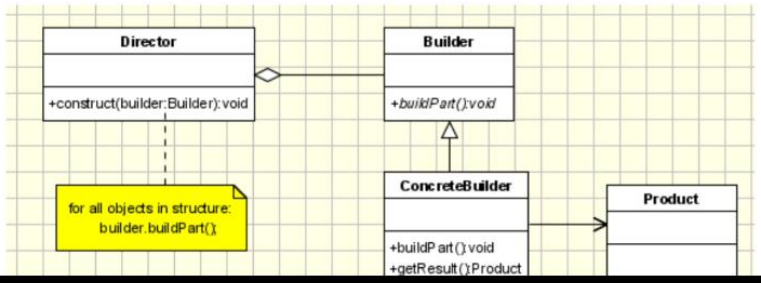
M2006C3MII

6:07 PM

isolated form the actual steps used in creating the complex object, so the process can be used again to create a different object form the same set of simple objects as the first one.

Implementation

The Builder design pattern uses the Factory Builder pattern to decide which concrete class to initiate in order to build the type of object, as we will see below in the UML diagram:

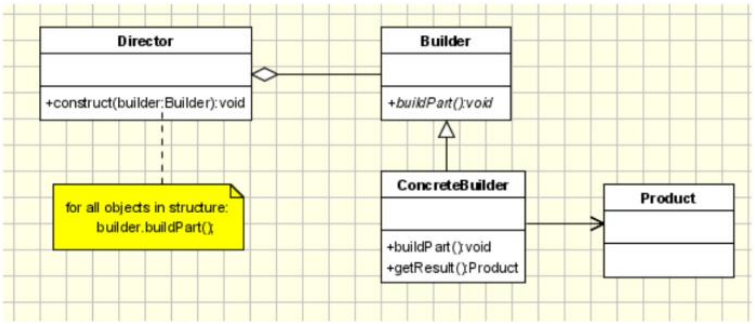


Type here to search 28°C ENG 18:06 03-08-2023

M2006C3MII

6:07 PM

initiate in order to build the type of object, as we will see below in the UML diagram:



Type here to search 28°C ENG 18:06 03-08-2023

M2006C3MII

6:08 PM

The participant's classes in this pattern are:

- The **Builder** class specifies an abstract interface for creating parts of a Product object.
- The **ConcreteBuilder** constructs and puts together parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for saving the product.
- The **Director** class constructs the complex object using the Builder interface.
- The **Product** represents the complex object that is being built.

Implementation

We have considered a case study of fast-food restaurant where a typical MyMealBuilder could be a Pizza and a cold drink. Pizza could be either a Veg Pizza or Chicken Pizza and will be packed by a wrapper. Cold drink could be either a coke or Mazza and will be packed in a bottle.

Type here to search 28°C ENG 18:06 03-08-2023

M2006C3MII

6:08 PM

Implementation

We have considered a case study of fast-food restaurant where a typical MyMealBuilder could be a Pizza and a cold drink. Pizza could be either a Veg Pizza or Chicken Pizza and will be packed by a wrapper. Cold drink could be either a coke or Mazza and will be packed in a bottle.

We are going to create an Item interface representing food items such as Pizza and cold drinks and concrete classes implementing the Item interface and a Wrapping interface representing

packaging of food items and concrete classes implementing the Wrapping interface as Pizza

Type here to search 28°C ENG 18:06 03-08-2023

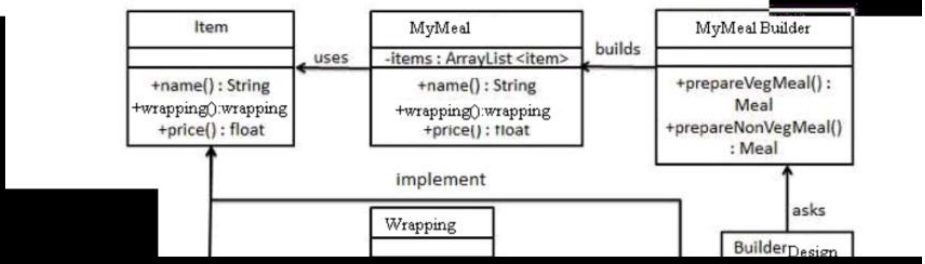
M2006C3MII

6:08 PM

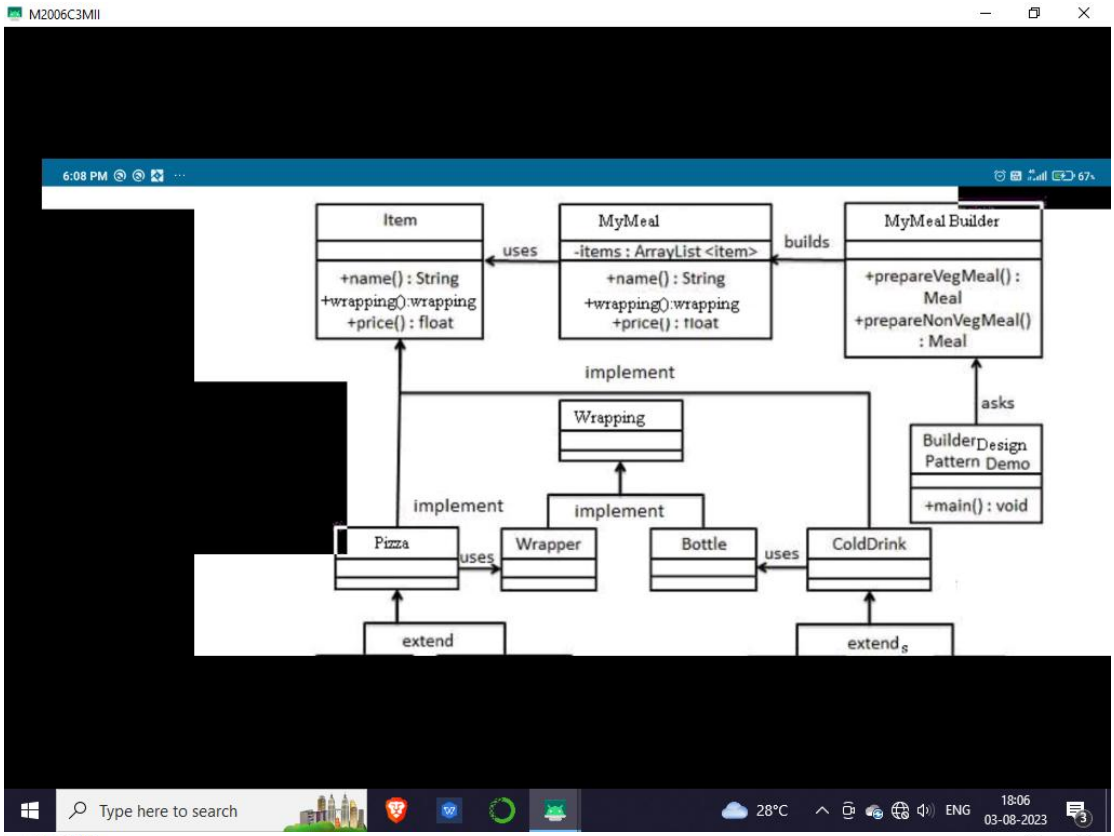
packaging of food items and concrete classes implementing the Wrapping interface as Pizza would be packed in wrapper and cold drink would be packed as bottle.

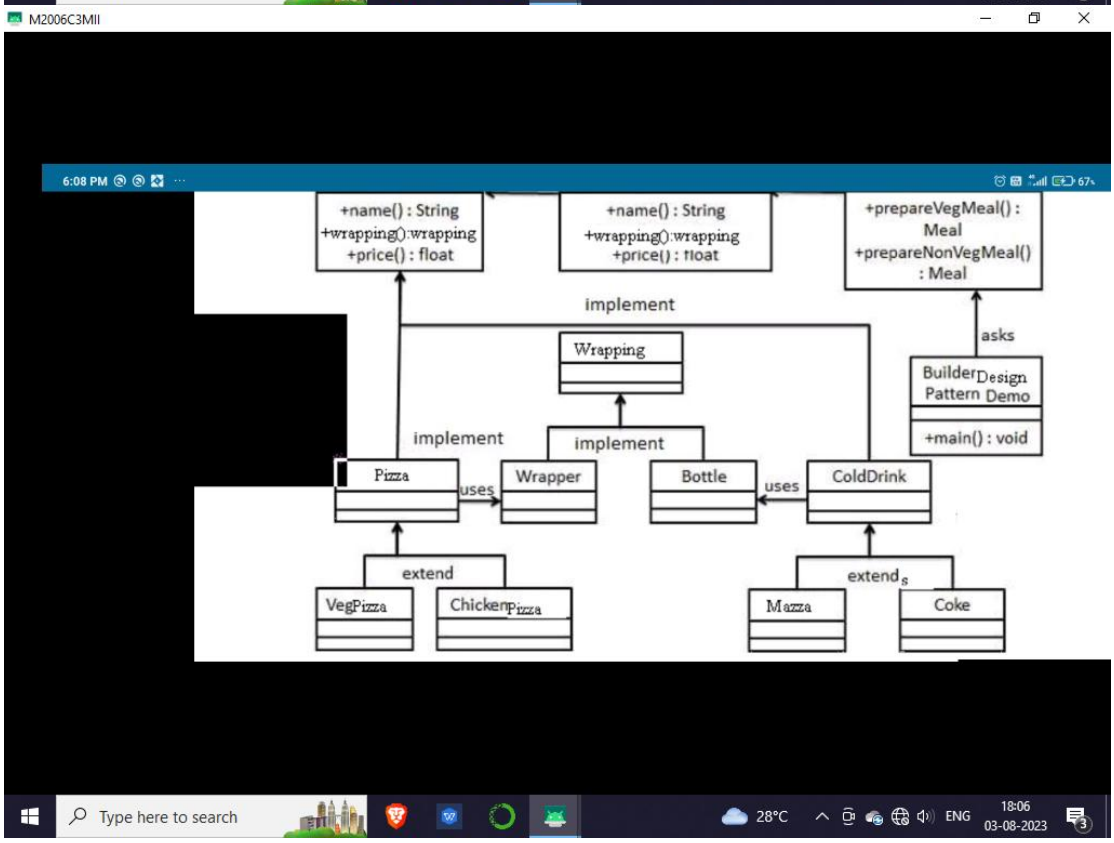
We then create a MyMyMealBuilder class having ArrayList of Item and a MyMyMealBuilder Builder to build different types of MyMealBuilder object by combining Item. BuilderDesignPatternDemo, class will use MyMyMealBuilder Builder to build a MyMealBuilder .

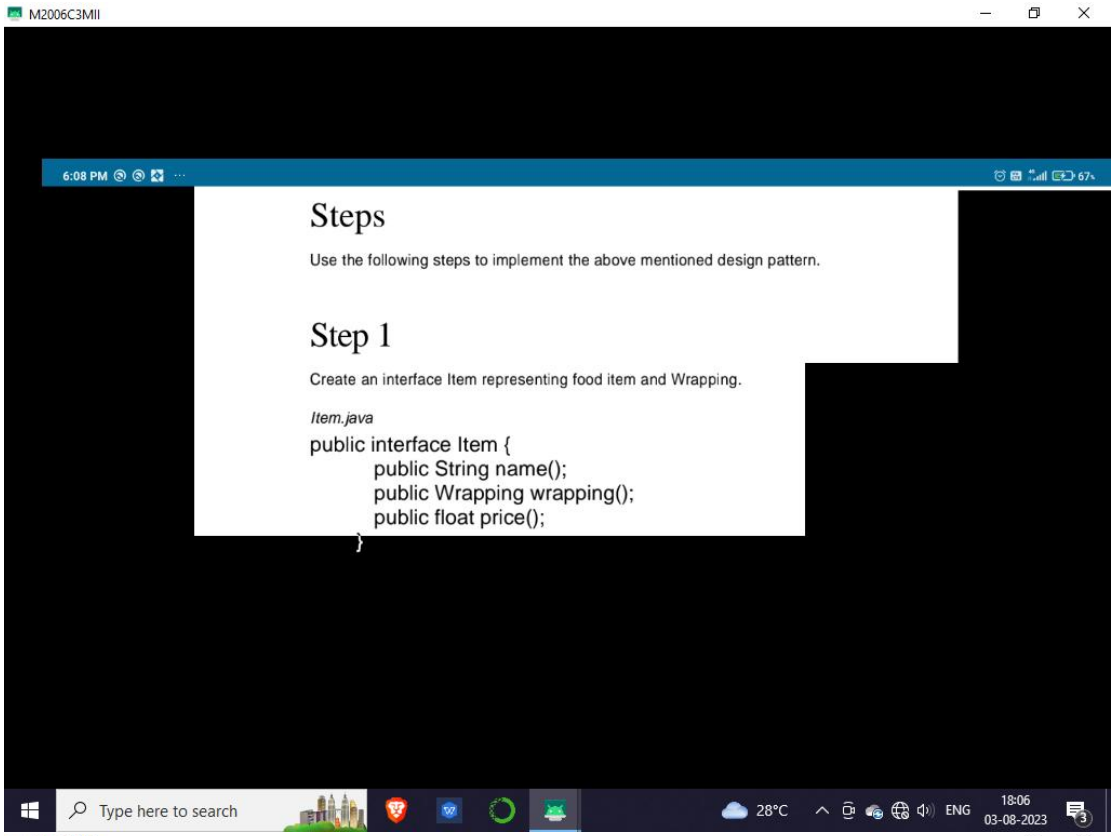
Class Diagram:

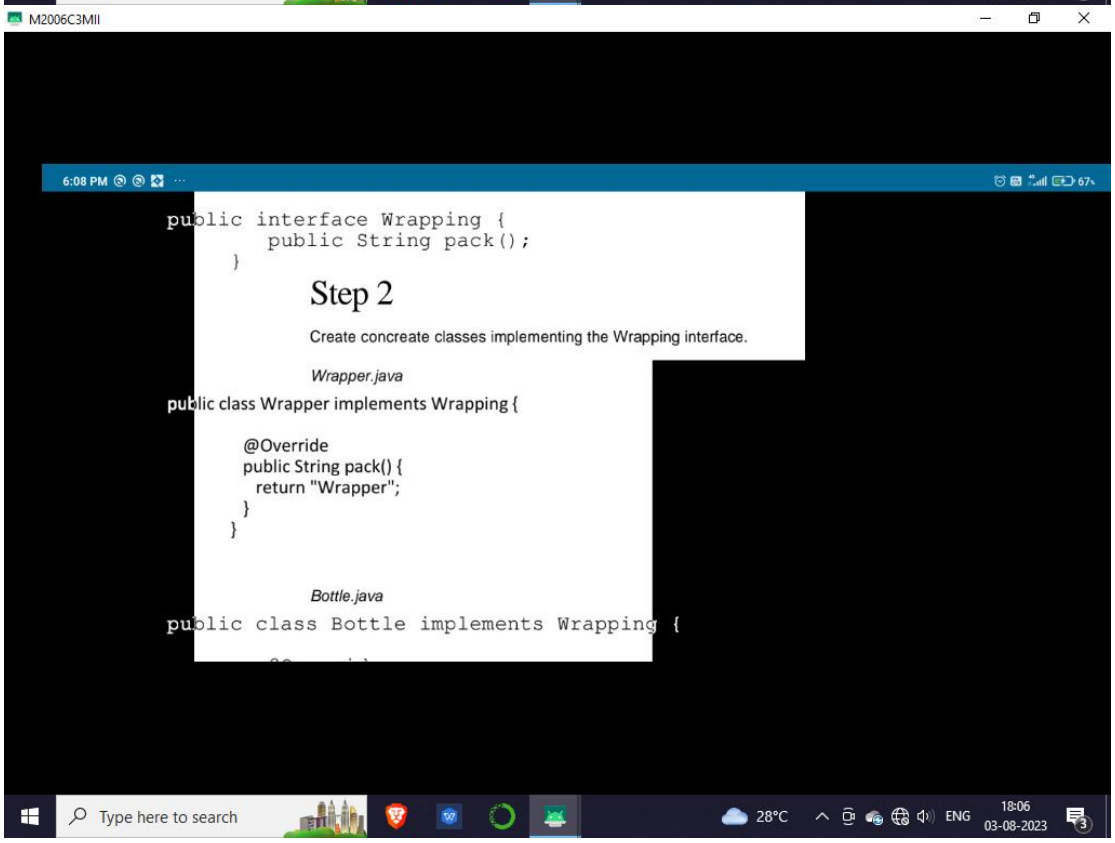


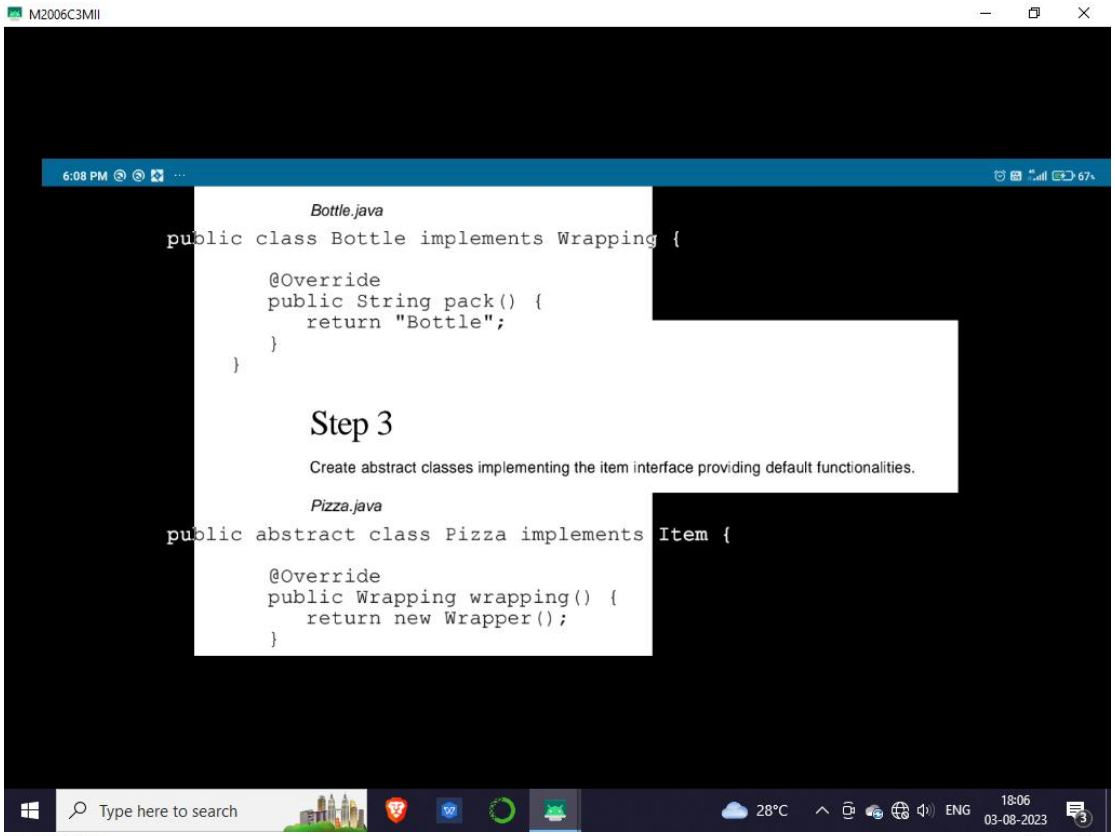
Type here to search 28°C ENG 18:06 03-08-2023

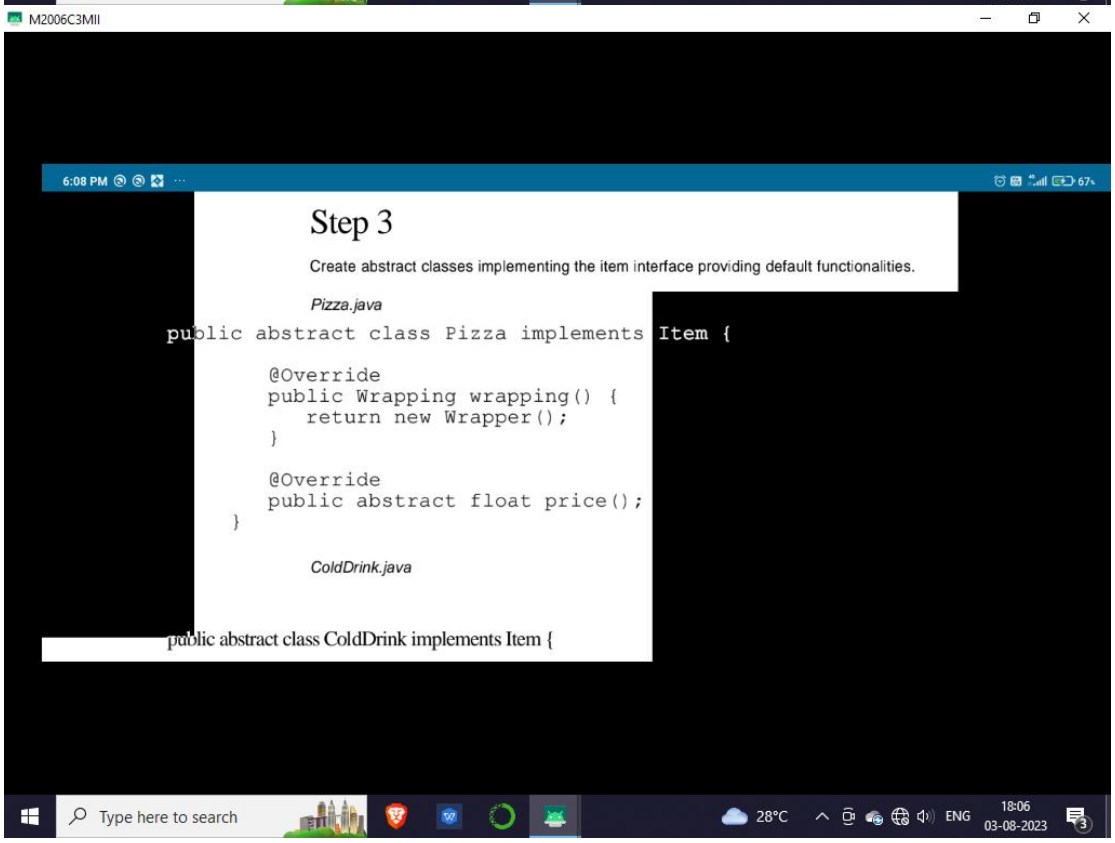

M2006C3MII
6:08 PM
Item
+name() : String
+wrapping():wrapping
+price() : float
uses
MyMeal
-items : ArrayList <item>
+name() : String
+wrapping():wrapping
+price() : float
builds
MyMeal Builder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Wrapping
BuilderDesign Pattern Demo
+main() : void
implement
implement
Pizza
uses
Wrapper
Bottle
uses
ColdDrink
extend
extends
Type here to search
28°C
ENG
18:06
03-08-2023

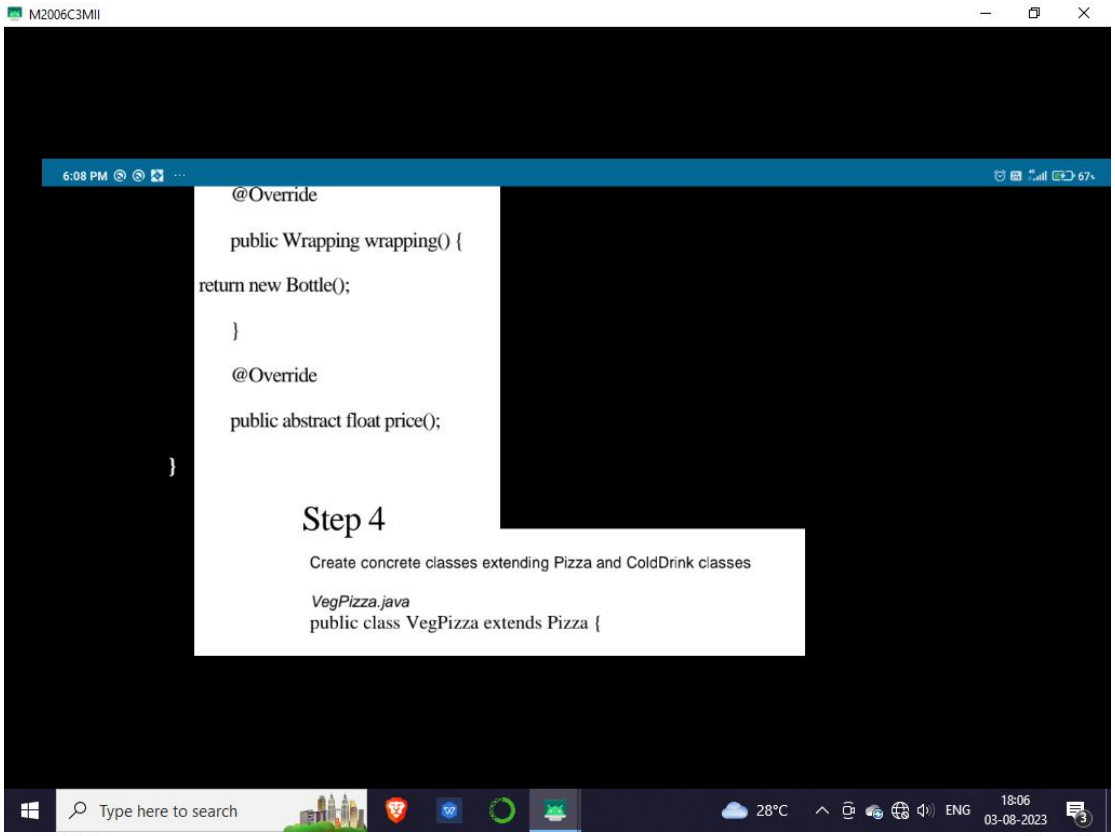

M2006C3MII
6:08 PM
+name() : String
+wrapping():wrapping
+price() : float
+name() : String
+wrapping():wrapping
+price() : float
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Wrapping
BuilderDesign Pattern Demo
+main() : void
implement
implement
Pizza
uses
Wrapper
Bottle
uses
ColdDrink
extend
extends
VegPizza
ChickenPizza
Mazza
Coke
Type here to search
28°C
ENG
18:06
03-08-2023

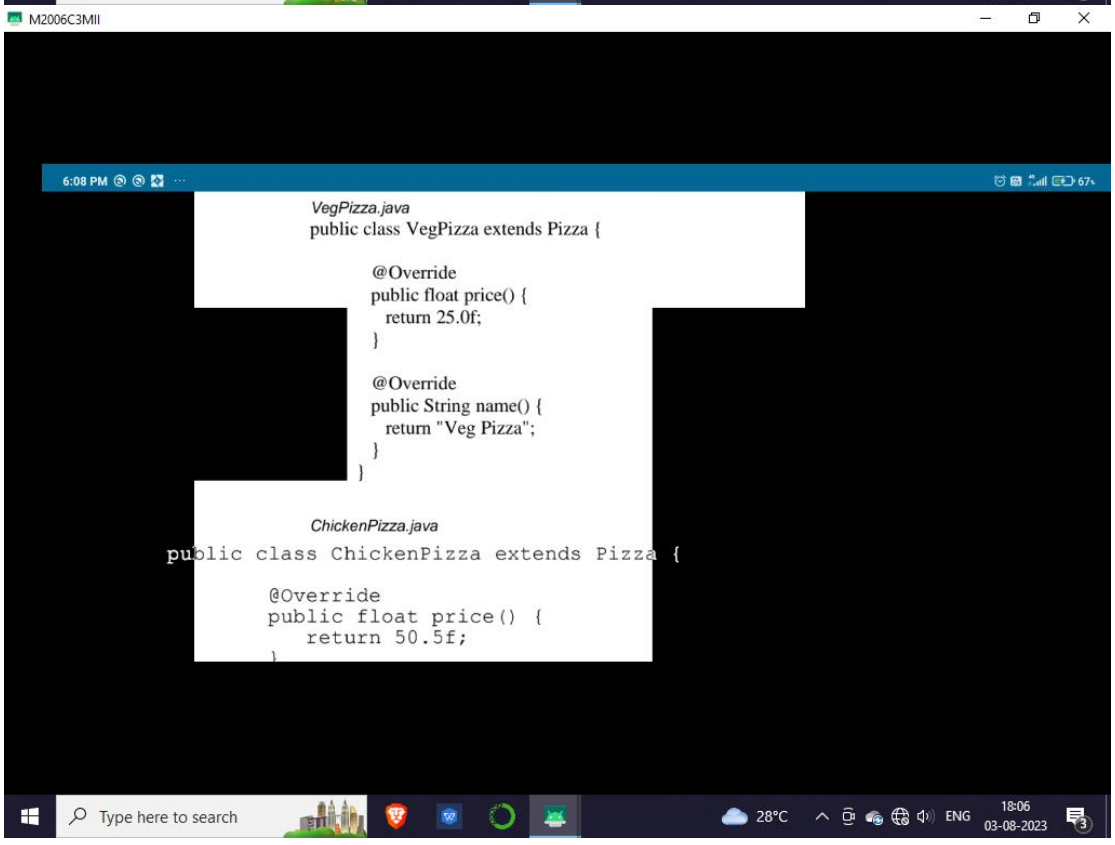

M2006C3MII
6:08 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create an interface Item representing food item and Wrapping.
Item.java
public interface Item {
public String name();
public Wrapping wrapping();
public float price();
}
Type here to search
28°C
ENG
18:06
03-08-2023

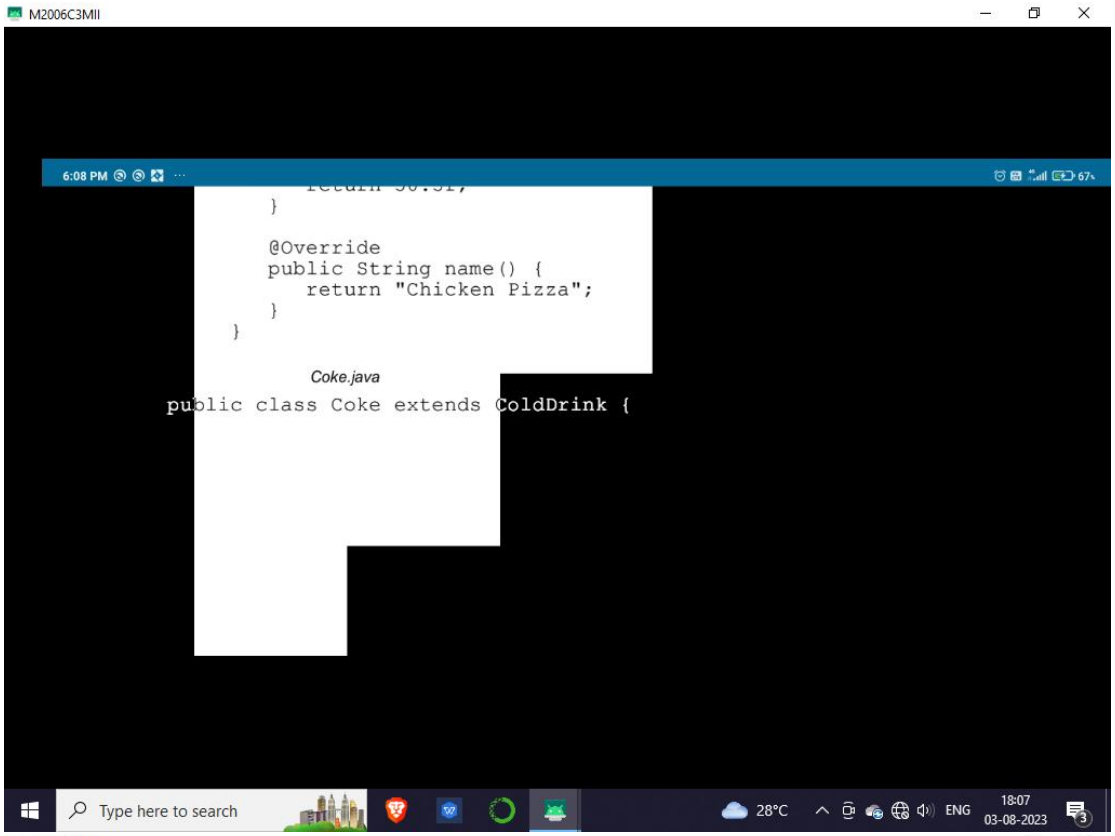

M2006C3MII
6:08 PM
public interface Wrapping {
public String pack();
}
Step 2
Create concreate classes implementing the Wrapping interface.
Wrapper.java
public class Wrapper implements Wrapping {
@Override
public String pack() {
return "Wrapper";
}
}
Bottle.java
public class Bottle implements Wrapping {
Type here to search
28°C
ENG
18:06
03-08-2023

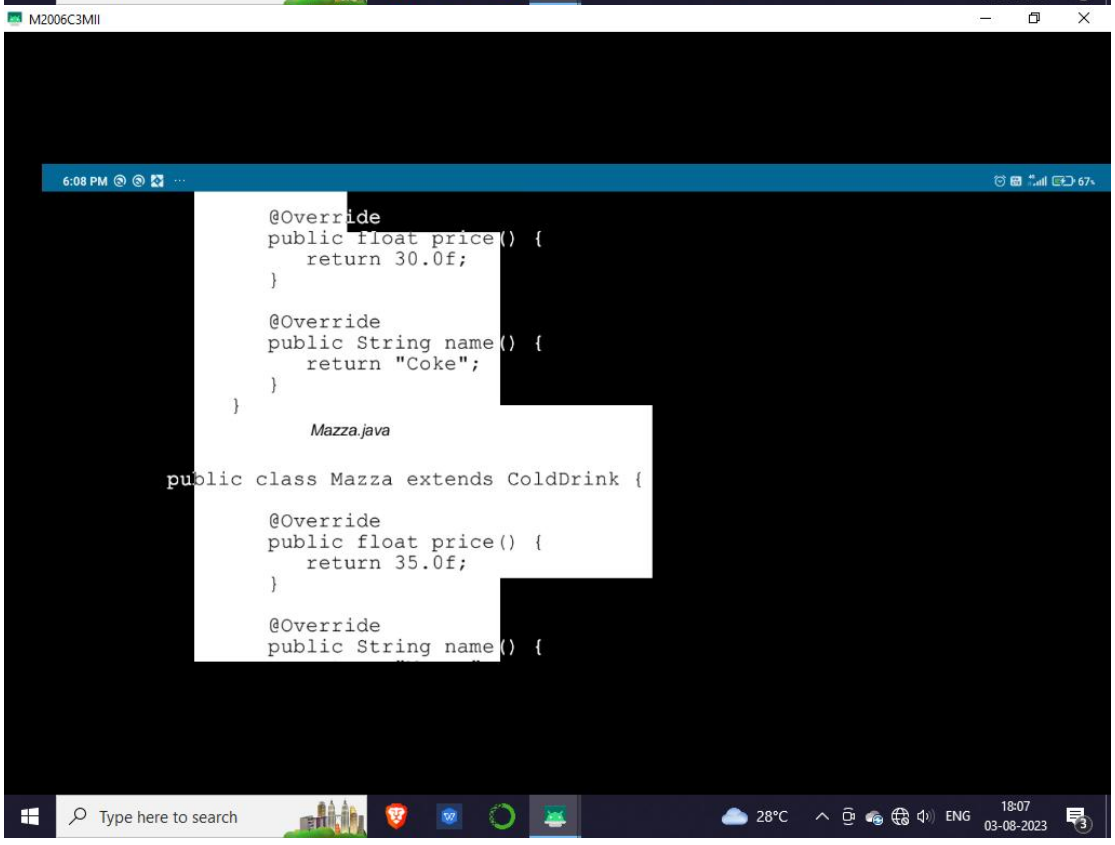

M2006C3MII
6:08 PM
Bottle.java
public class Bottle implements Wrapping {
@Override
public String pack() {
return "Bottle";
}
}
Step 3
Create abstract classes implementing the item interface providing default functionalities.
Pizza.java
public abstract class Pizza implements Item {
@Override
public Wrapping wrapping() {
return new Wrapper();
}
Type here to search
28°C
ENG
18:06
03-08-2023

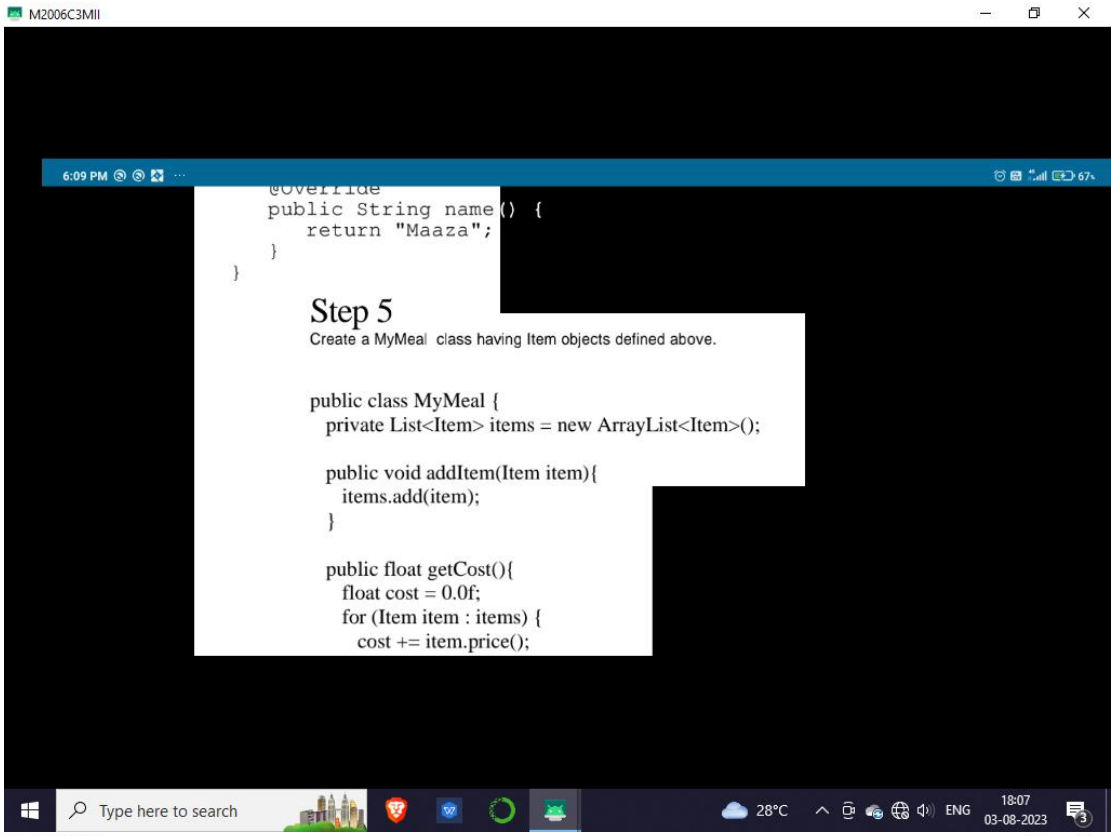

M2006C3MII
6:08 PM
Step 3
Create abstract classes implementing the item interface providing default functionalities.
Pizza.java
public abstract class Pizza implements Item {
@Override
public Wrapping wrapping() {
return new Wrapper();
}
@Override
public abstract float price();
}
ColdDrink.java
public abstract class ColdDrink implements Item {
Type here to search
28°C
ENG
18:06
03-08-2023

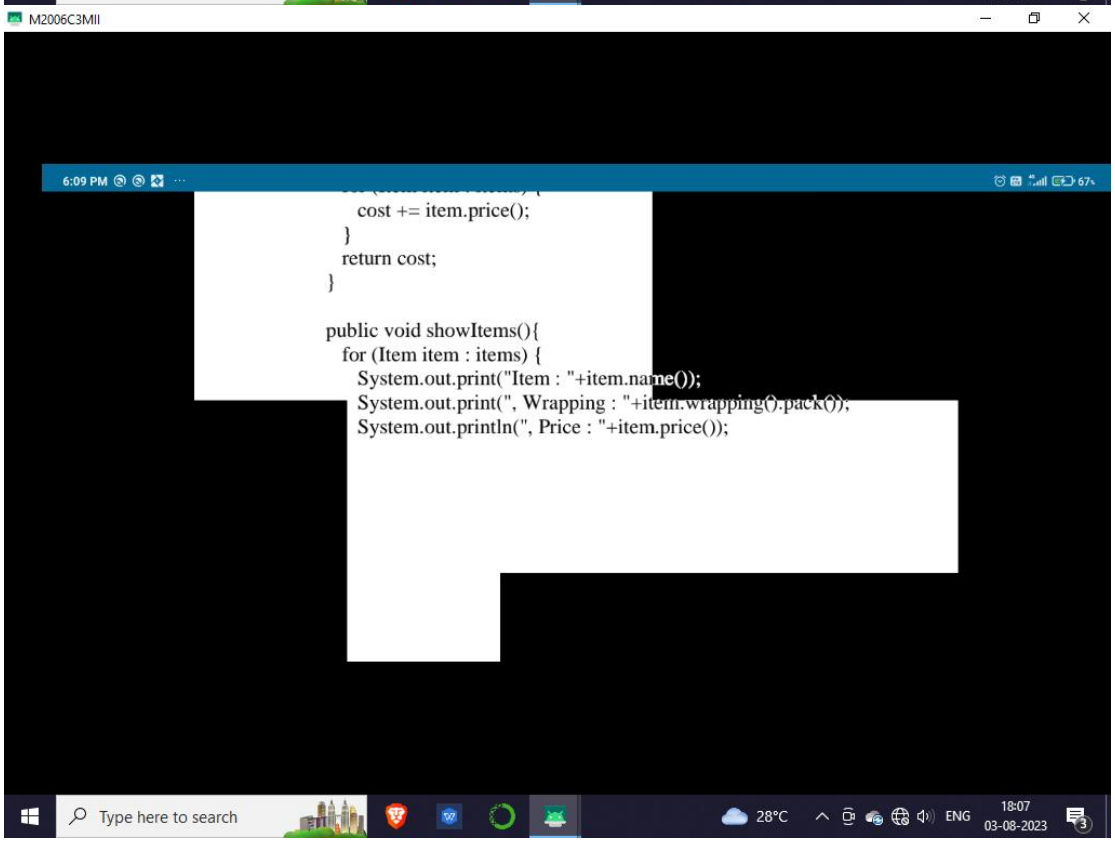

M2006C3MII
6:08 PM
@Override
public Wrapping wrapping() {
return new Bottle();
}
@Override
public abstract float price();
}
Step 4
Create concrete classes extending Pizza and ColdDrink classes
VegPizza.java
public class VegPizza extends Pizza {
Type here to search
28°C
ENG
18:06
03-08-2023

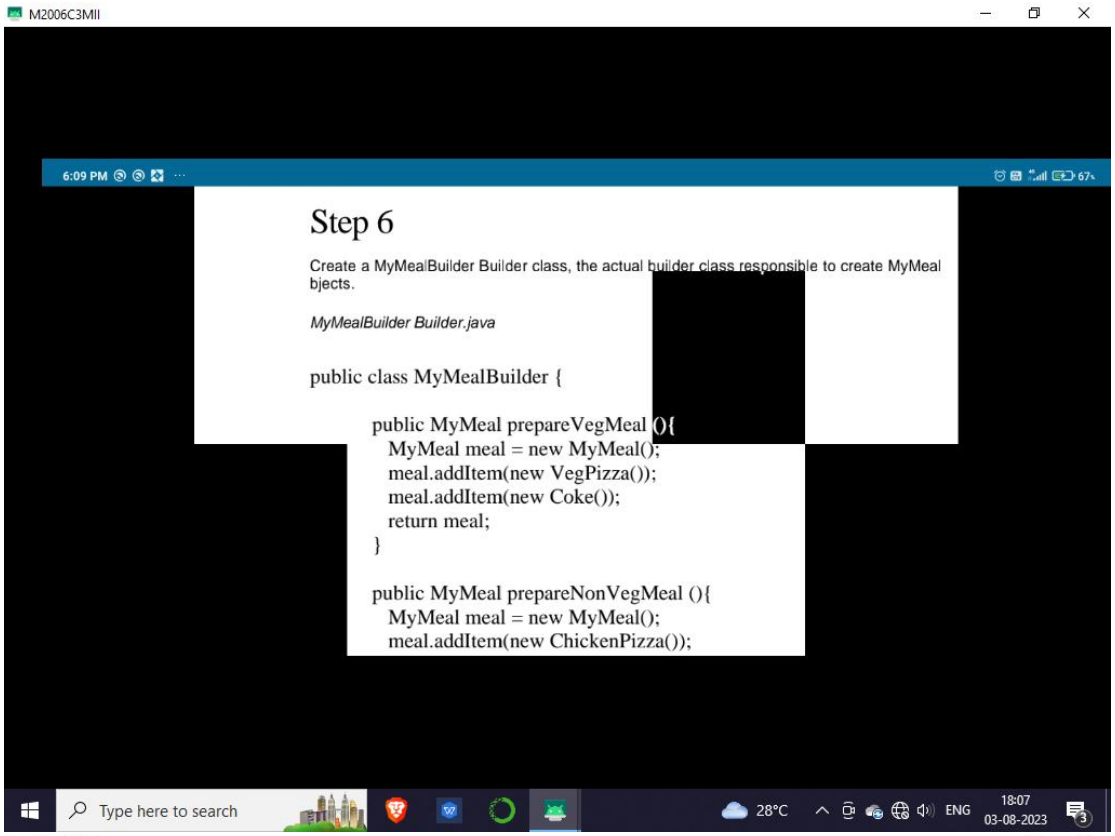

M2006C3MII
6:08 PM
VegPizza.java
public class VegPizza extends Pizza {
@Override
public float price() {
return 25.0f;
}
@Override
public String name() {
return "Veg Pizza";
}
}
ChickenPizza.java
public class ChickenPizza extends Pizza {
@Override
public float price() {
return 50.5f;
}
Type here to search
28°C
ENG
18:06
03-08-2023

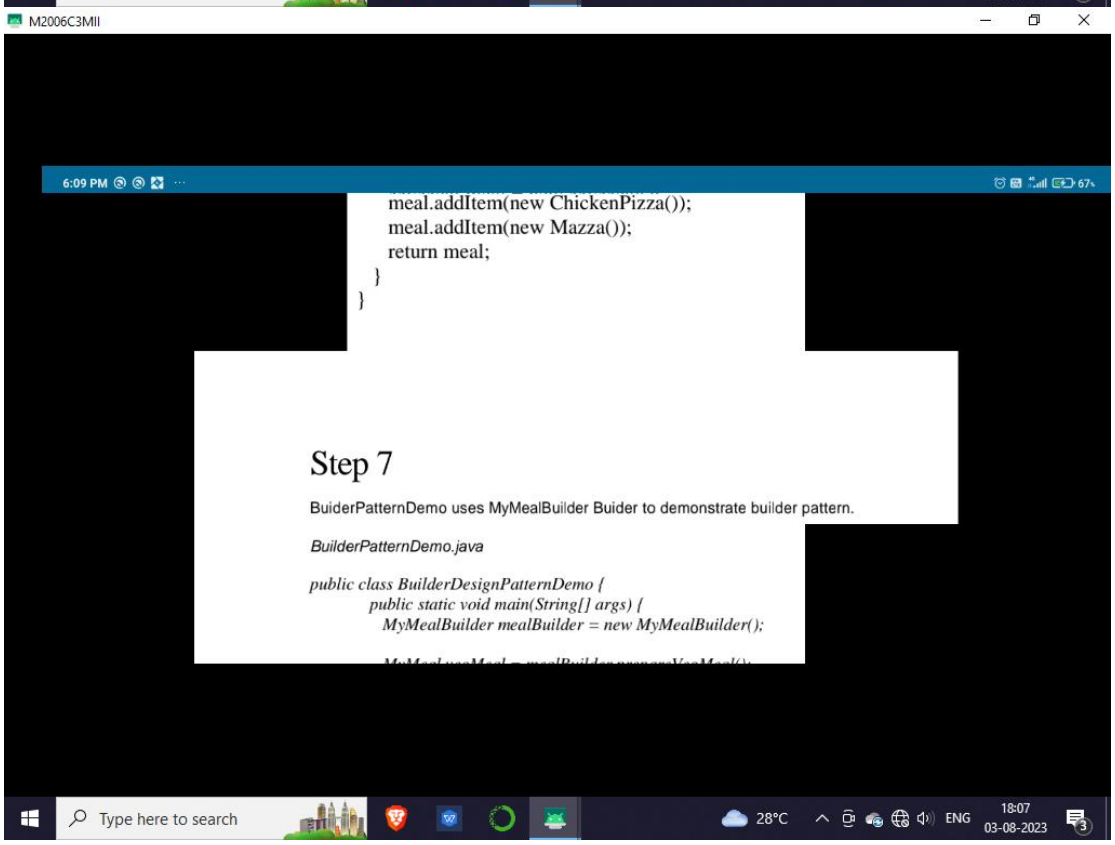

M2006C3MII
6:08 PM
}
@Override
public String name() {
return "Chicken Pizza";
}
}
Coke.java
public class Coke extends ColdDrink {
Type here to search
28°C
ENG
18:07
03-08-2023

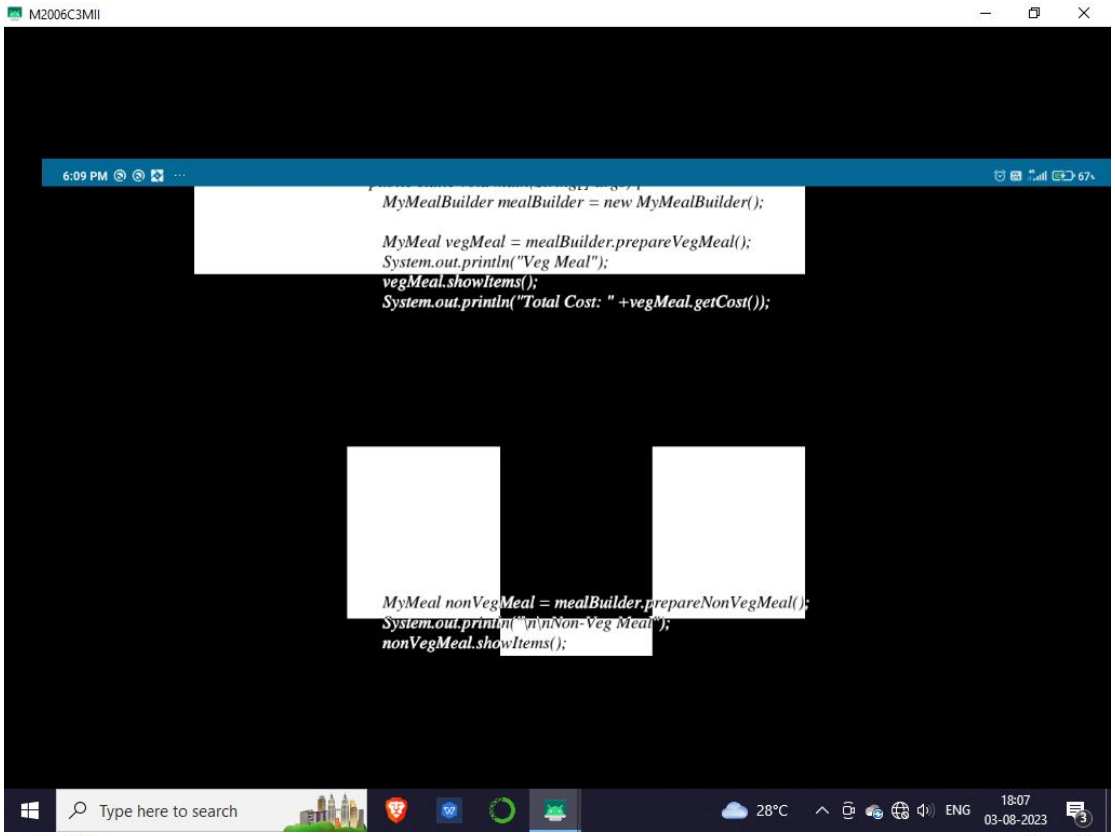

M2006C3MII
6:08 PM
@Override
public float price() {
return 30.0f;
}
@Override
public String name() {
return "Coke";
}
}
Mazza.java
public class Mazza extends ColdDrink {
@Override
public float price() {
return 35.0f;
}
@Override
public String name() {
Type here to search
28°C
ENG
18:07
03-08-2023

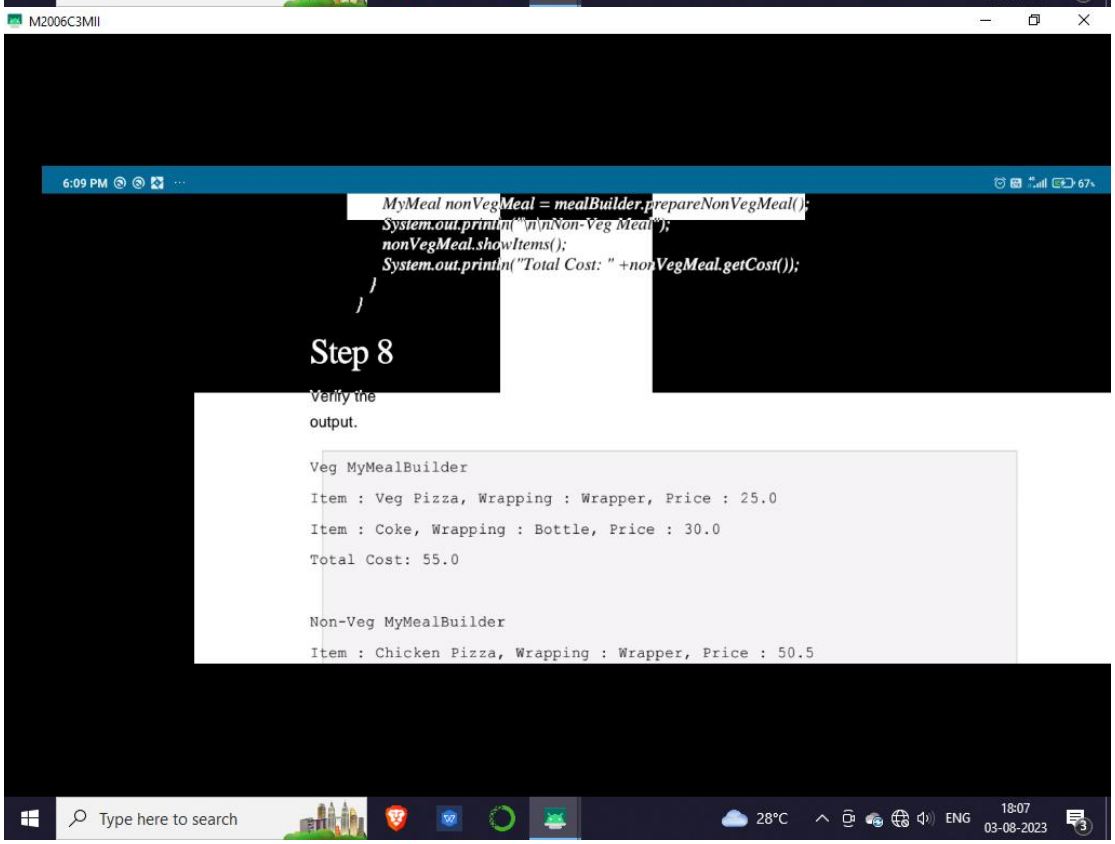

M2006C3MII
6:09 PM
@Override
public String name() {
return "Maaza";
}
}
Step 5
Create a MyMeal class having Item objects defined above.
public class MyMeal {
private List<Item> items = new ArrayList<Item>();
public void addItem(Item item){
items.add(item);
}
public float getCost(){
float cost = 0.0f;
for (Item item : items) {
cost += item.price();
Type here to search
28°C
ENG
18:07
03-08-2023

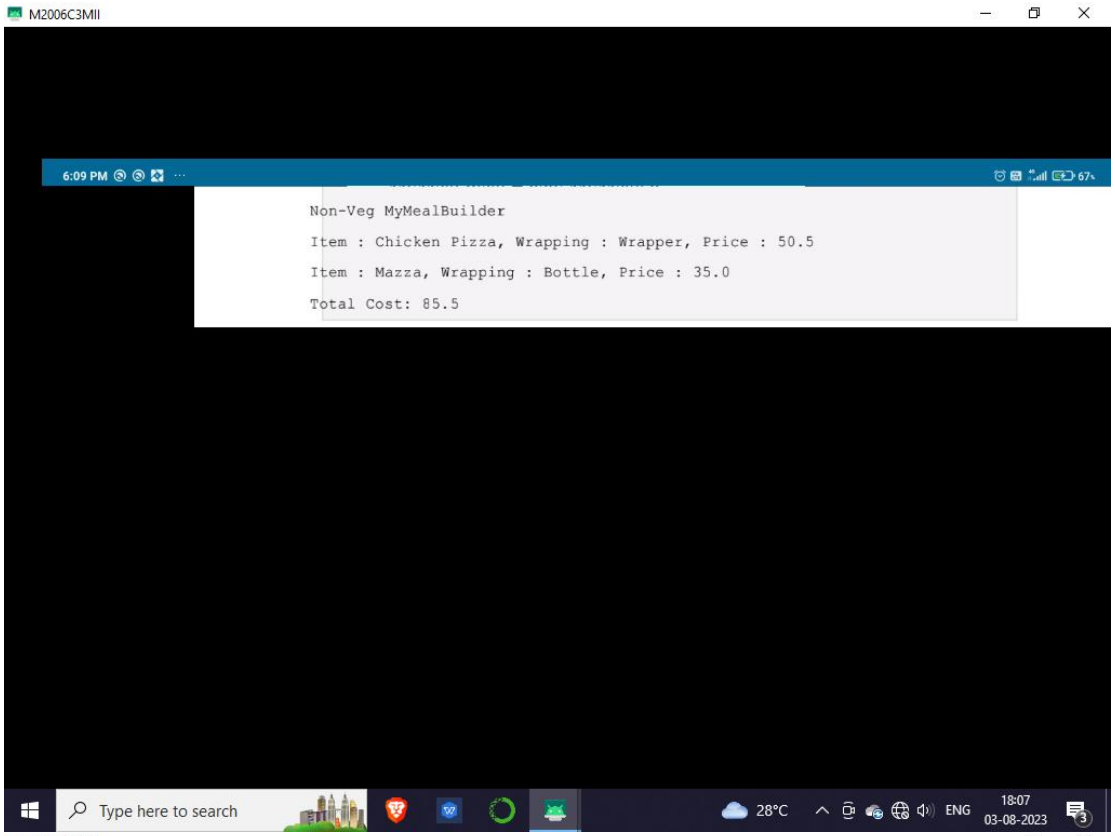

M2006C3MII
6:09 PM
cost += item.price();
}
return cost;
}
public void showItems(){
for (Item item : items) {
System.out.print("Item : "+item.name());
System.out.print(", Wrapping : "+item.wrapping().pack());
System.out.println(", Price : "+item.price());
Type here to search
28°C
ENG
18:07
03-08-2023

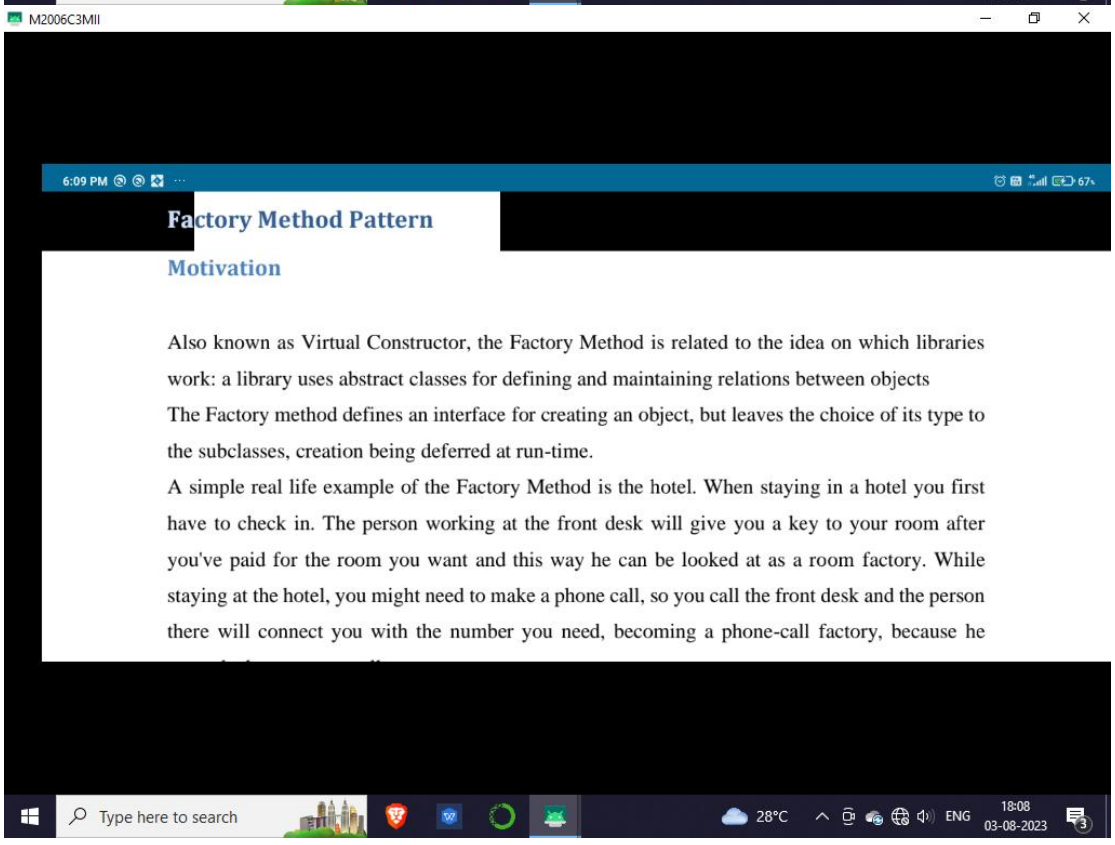

M2006C3MII
6:09 PM
Step 6
Create a MyMealBuilder Builder class, the actual builder class responsible to create MyMeal bjects.
MyMealBuilder Builder.java
public class MyMealBuilder {
public MyMeal prepareVegMeal (){
MyMeal meal = new MyMeal();
meal.addItem(new VegPizza());
meal.addItem(new Coke());
return meal;
}
public MyMeal prepareNonVegMeal (){
MyMeal meal = new MyMeal();
meal.addItem(new ChickenPizza());
Type here to search
28°C
ENG
18:07
03-08-2023


M2006C3MII
6:09 PM
meal.addItem(new ChickenPizza());
meal.addItem(new Mazza());
return meal;
}
}
Step 7
BuiderPatternDemo uses MyMealBuilder Buider to demonstrate builder pattern.
BuilderPatternDemo.java
public class BuilderDesignPatternDemo {
public static void main(String[] args) {
MyMealBuilder mealBuilder = new MyMealBuilder();
Type here to search
28°C
ENG
18:07
03-08-2023


M2006C3MII
6:09 PM
MyMealBuilder mealBuilder = new MyMealBuilder();
MyMeal vegMeal = mealBuilder.prepareVegMeal();
System.out.println("Veg Meal");
vegMeal.showItems();
System.out.println("Total Cost: " +vegMeal.getCost());
MyMeal nonVegMeal = mealBuilder.prepareNonVegMeal();
System.out.println("\n\nNon-Veg Meal");
nonVegMeal.showItems();
Type here to search
28°C
ENG
18:07
03-08-2023


M2006C3MII
6:09 PM
MyMeal nonVegMeal = mealBuilder.prepareNonVegMeal();
System.out.println("\n\nNon-Veg Meal");
nonVegMeal.showItems();
System.out.println("Total Cost: " +nonVegMeal.getCost());
}
}
Step 8
Verify the
output.
Veg MyMealBuilder
Item : Veg Pizza, Wrapping : Wrapper, Price : 25.0
Item : Coke, Wrapping : Bottle, Price : 30.0
Total Cost: 55.0
Non-Veg MyMealBuilder
Item : Chicken Pizza, Wrapping : Wrapper, Price : 50.5
Type here to search
28°C
ENG
18:07
03-08-2023


M2006C3MII
6:09 PM
Non-Veg MyMealBuilder
Item : Chicken Pizza, Wrapping : Wrapper, Price : 50.5
Item : Mazza, Wrapping : Bottle, Price : 35.0
Total Cost: 85.5
Type here to search
28°C
ENG
18:07
03-08-2023


M2006C3MII
6:09 PM
Factory Method Pattern
Motivation
Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects
The Factory method defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.
A simple real life example of the Factory Method is the hotel. When staying in a hotel you first have to check in. The person working at the front desk will give you a key to your room after you've paid for the room you want and this way he can be looked at as a room factory. While staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a phone-call factory, because he
Type here to search
28°C
ENG
18:08
03-08-2023

M2006C3MII

6:09 PM

staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a phone-call factory, because he controls the access to calls, too.

Intent

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate
- Refers to the newly created object through a common interface

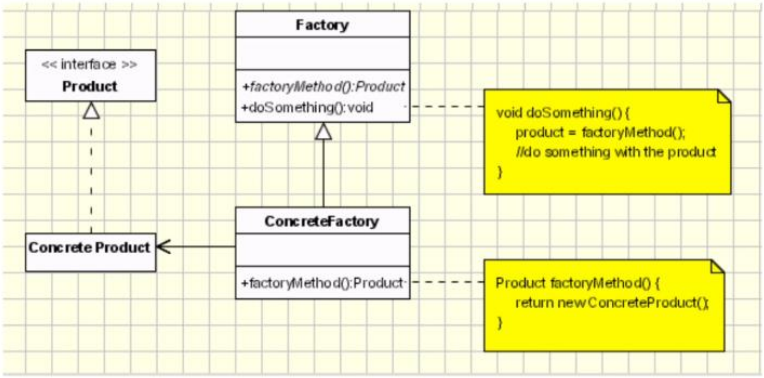
Implementation

The pattern basically works as shown below, in the UML diagram:

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

M2006C3MII

6:10 PM



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

M2006C3MII

The participants classes in this pattern are:

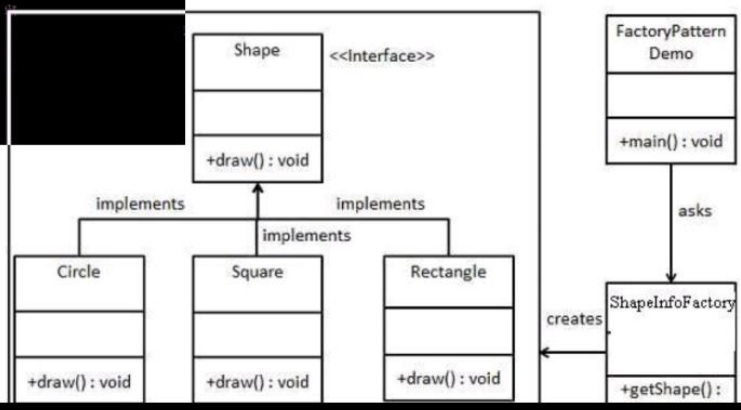
- **Product** defines the interface for objects the factory method creates.
- **ConcreteProduct** implements the Product interface.
- **Creator**(also refered as **Factory** because it creates the Product objects) declares the method **FactoryMethod**, which returns a Product object. May call the generating method for creating Product objects
- **ConcreteCreator** overrides the generating method for creating ConcreteProduct objects

All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Creator class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteCreator. Having this diagram in mind, it is easy for us now to produce the code related to it. Here is how the implementation of the classic Factory method should look:

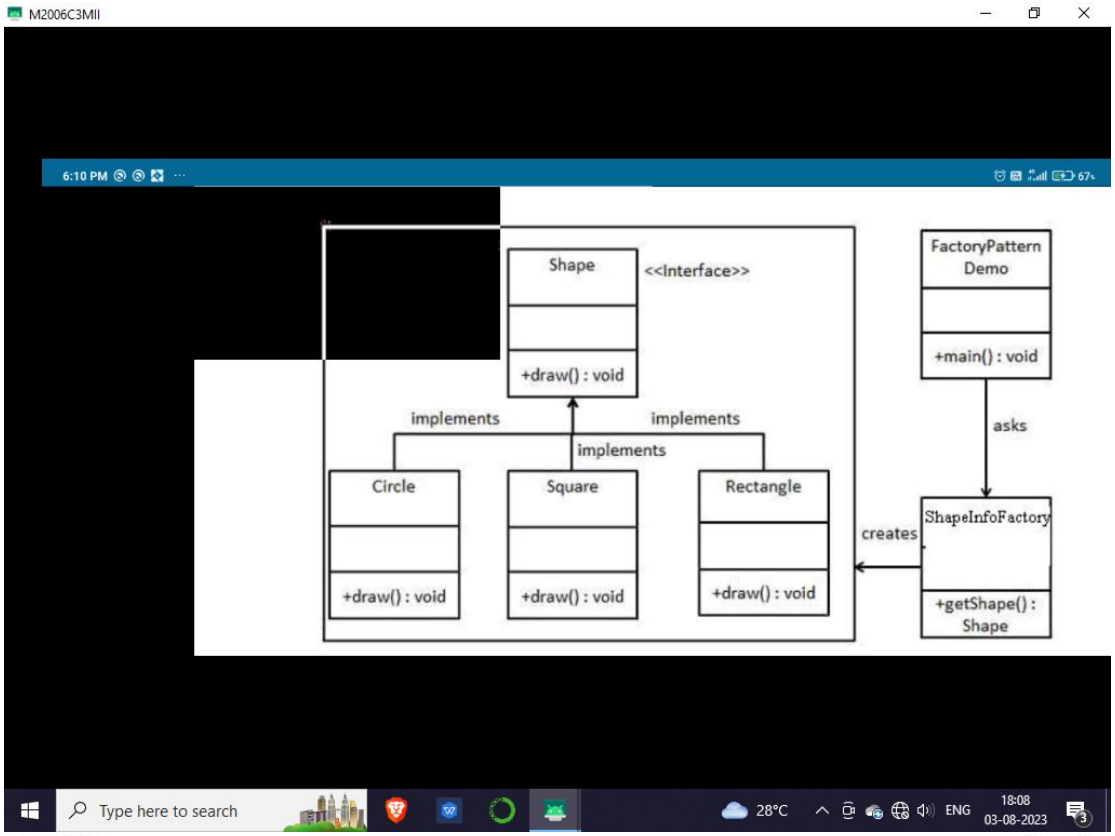
Type here to search 28°C ENG 18:08 03-08-2023

M2006C3MII

Class Diagram



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



Steps

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

Step 1

Create a interface.

Shape.java

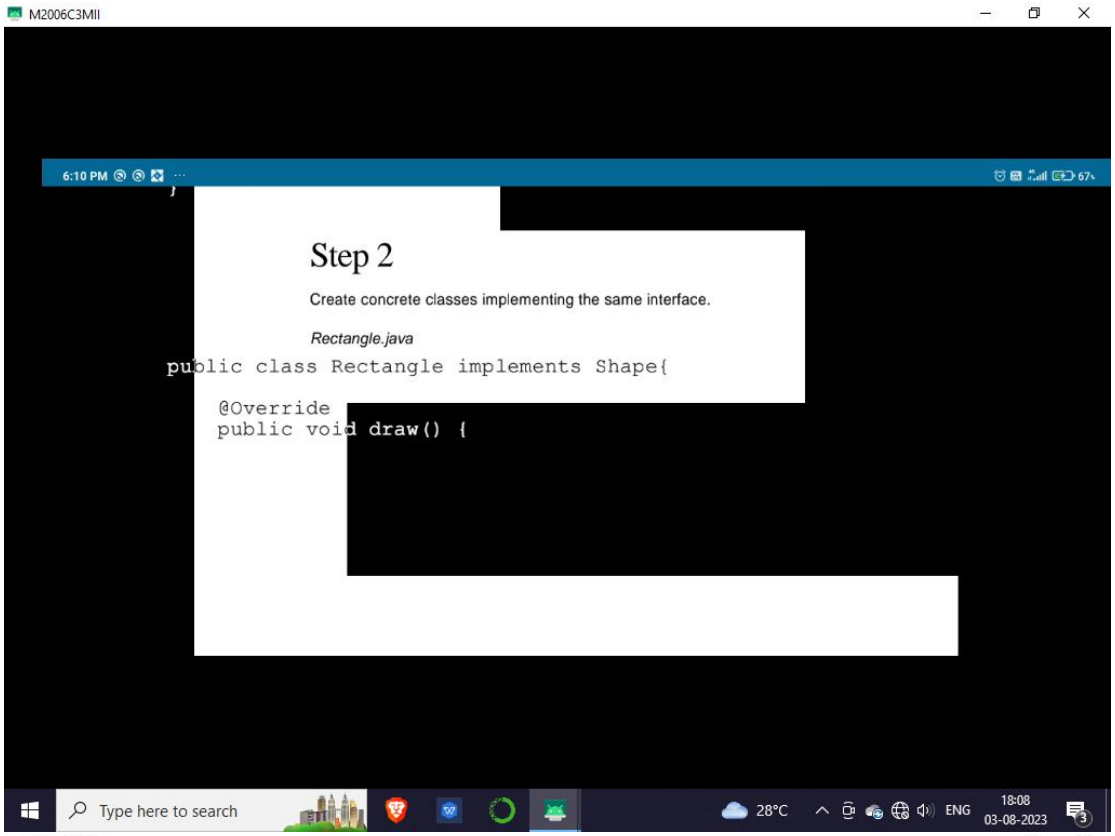
```
public interface Shape {
   void draw();
}
```

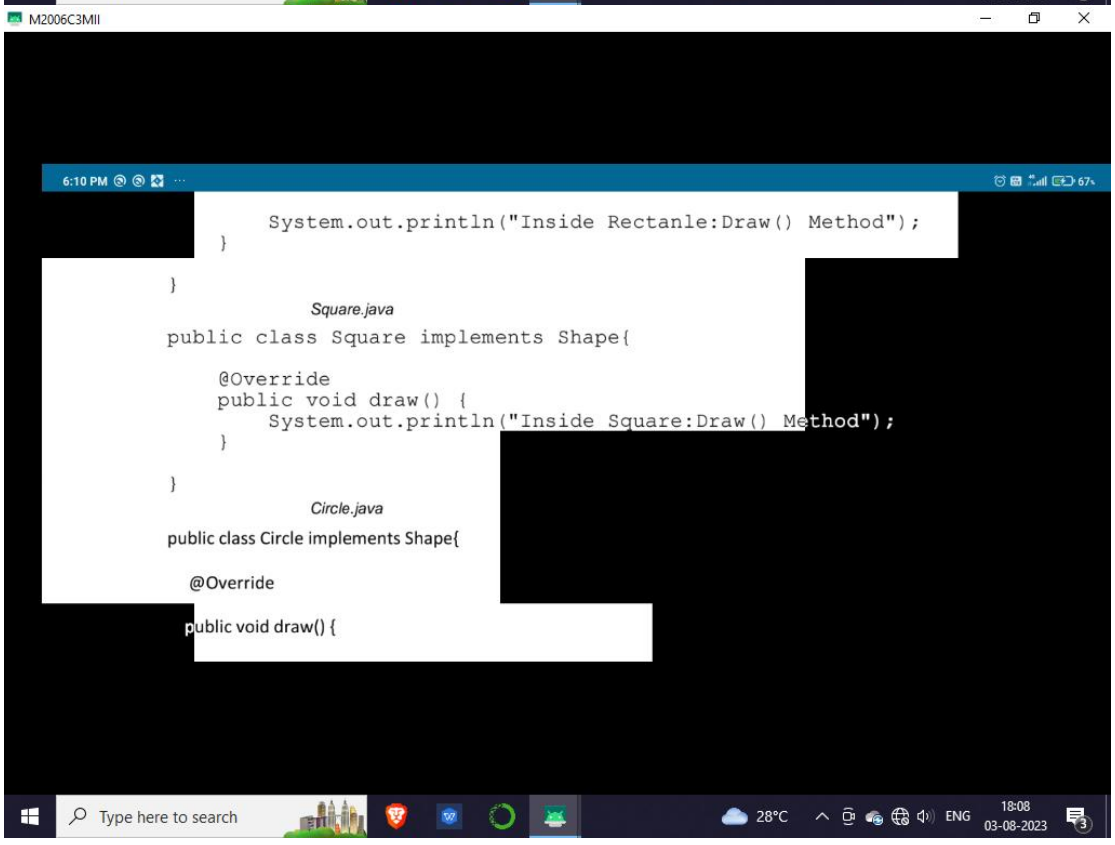
Step 2

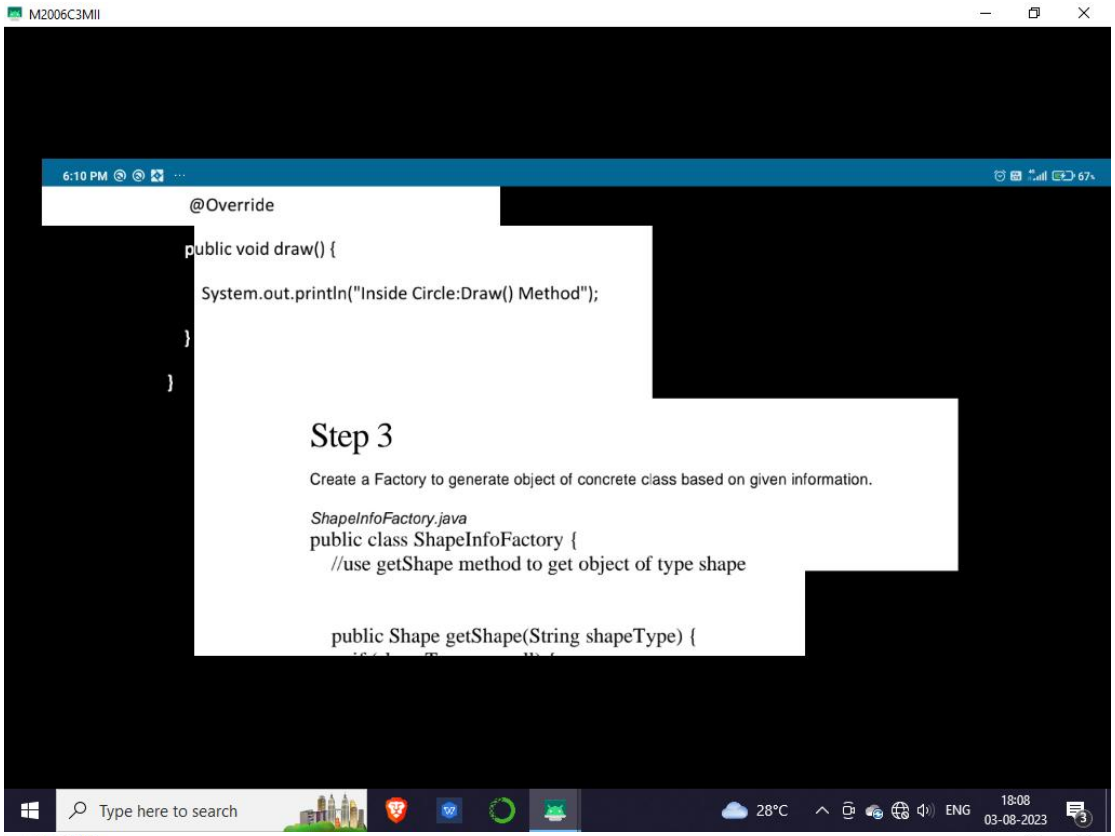
Create concrete classes implementing the same interface.

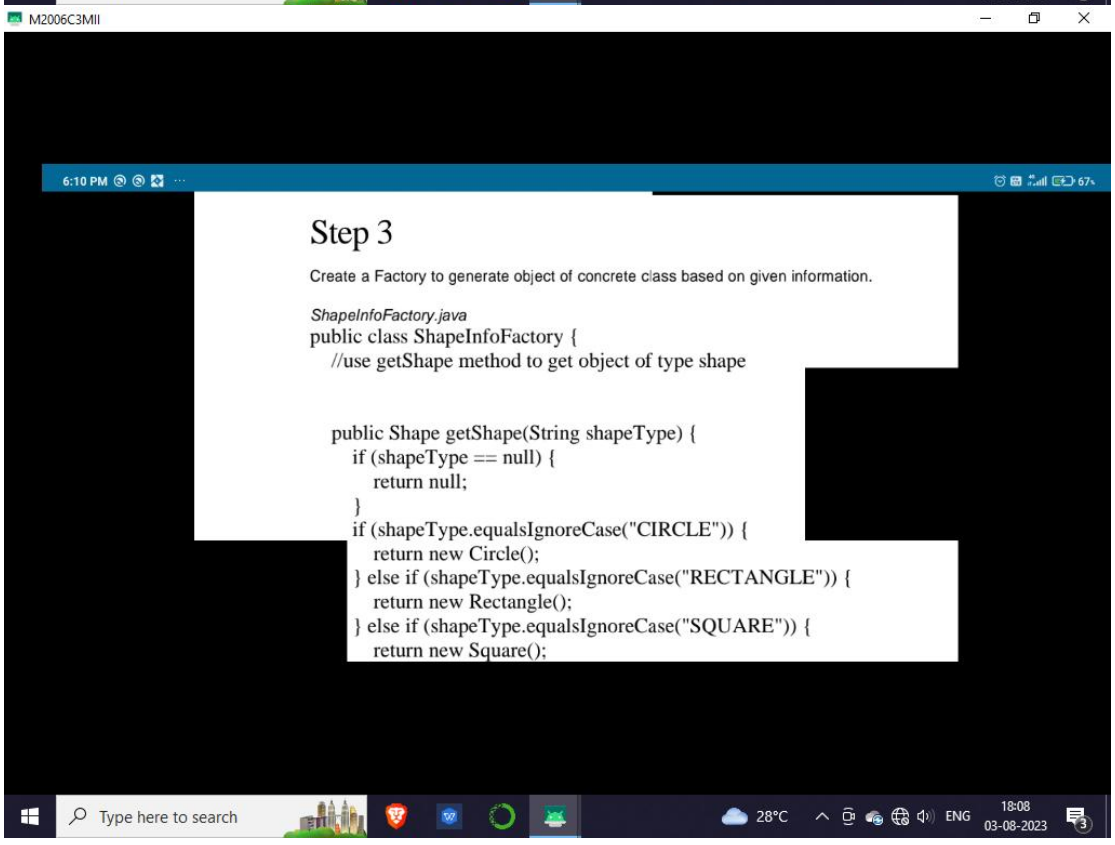
Rectangle.java

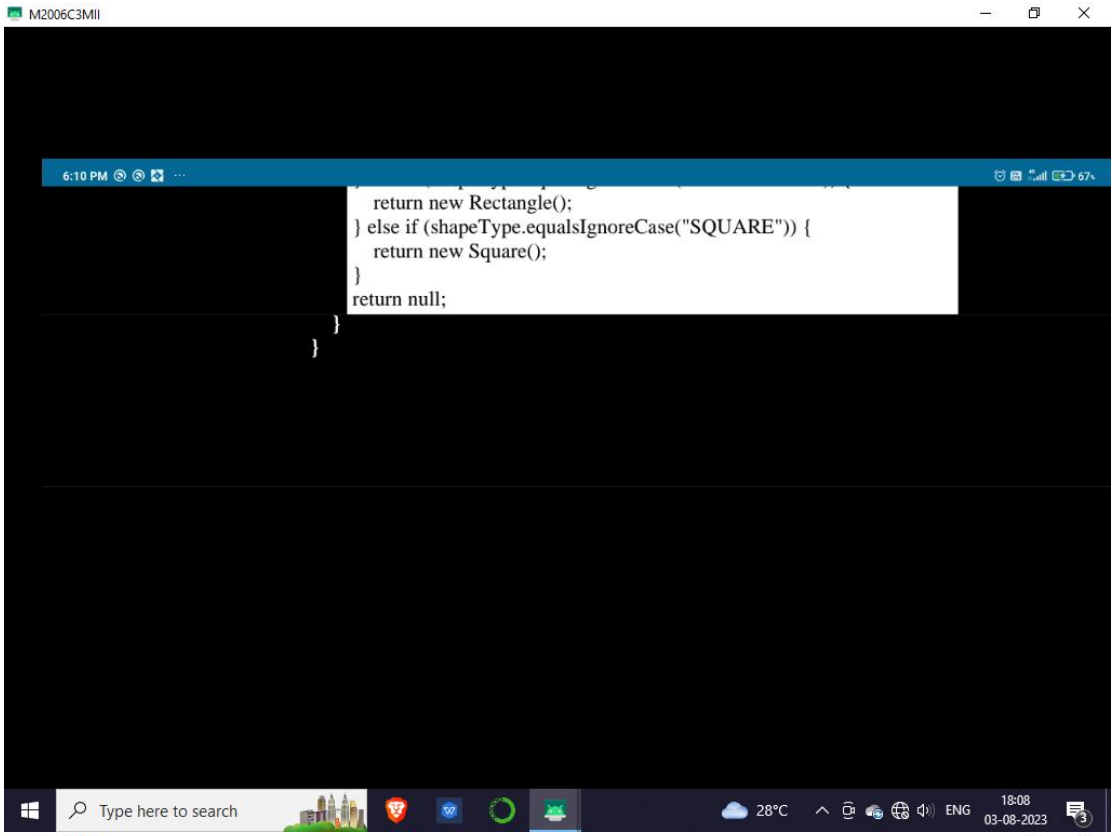
```
public class Rectangle implements Shape{
```

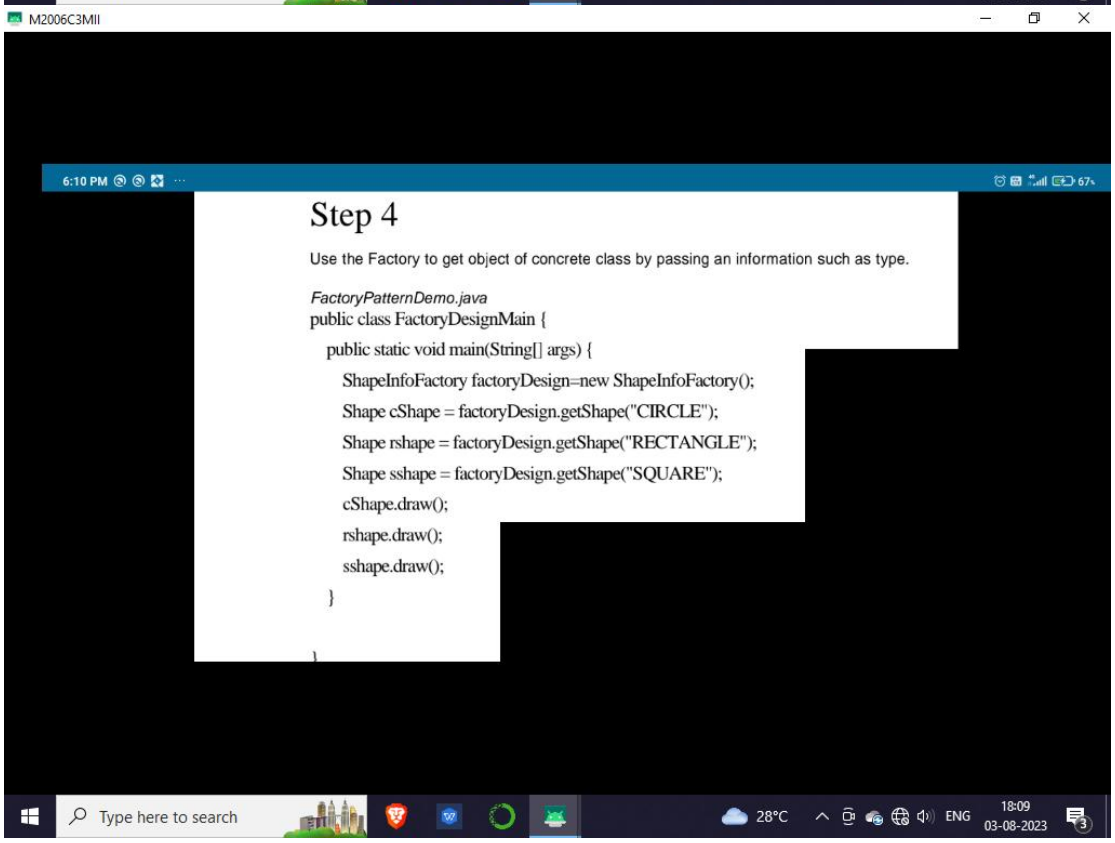

M2006C3MII
6:10 PM
}
Step 2
Create concrete classes implementing the same interface.
Rectangle.java
public class Rectangle implements Shape{
@Override
public void draw() {
Type here to search
28°C
ENG
18:08
03-08-2023


M2006C3MII
6:10 PM
System.out.println("Inside Rectanle:Draw() Method");
}
}
Square.java
public class Square implements Shape{
@Override
public void draw() {
System.out.println("Inside Square:Draw() Method");
}
}
Circle.java
public class Circle implements Shape{
@Override
public void draw() {
Type here to search
28°C
ENG
18:08
03-08-2023


M2006C3MII
6:10 PM
@Override
public void draw() {
System.out.println("Inside Circle:Draw() Method");
}
}
Step 3
Create a Factory to generate object of concrete class based on given information.
ShapeInfoFactory.java
public class ShapeInfoFactory {
//use getShape method to get object of type shape
public Shape getShape(String shapeType) {
Type here to search
28°C
ENG
18:08
03-08-2023


M2006C3MII
6:10 PM
Step 3
Create a Factory to generate object of concrete class based on given information.
ShapeInfoFactory.java
public class ShapeInfoFactory {
//use getShape method to get object of type shape
public Shape getShape(String shapeType) {
if (shapeType == null) {
return null;
}
if (shapeType.equalsIgnoreCase("CIRCLE")) {
return new Circle();
} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
return new Rectangle();
} else if (shapeType.equalsIgnoreCase("SQUARE")) {
return new Square();
Type here to search
28°C
ENG
18:08
03-08-2023


M2006C3MII
6:10 PM
return new Rectangle();
} else if (shapeType.equalsIgnoreCase("SQUARE")) {
return new Square();
}
return null;
}
}
Type here to search
28°C
ENG
18:08
03-08-2023


M2006C3MII
6:10 PM
Step 4
Use the Factory to get object of concrete class by passing an information such as type.
FactoryPatternDemo.java
public class FactoryDesignMain {
public static void main(String[] args) {
ShapeInfoFactory factoryDesign=new ShapeInfoFactory();
Shape cShape = factoryDesign.getShape("CIRCLE");
Shape rshape = factoryDesign.getShape("RECTANGLE");
Shape sshape = factoryDesign.getShape("SQUARE");
cShape.draw();
rshape.draw();
sshape.draw();
}
}
Type here to search
28°C
ENG
18:09
03-08-2023

M2006C3MII

6:10 PM

```
        sshape.draw();
    }

}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Key Points:

Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

Key Points:

The factory method is one of the most used and one of the more robust design patterns. There are only few points which have to be considered when you implement a factory method.

When you design an application just think if you really need it a factory to create objects. Maybe using it will bring unnecessary complexity in your application. Anyway if you have many object

Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

of the same base type and you manipulate them mostly as abstract objects, then you need a factory.

If you're code should have a lot of code like the following, reconsider it.

```
if (genericProduct typeof ConcreteProduct)
        ((ConcreteProduct)genericProduct).doSomeConcreteOperation();
```

Prototype Pattern

Intent

- specifying the kind of objects to create using a prototypical instance
- creating new objects by copying this prototype

Motivation

Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

- specifying the kind of objects to create using a prototypical instance
- creating new objects by copying this prototype

Motivation

Today's programming is all about costs. Saving is a big issue when it comes to using computer resources, so programmers are doing their best to find ways of improving the performance When we talk about object creation we can find a better way to have new objects: cloning. To this idea one particular design pattern is related: rather than creation it uses cloning. If the cost of creating a new object is large and creation is resource intensive, we clone the object.

.

Implementation

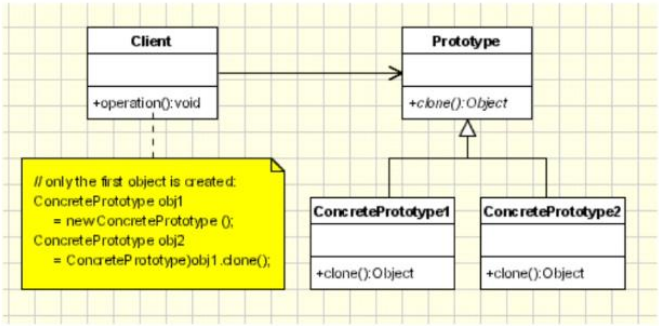
The pattern uses abstract classes, as we will see below and only three types of classes making its implementation rather easy.

Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

The pattern uses abstract classes, as we will see below and only three types of classes making its implementation rather easy.



Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

The classes participating to the Prototype Pattern are:

- **Client** - creates a new object by asking a prototype to clone itself.
- **Prototype** - declares an interface for cloning itself.
- **ConcretePrototype** - implements the operation for cloning itself.

The process of cloning starts with an initialized and instantiated class. The Client asks for a new object of that type and sends the request to the Prototype class. A ConcretePrototype, depending of the type of object is needed, will handle the cloning through the Clone() method, making a new instance of itself.

Here is a sample code for the Prototype pattern:

Applicability & Examples with Source Code:

Use Prototype Pattern when a system should be independent of how its products are created,

Type here to search 28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

Applicability & Examples with Source Code:

Use Prototype Pattern when a system should be independent of how its products are created, composed, and represented, and:

- Classes to be instantiated are specified at run-time
- Avoiding the creation of a factory hierarchy is needed
- It is more convenient to copy an existing instance than to create a new one.

Example 1

Prototype pattern notion is to create duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves implementing a prototype interface which tells to

Type here to search

28°C ENG 18:09 03-08-2023

M2006C3MII

6:11 PM

Example 1

Prototype pattern notion is to create duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

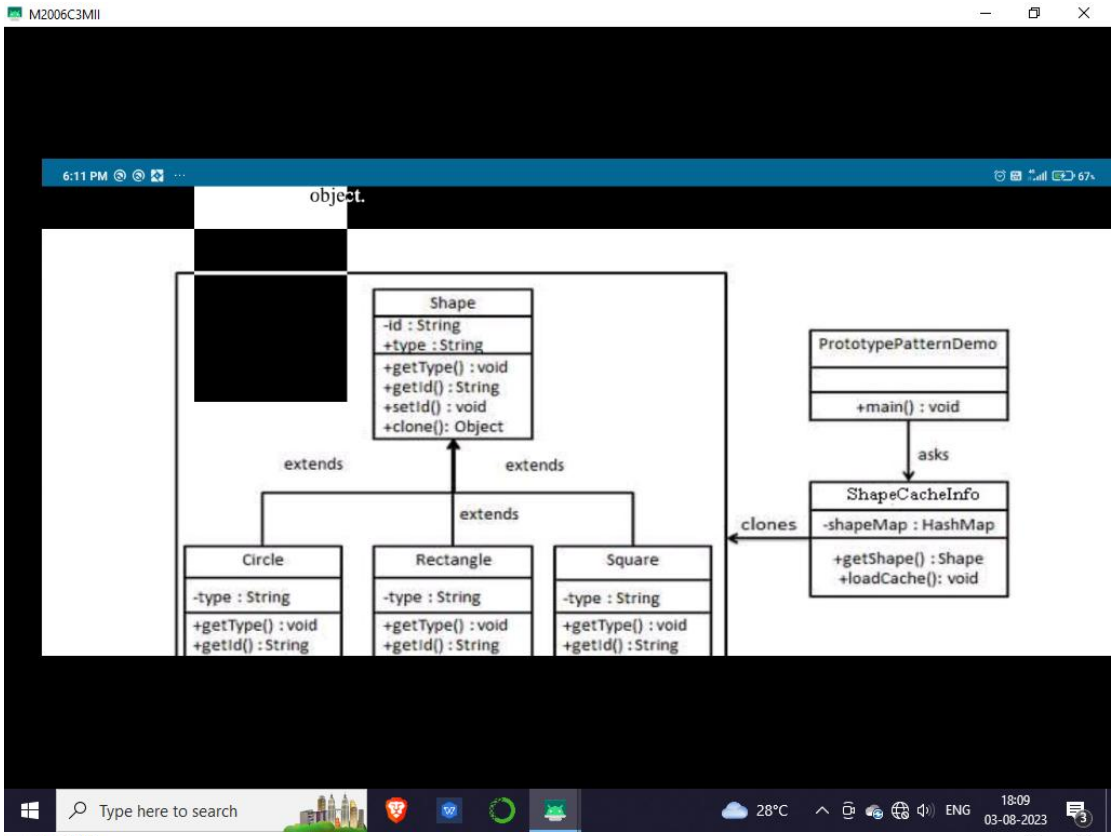
This pattern involves implementing a prototype interface which tells to create a replica of the current object. This pattern is used when creation of object directly is very costly. Consider an example; a object is to be created after a costly database operation. We can cache the object, returns its replica on next request and update the database as and when needed thus reducing database calls (Server trips).

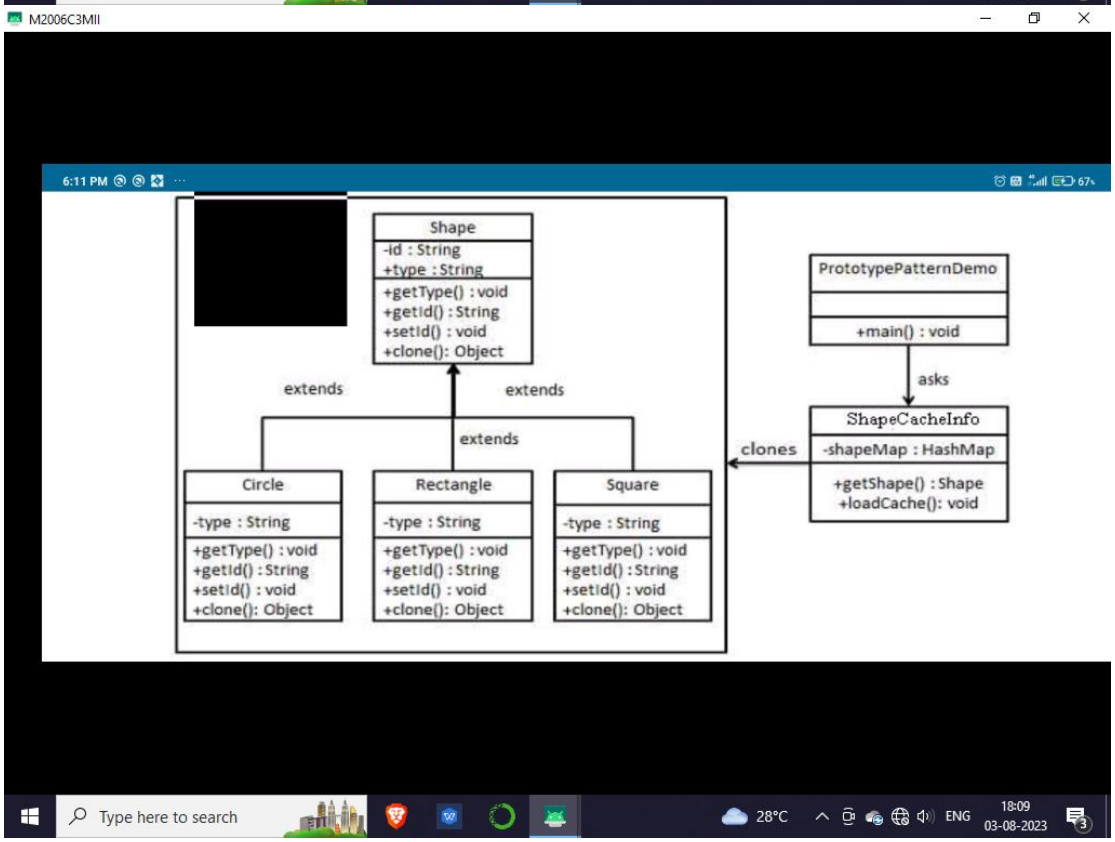
Implementation

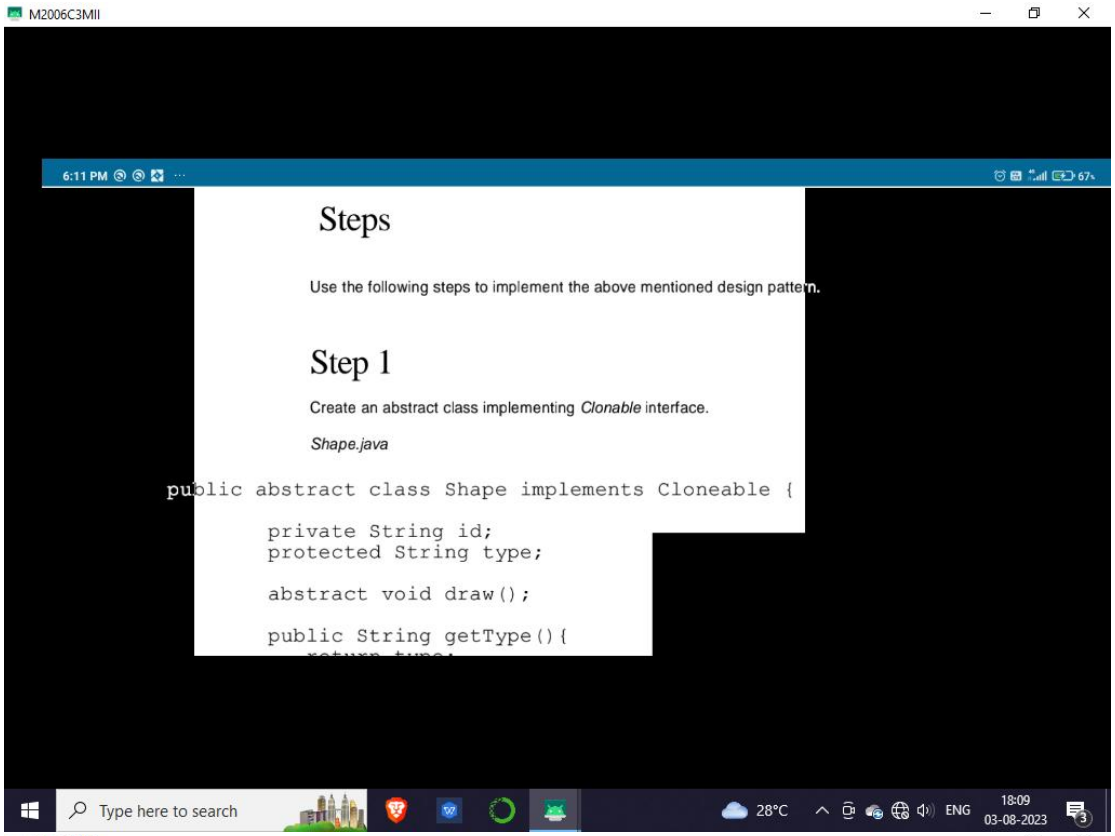
We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCacheInfo* is defined as a next step which stores shape objects in a *Hashtable* and returns their replica when requested. *PrototypPatternDemo*, class will use *ShapeCacheInfo* class to get a *Shape*

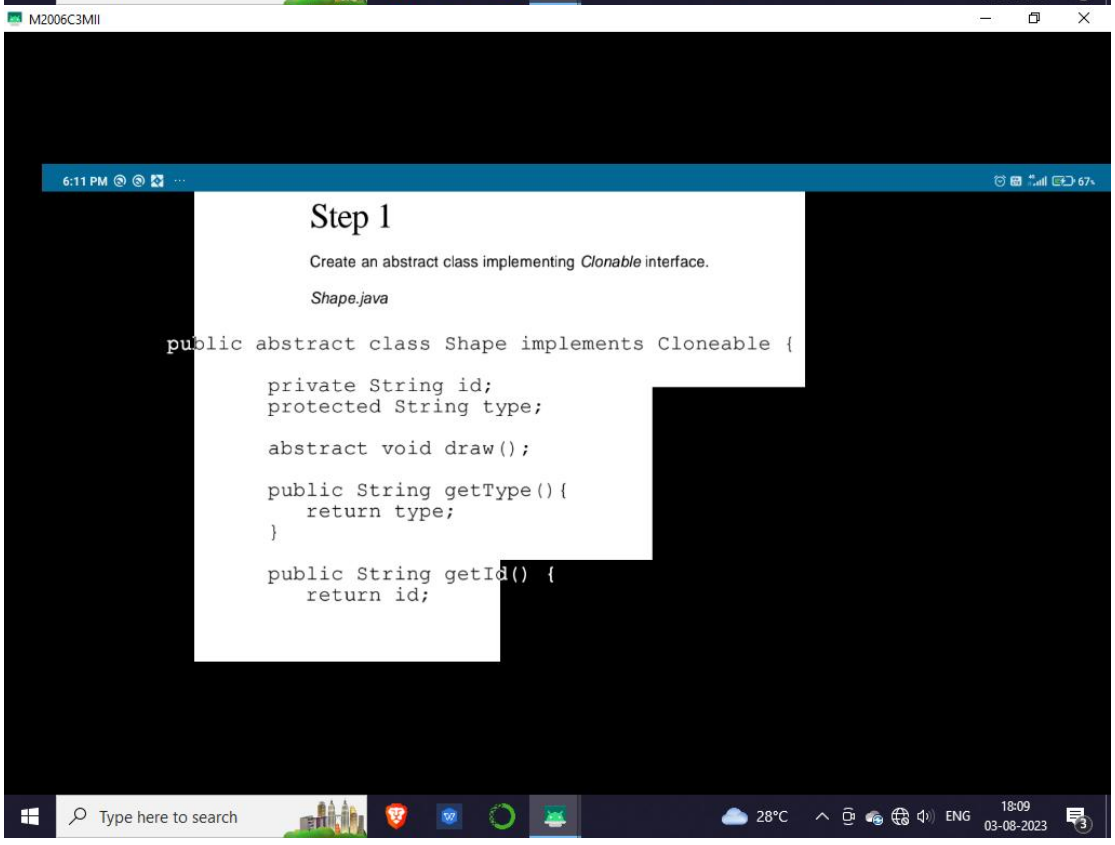
Type here to search

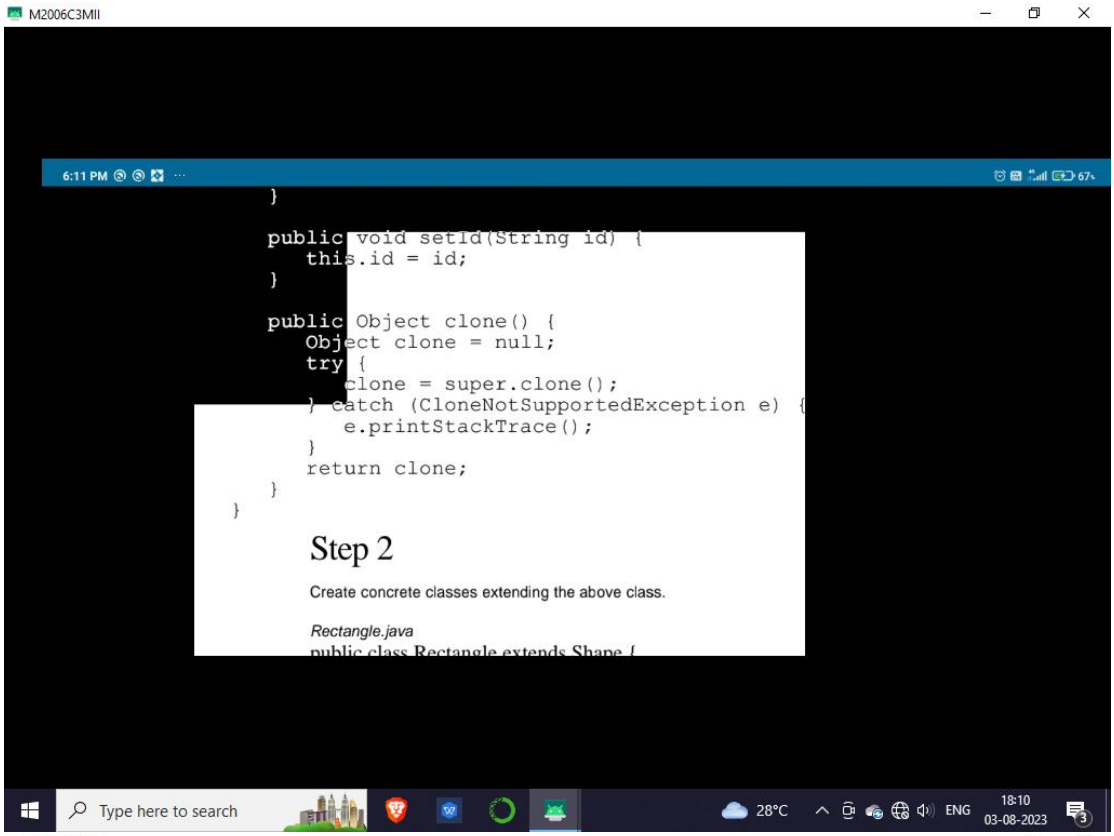
28°C ENG 18:09 03-08-2023

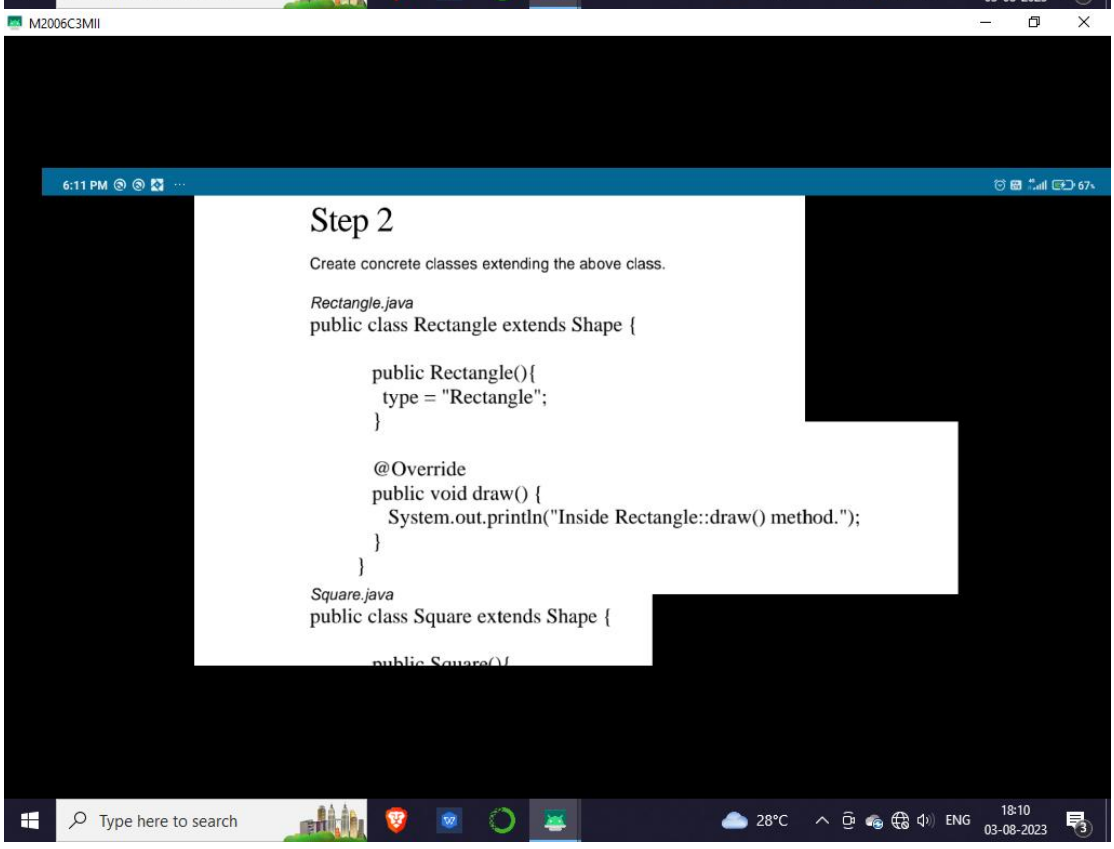

M2006C3MII
6:11 PM
object.
Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
Rectangle
-type : String
+getType() : void
+getId() : String
Square
-type : String
+getType() : void
+getId() : String
PrototypePatternDemo
+main() : void
asks
ShapeCacheInfo
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void
clones
Type here to search
28°C
ENG
18:09
03-08-2023

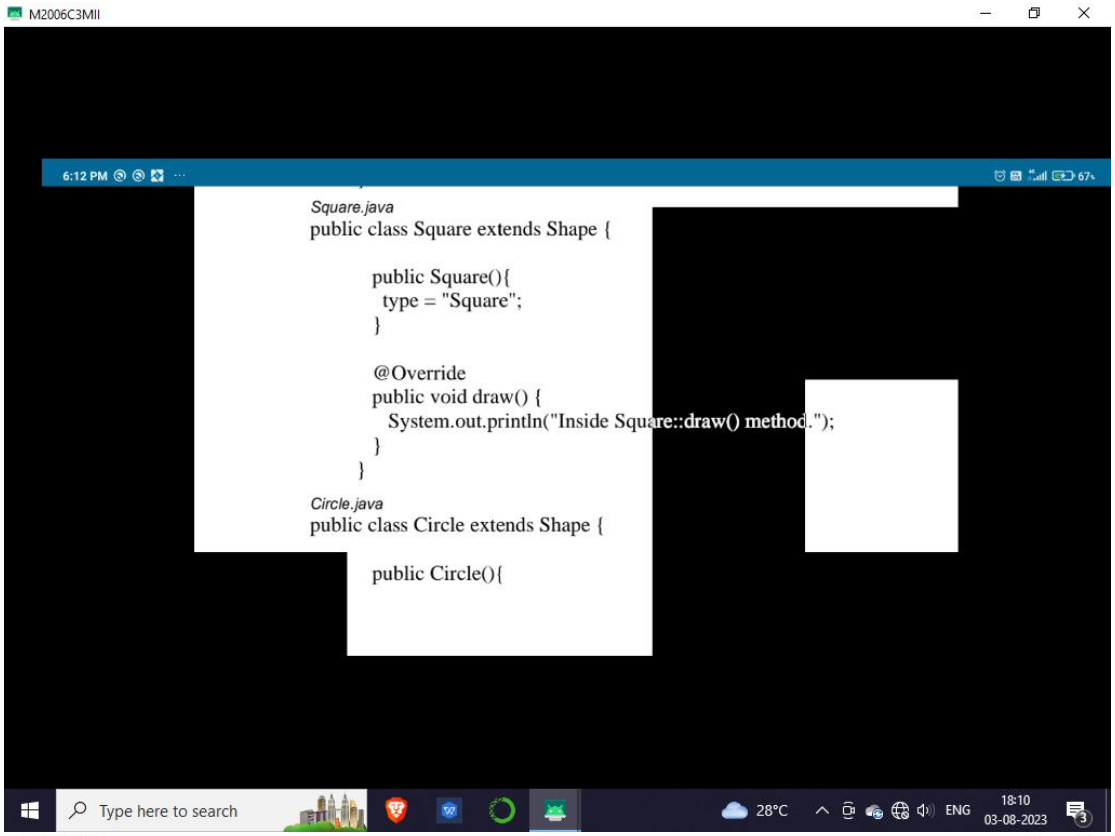

M2006C3MII
6:11 PM
Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
PrototypePatternDemo
+main() : void
asks
ShapeCacheInfo
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void
clones
Type here to search
28°C
ENG
18:09
03-08-2023

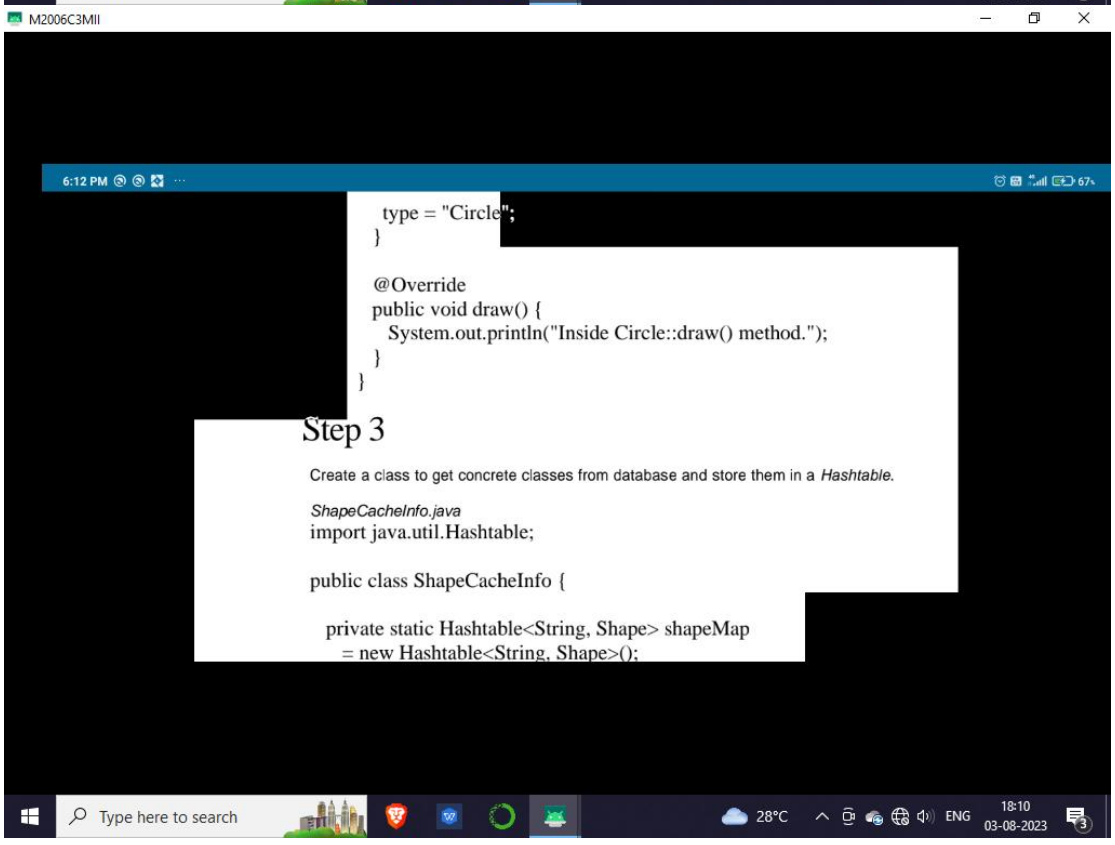

M2006C3MII
6:11 PM
Steps
Use the following steps to implement the above mentioned design pattern.
Step 1
Create an abstract class implementing Clonable interface.
Shape.java
public abstract class Shape implements Cloneable {
private String id;
protected String type;
abstract void draw();
public String getType(){
Type here to search
28°C
ENG
18:09
03-08-2023

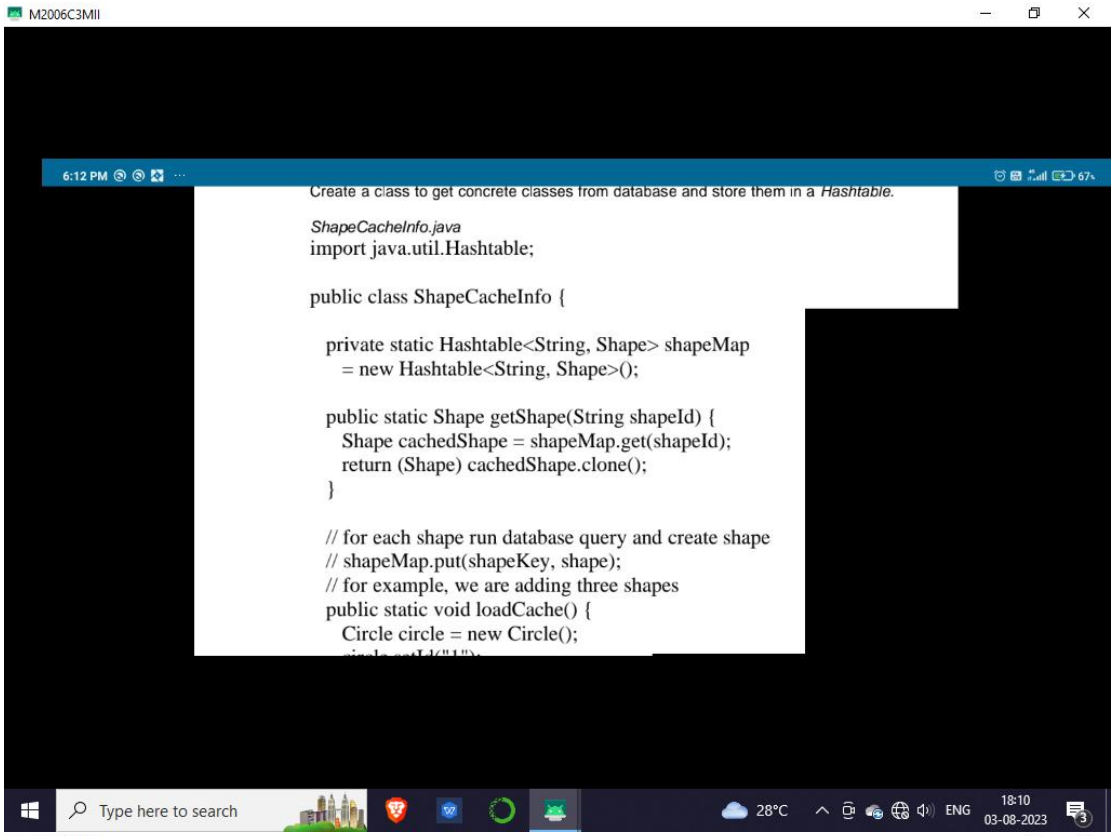

M2006C3MII
6:11 PM
Step 1
Create an abstract class implementing Clonable interface.
Shape.java
public abstract class Shape implements Cloneable {
private String id;
protected String type;
abstract void draw();
public String getType(){
return type;
}
public String getId() {
return id;
Type here to search
28°C
ENG
18:09
03-08-2023

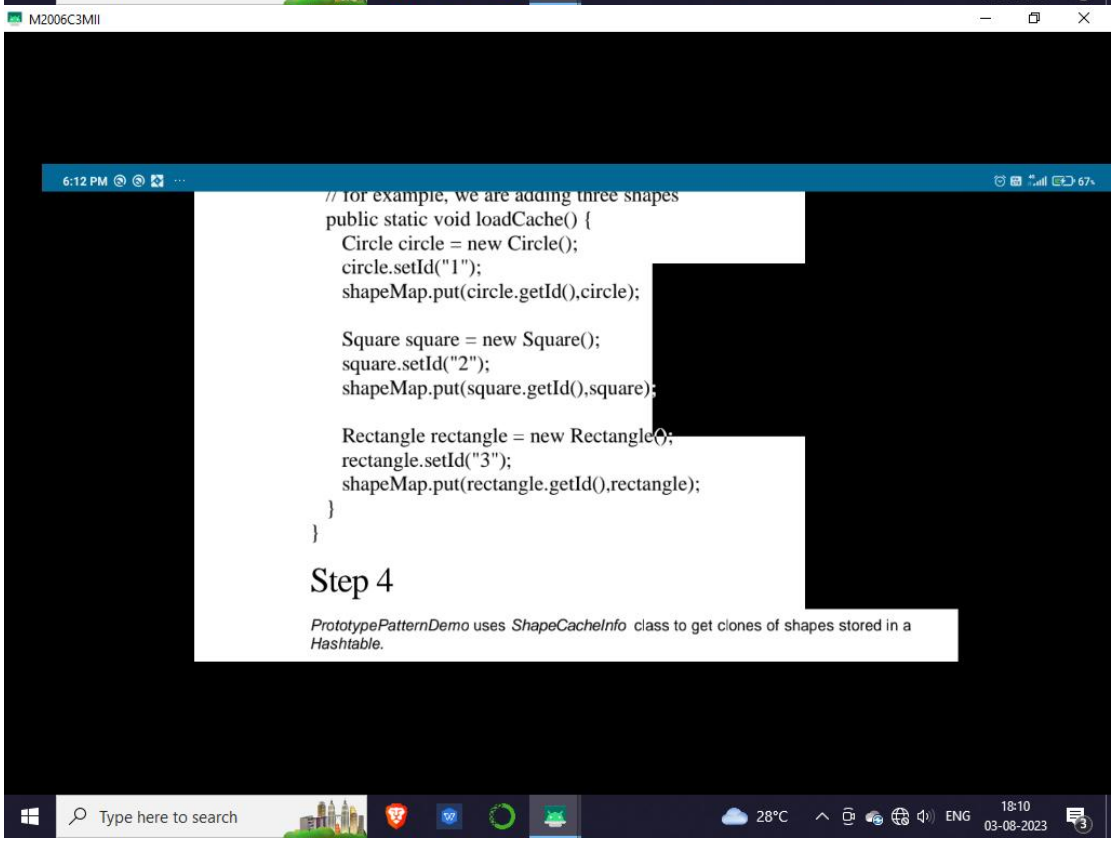

M2006C3MII
6:11 PM
}
public void setId(String id) {
this.id = id;
}
public Object clone() {
Object clone = null;
try {
clone = super.clone();
} catch (CloneNotSupportedException e) {
e.printStackTrace();
}
return clone;
}
}
Step 2
Create concrete classes extending the above class.
Rectangle.java
public class Rectangle extends Shape {
Type here to search
28°C
ENG
18:10
03-08-2023

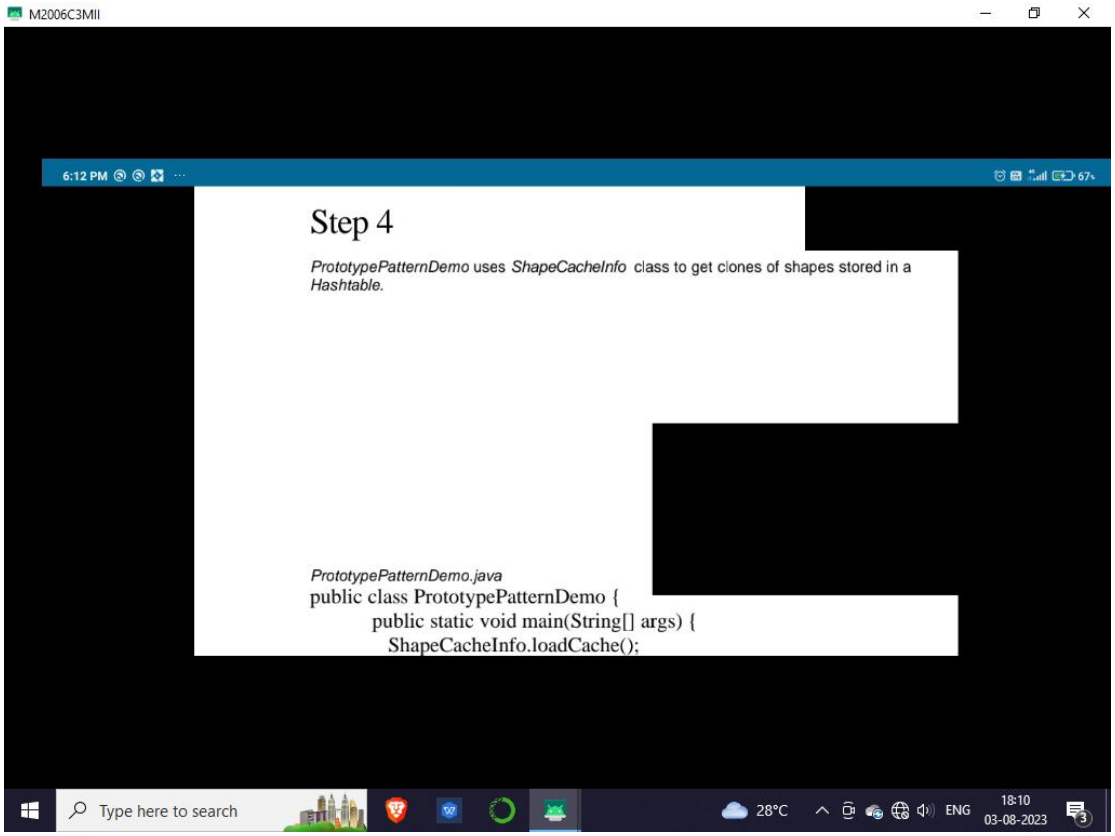

M2006C3MII
6:11 PM
Step 2
Create concrete classes extending the above class.
Rectangle.java
public class Rectangle extends Shape {
public Rectangle(){
type = "Rectangle";
}
@Override
public void draw() {
System.out.println("Inside Rectangle::draw() method.");
}
}
Square.java
public class Square extends Shape {
Type here to search
28°C
ENG
18:10
03-08-2023

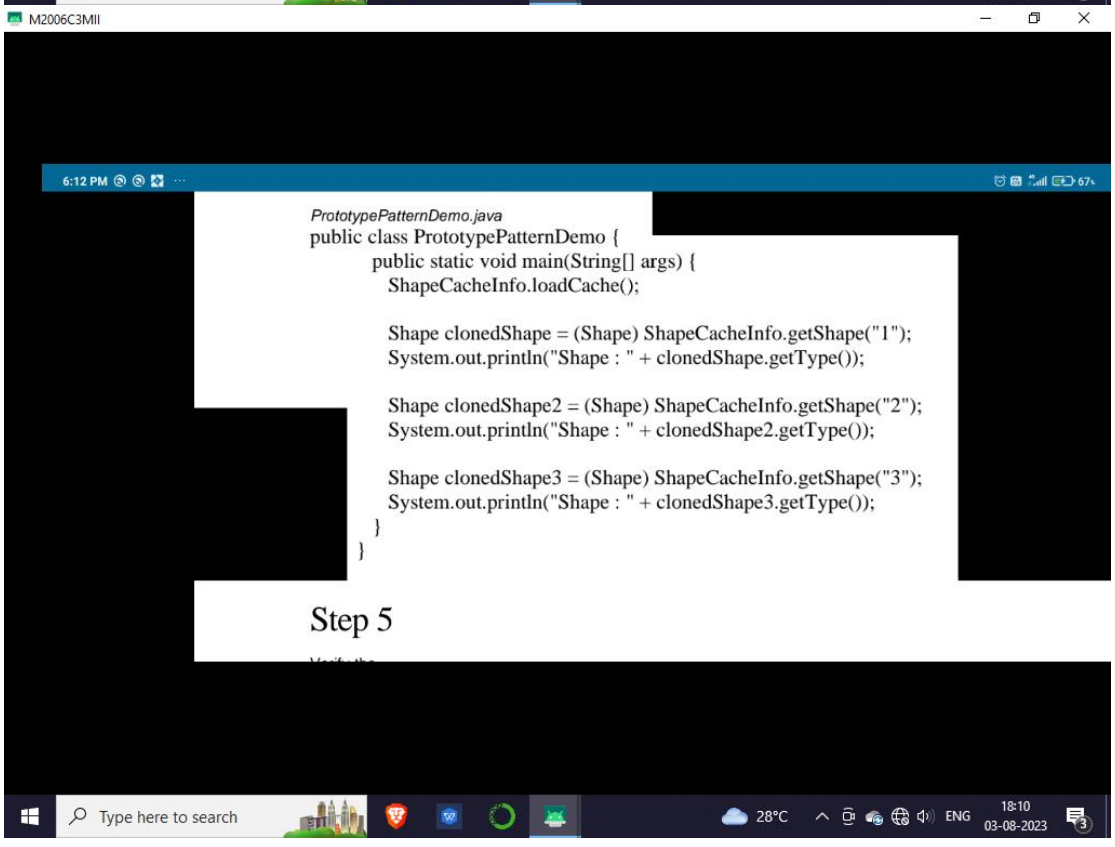

M2006C3MII
6:12 PM
Square.java
public class Square extends Shape {
public Square(){
type = "Square";
}
@Override
public void draw() {
System.out.println("Inside Square::draw() method.");
}
}
Circle.java
public class Circle extends Shape {
public Circle(){
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
type = "Circle";
}
@Override
public void draw() {
System.out.println("Inside Circle::draw() method.");
}
}
Step 3
Create a class to get concrete classes from database and store them in a Hashtable.
ShapeCacheInfo.java
import java.util.Hashtable;
public class ShapeCacheInfo {
private static Hashtable<String, Shape> shapeMap
= new Hashtable<String, Shape>();
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
Create a class to get concrete classes from database and store them in a Hashtable.
ShapeCacheInfo.java
import java.util.Hashtable;
public class ShapeCacheInfo {
private static Hashtable<String, Shape> shapeMap
= new Hashtable<String, Shape>();
public static Shape getShape(String shapeId) {
Shape cachedShape = shapeMap.get(shapeId);
return (Shape) cachedShape.clone();
}
// for each shape run database query and create shape
// shapeMap.put(shapeKey, shape);
// for example, we are adding three shapes
public static void loadCache() {
Circle circle = new Circle();
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
// for example, we are adding three shapes
public static void loadCache() {
Circle circle = new Circle();
circle.setId("1");
shapeMap.put(circle.getId(),circle);
Square square = new Square();
square.setId("2");
shapeMap.put(square.getId(),square);
Rectangle rectangle = new Rectangle();
rectangle.setId("3");
shapeMap.put(rectangle.getId(),rectangle);
}
}
Step 4
PrototypePatternDemo uses ShapeCacheInfo class to get clones of shapes stored in a Hashtable.
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
Step 4
PrototypePatternDemo uses ShapeCacheInfo class to get clones of shapes stored in a Hashtable.
PrototypePatternDemo.java
public class PrototypePatternDemo {
public static void main(String[] args) {
ShapeCacheInfo.loadCache();
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
PrototypePatternDemo.java
public class PrototypePatternDemo {
public static void main(String[] args) {
ShapeCacheInfo.loadCache();
Shape clonedShape = (Shape) ShapeCacheInfo.getShape("1");
System.out.println("Shape : " + clonedShape.getType());
Shape clonedShape2 = (Shape) ShapeCacheInfo.getShape("2");
System.out.println("Shape : " + clonedShape2.getType());
Shape clonedShape3 = (Shape) ShapeCacheInfo.getShape("3");
System.out.println("Shape : " + clonedShape3.getType());
}
}
Step 5
Type here to search
28°C
ENG
18:10
03-08-2023

M2006C3MII

6:12 PM

Step 5

Verify the output.

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

Example 2

In building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the haze map using the same objects: wall, door, passage, room... The Prototype pattern is useful in this case because instead of hard coding (using new operation) the room, door, passage and wall objects that get instantiated, CreateMaze

Type here to search 28°C ENG 18:10 03-08-2023

M2006C3MII

6:12 PM

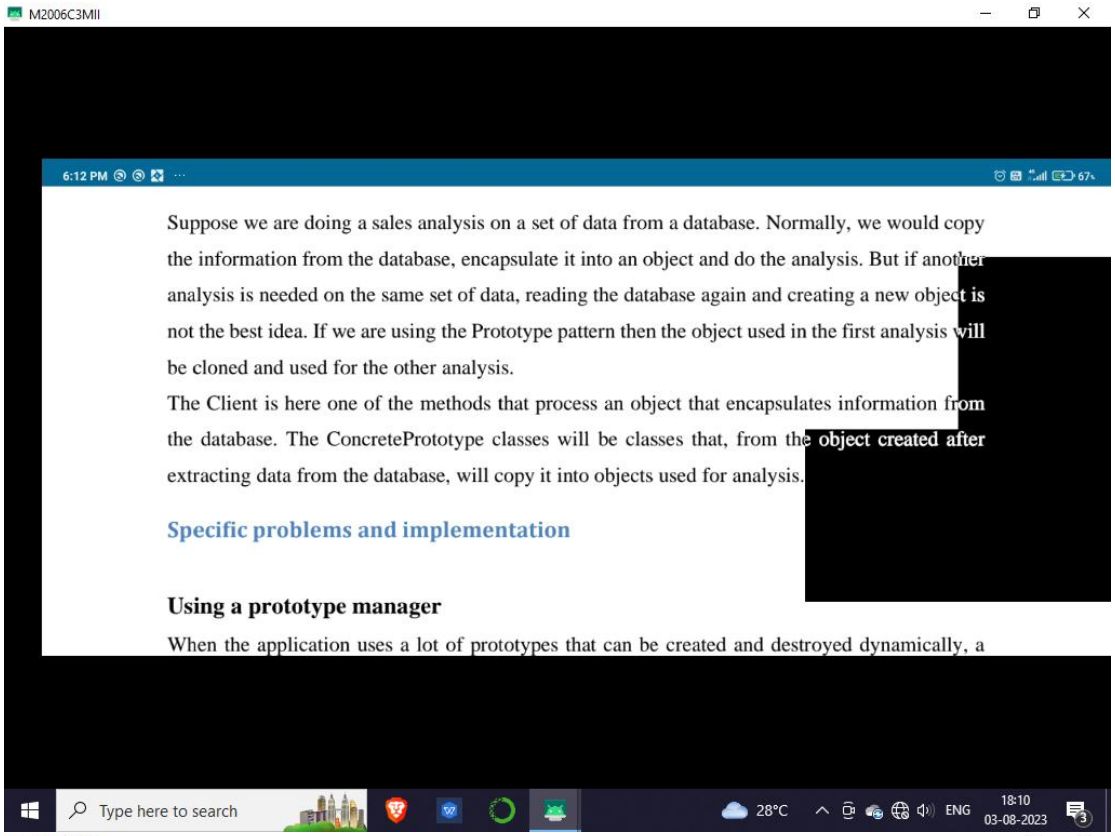
Example 2

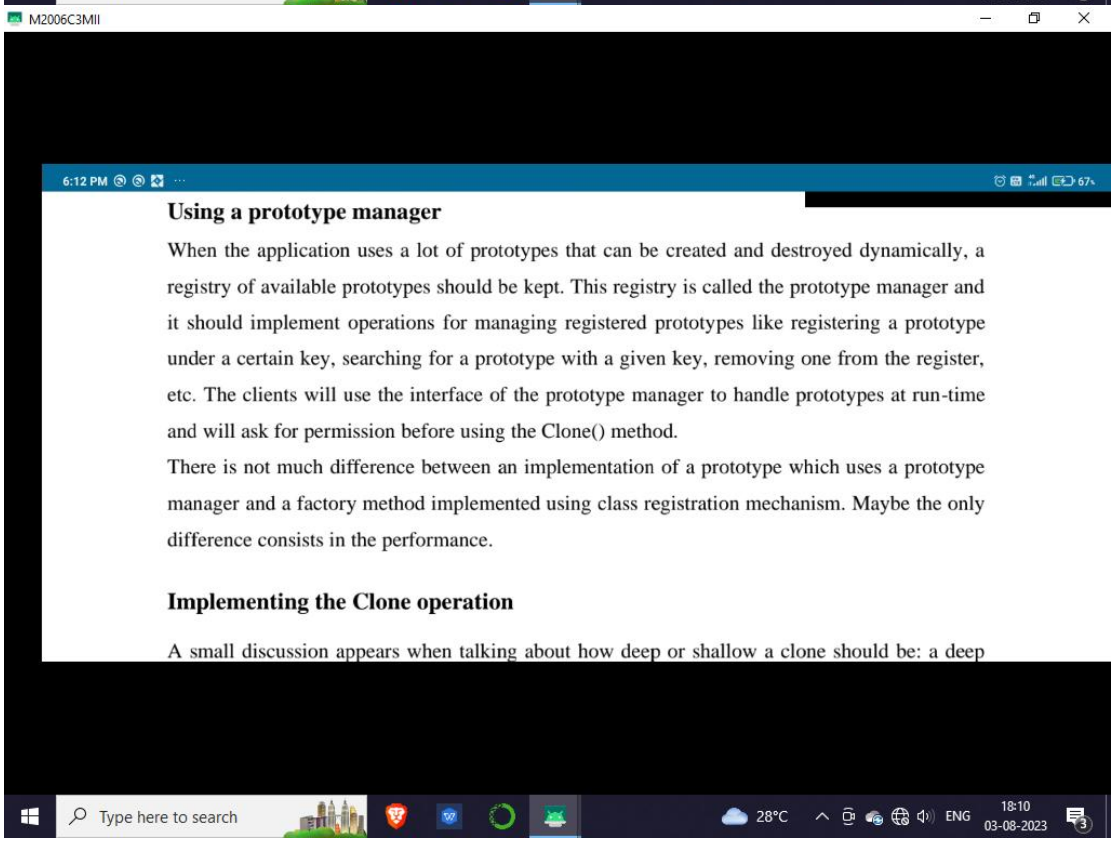
In building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the haze map using the same objects: wall, door, passage, room... The Prototype pattern is useful in this case because instead of hard coding (using new operation) the room, door, passage and wall objects that get instantiated, CreateMaze method will be parameterized by various prototypical room, door, wall and passage objects, so the composition of the map can be easily changed by replacing the prototypical objects with different ones.

The Client is the CreateMaze method and the ConcretePrototype classes will be the ones creating copies for different objects.

Example 3:

Type here to search 28°C ENG 18:10 03-08-2023


M2006C3MII
6:12 PM
Suppose we are doing a sales analysis on a set of data from a database. Normally, we would copy the information from the database, encapsulate it into an object and do the analysis. But if another analysis is needed on the same set of data, reading the database again and creating a new object is not the best idea. If we are using the Prototype pattern then the object used in the first analysis will be cloned and used for the other analysis.
The Client is here one of the methods that process an object that encapsulates information from the database. The ConcretePrototype classes will be classes that, from the object created after extracting data from the database, will copy it into objects used for analysis.
Specific problems and implementation
Using a prototype manager
When the application uses a lot of prototypes that can be created and destroyed dynamically, a
Type here to search
28°C
ENG
18:10
03-08-2023


M2006C3MII
6:12 PM
Using a prototype manager
When the application uses a lot of prototypes that can be created and destroyed dynamically, a registry of available prototypes should be kept. This registry is called the prototype manager and it should implement operations for managing registered prototypes like registering a prototype under a certain key, searching for a prototype with a given key, removing one from the register, etc. The clients will use the interface of the prototype manager to handle prototypes at run-time and will ask for permission before using the Clone() method.
There is not much difference between an implementation of a prototype which uses a prototype manager and a factory method implemented using class registration mechanism. Maybe the only difference consists in the performance.
Implementing the Clone operation
A small discussion appears when talking about how deep or shallow a clone should be: a deep
Type here to search
28°C
ENG
18:10
03-08-2023

M2006C3MII

6:12 PM

Implementing the Clone operation

A small discussion appears when talking about how deep or shallow a clone should be: a deep clone clones the instance variables in the cloning object while a shallow clone shares the instance variables between the clone and the original. Usually, a shallow clone is enough and very simple, but cloning complex prototypes should use deep clones so the clone and the original are independent, a deep clone needing its components to be the clones of the complex object's components.

Initializing clones

Type here to search 28°C ENG 18:10 03-08-2023

M2006C3MII

6:12 PM

There are cases when the internal states of a clone should be initialized after it is created. This happens because these values cannot be passed to the Clone() method, that uses an interface which would be destroyed if such parameters were used. In this case the initialization should be done by using setting and resetting operations of the prototype class or by using an initializing method that takes as parameters the values at which the clone's internal states should be set.

Key points

- Prototype Manager – implemented usually as a hashtable keeping the object to clone. When use it, prototype become a factory method which uses cloning instead of instantiation.
- Deep Clones vs. Shallow Clones – when we clone complex objects which contains other objects, we should take care how they are cloned. We can clone contained objects also (deep cloning) or we can the same reference for them, and to share them between cloned container objects.

Type here to search 28°C ENG 18:10 03-08-2023

M2006C3MII

6:12 PM

(deep cloning) or we can the same reference for them, and to share them between cloned container objects.

- Initializing Internal States – there are certain situations when objects need to be initialized after they are created.

Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

6:13 PM

Singleton Pattern

Intent

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.

Motivation

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

6:13 PM

The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

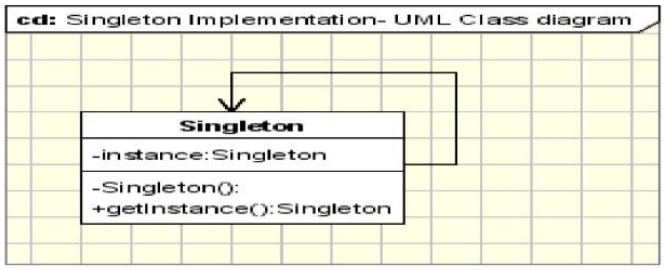
Implementation

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.

Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

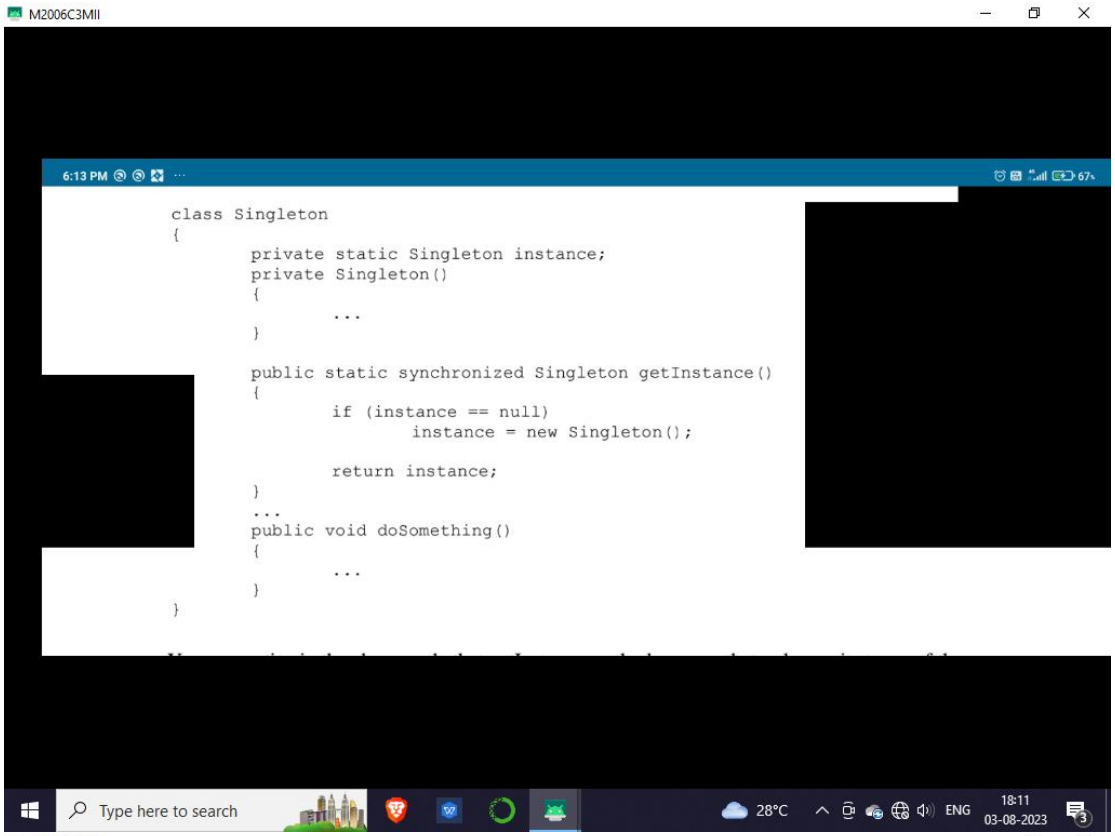
6:13 PM

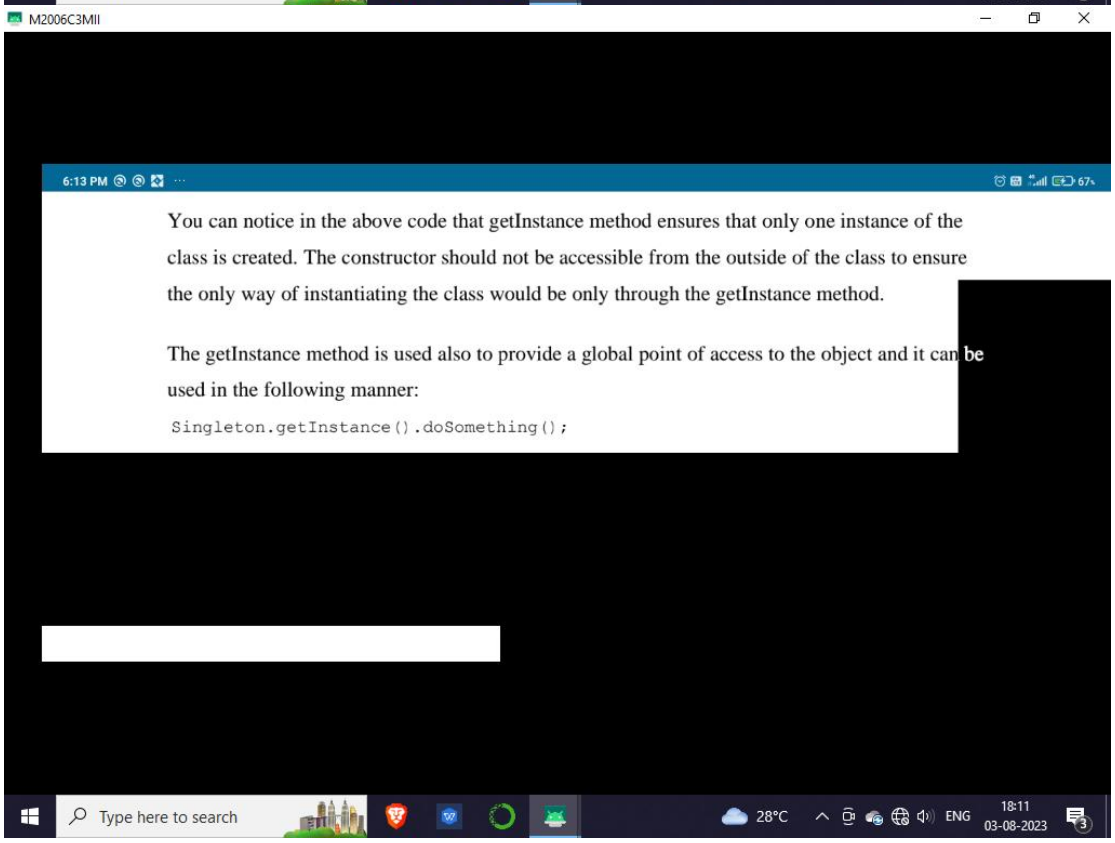


The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

```
class Singleton
{
```

Type here to search 28°C ENG 18:11 03-08-2023


M2006C3MII
6:13 PM
class Singleton
{
private static Singleton instance;
private Singleton()
{
...
}
public static synchronized Singleton getInstance()
{
if (instance == null)
instance = new Singleton();
return instance;
}
...
public void doSomething()
{
...
}
}
Type here to search
28°C
ENG
18:11
03-08-2023


M2006C3MII
6:13 PM
You can notice in the above code that getInstance method ensures that only one instance of the class is created. The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through the getInstance method.
The getInstance method is used also to provide a global point of access to the object and it can be used in the following manner:
Singleton.getInstance().doSomething();
Type here to search
28°C
ENG
18:11
03-08-2023

M2006C3MII

6:13 PM

Applicability & Examples With Source code

According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point. Here are some real situations where the singleton is used:

Example 1

Singleton pattern is one of the simplest design patterns. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

As its name indicate this pattern involves a single class which is responsible to creates own object while making sure that only single object get created

Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

6:13 PM

Singleton pattern is one of the simplest design patterns. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

As its name indicate this pattern involves a single class which is responsible to creates own object while making sure that only single object get created throughout the project implementation . This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

From implementation point of view we're going to create a *SingleObject* class. *SingleObject* class has its constructor as private and has a static instance of itself.

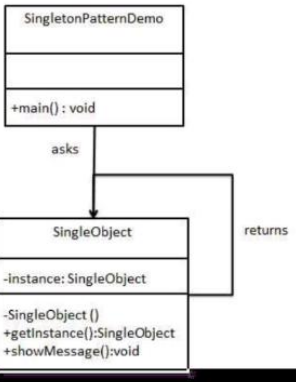
Class Diagram:

Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

6:13 PM

Class Diagram:



Type here to search 28°C ENG 18:11 03-08-2023

M2006C3MII

6:13 PM

Steps

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

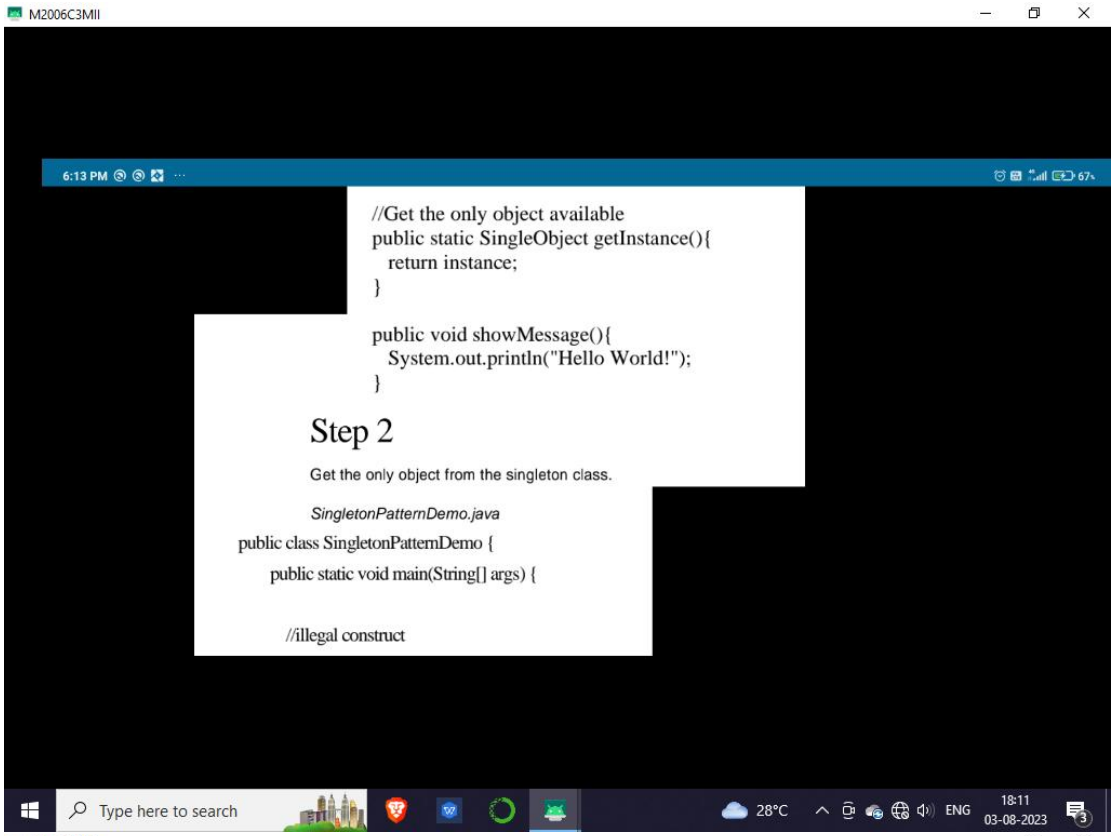
Step 1

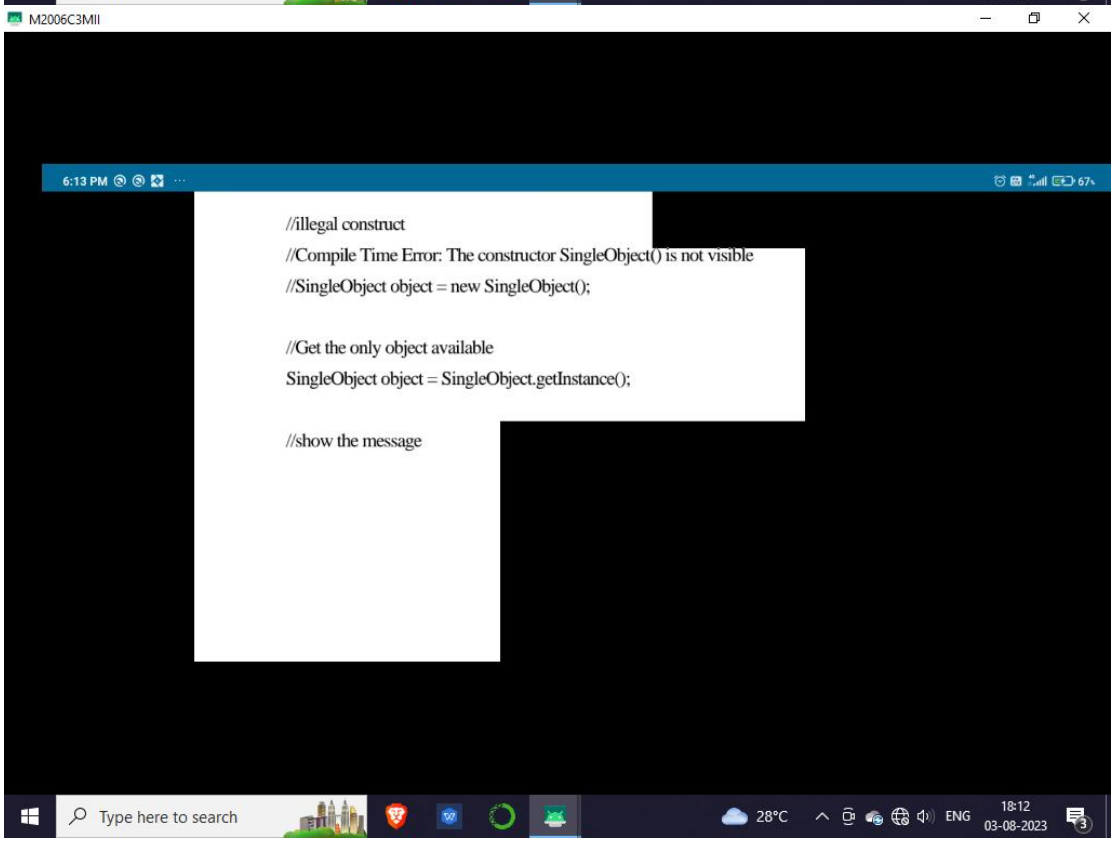
Create a Singleton Class.

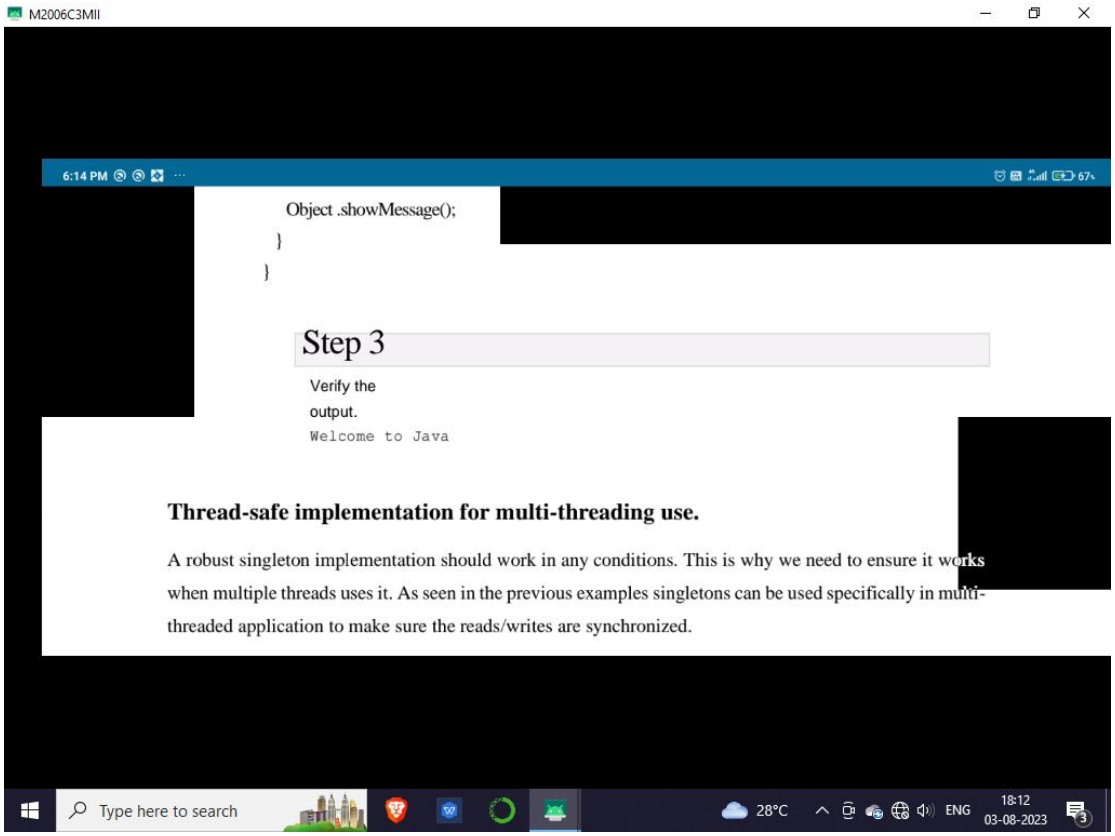
SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
```

Type here to search 28°C ENG 18:11 03-08-2023


M2006C3MII
6:13 PM
//Get the only object available
public static SingleObject getInstance(){
return instance;
}
public void showMessage(){
System.out.println("Hello World!");
}
Step 2
Get the only object from the singleton class.
SingletonPatternDemo.java
public class SingletonPatternDemo {
public static void main(String[] args) {
//illegal construct
Type here to search
28°C
ENG
18:11
03-08-2023


M2006C3MII
6:13 PM
//illegal construct
//Compile Time Error: The constructor SingleObject() is not visible
//SingleObject object = new SingleObject();
//Get the only object available
SingleObject object = SingleObject.getInstance();
//show the message
Type here to search
28°C
ENG
18:12
03-08-2023


M2006C3MII
6:14 PM
Object .showMessage();
}
}
Step 3
Verify the
output.
Welcome to Java
Thread-safe implementation for multi-threading use.
A robust singleton implementation should work in any conditions. This is why we need to ensure it works when multiple threads uses it. As seen in the previous examples singletons can be used specifically in multi-threaded application to make sure the reads/writes are synchronized.
Type here to search
28°C
ENG
18:12
03-08-2023

M2006C3MII

6:14 PM

Thread-safe implementation for multi-threading use.

A robust singleton implementation should work in any conditions. This is why we need to ensure it works when multiple threads uses it. As seen in the previous examples singletons can be used specifically in multi-threaded application to make sure the reads/writes are synchronized.

Lazy instantiation using double locking mechanism.

The standard implementation shown in the above code is a thread safe implementation, but it's not the best thread-safe implementation because synchronization is very expensive when we are talking about the performance. We can see that the synchronized method getInstance does not need to be checked for syncronization after the object is initialized. If we see that the singleton object is already created we just have to return it without using any syncronized block. This optimization consist in checking in an unsynchronized block if the object is null and if not to check again and create it in an syncronized block. This is called double locking mechanism.

Type here to search

28°C ENG 18:12 03-08-2023

M2006C3MII

6:14 PM

The standard implementation shown in the above code is a thread safe implementation, but it's not the best thread-safe implementation because synchronization is very expensive when we are talking about the performance. We can see that the synchronized method getInstance does not need to be checked for syncronization after the object is initialized. If we see that the singleton object is already created we just have to return it without using any syncronized block. This optimization consist in checking in an unsynchronized block if the object is null and if not to check again and create it in an syncronized block. This is called double locking mechanism.

In this case case the singleton instance is created when the getInstance() method is called for the first time. This is called lazy instantiation and it ensures that the singleton instance is created only when it is needed.

Type here to search

28°C ENG 18:12 03-08-2023