**Design And Patterns:**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**SOLUTION:**

**Logger.java:**

public class Logger {

    private static Logger instance;

    private Logger() {

        System.out.println("Logger instance created!");

    }

    public static Logger **getInstance**() {

         if (instance == null) {

             instance = new Logger();

        }

        return instance;

    }

    public void **log**(String *message*) {

        System.out.println("LOG: " + *message*); }

    public void **logError**(String *error*) {

        System.out.println("ERROR: " + *error*);

    }

    public void **logInfo**(String *info*) {

        System.out.println("INFO: " + *info*);

    }

}

**SingletonTest.java:**

public class SingletonTest {

        public static void **main**(String[] *args*) {

        System.out.println("=== Singleton Pattern Test ===\n");

        System.out.println("Test 1: Getting first Logger instance...");

        Logger logger1 = Logger.getInstance();

        logger1.log("This is the first log message");

        System.out.println("\nTest 2: Getting second Logger instance...");

        Logger logger2 = Logger.getInstance();

        logger2.logInfo("This is an info message from second reference");

        System.out.println("\nTest 3: Getting third Logger instance...");

        Logger logger3 = Logger.getInstance();

        logger3.logError("This is an error message from third reference");

        System.out.println("\nTest 4: Verifying all references are the same object...");

        System.out.println("logger1 == logger2: " + (logger1 == logger2));

        System.out.println("logger2 == logger3: " + (logger2 == logger3));

        System.out.println("logger1 == logger3: " + (logger1 == logger3));

        System.out.println("\nTest 5: Object identity (hash codes)...");

        System.out.println("logger1 hash code: " + System.identityHashCode(logger1));

        System.out.println("logger2 hash code: " + System.identityHashCode(logger2));

        System.out.println("logger3 hash code: " + System.identityHashCode(logger3));

        System.out.println("\nTest 6: Attempting to create new instances...");

        System.out.println("Note: The constructor is private, so we can only use getInstance()");

        System.out.println("\n=== Test Complete ===");

        System.out.println("Conclusion: All Logger references point to the same object!");

        System.out.println("The Singleton pattern is working correctly!");

    }

}

**ApplicationExample.java:**

public class ApplicationExample {

    public static void **main**(String[] *args*) {

        System.out.println("=== Application Example ===\n");

       System.out.println("Starting application...");

         authenticateUser("john\_doe");

         performDatabaseOperation("SELECT \* FROM users");

         processFile("data.txt");

         handleError("Connection timeout");

        System.out.println("\nApplication completed!");

    }

    private static void **authenticateUser**(String *username*) {

        Logger logger = Logger.getInstance();

        logger.logInfo("Attempting to authenticate user: " + *username*);

         if (*username*.equals("john\_doe")) {

            logger.log("User authentication successful");

        } else {

            logger.logError("User authentication failed for: " + *username*);

        }

    }

    private static void **performDatabaseOperation**(String *query*) {

        Logger logger = Logger.getInstance();

        logger.logInfo("Executing database query: " + *query*);

        logger.log("Database query executed successfully");

    }

    private static void **processFile**(String *filename*) {

        Logger logger = Logger.getInstance();

        logger.logInfo("Processing file: " + *filename*);

        logger.log("File processed successfully");

    }

    private static void **handleError**(String *errorMessage*) {

        Logger logger = Logger.getInstance();

        logger.logError("Application error: " + *errorMessage*);

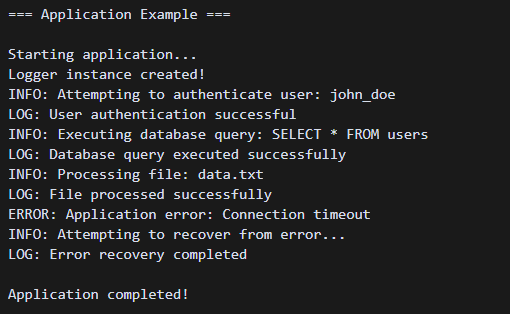
        logger.logInfo("Attempting to recover from error...");

        logger.log("Error recovery completed");

    }

}

**OUTPUT:**

****

**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**SOLUTION:**

**ExcelDocument.java**

public class ExcelDocument implements Document {

    public void **open**() {

        System.out.println("Opening Excel document...");

    }

    public void **close**() {

        System.out.println("Closing Excel document...");

    }

    public void **save**() {

        System.out.println("Saving Excel document...");

    }

}

**Main.java:**

public class Main {

    public static void main(String[] *args*) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        wordDoc.save();

        wordDoc.close();

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

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        pdfDoc.save();

        pdfDoc.close();

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

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

        excelDoc.save();

        excelDoc.close();

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

    public void **open**() {

        System.out.println("Opening PDF document...");}

    public void **close**() {

        System.out.println("Closing PDF document..."); }

    public void **save**() {

        System.out.println("Saving PDF document...");

    } }

**WordDocument.java:**

public class WordDocument implements Document {

    public void **open**() {

        System.out.println("Opening Word document...");

    }

    public void **close**() {

        System.out.println("Closing Word document...");

    }

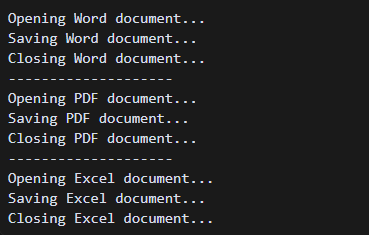
    public void **save**() {

        System.out.println("Saving Word document...");

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**SOLUTION:**

**BuildPatternTest.java:**

**public class BuilderPatternTest {**

**public static void main(String[] *args*) {**

**System.out.println("=== Builder Pattern Example: Computer Configuration ===\n");**

**System.out.println("1. Basic Computer Configuration:");**

**Computer basicComputer = new Computer.Builder("Intel Core i3", "8GB DDR4", "256GB SSD")**

**.build();**

**System.out.println(basicComputer);**

**System.out.println("\n2. Gaming Computer Configuration:");**

**Computer gamingComputer = new Computer.Builder("Intel Core i7", "16GB DDR4", "1TB NVMe SSD")**

**.graphicsCard("NVIDIA RTX 3080")**

**.motherboard("ASUS ROG Strix Z590")**

**.powerSupply("750W Gold Certified")**

**.computerCase("NZXT H510 Elite")**

**.hasWifi(true)**

**.hasBluetooth(true)**

**.build();**

**System.out.println(gamingComputer);**

**System.out.println("\n3. Workstation Computer Configuration:");**

**Computer workstationComputer = new Computer.Builder("AMD Ryzen 9", "32GB DDR4", "2TB NVMe SSD")**

**.graphicsCard("NVIDIA Quadro RTX 4000")**

**.motherboard("MSI MPG X570")**

**.powerSupply("850W Platinum Certified")**

**.computerCase("Fractal Design Define 7")**

**.hasWifi(true)**

**.hasBluetooth(true)**

**.build();**

**System.out.println(workstationComputer);**

**System.out.println("\n4. Budget Computer Configuration:");**

**Computer budgetComputer = new Computer.Builder("AMD Ryzen 5", "8GB DDR4", "500GB HDD")**

**.graphicsCard("AMD Radeon RX 550")**

**.powerSupply("450W Bronze Certified")**

**.computerCase("Cooler Master Q300L")**

**.hasWifi(true)**

**.build();**

**System.out.println(budgetComputer);**

**System.out.println("\n5. Server Computer Configuration:");**

**Computer serverComputer = new Computer.Builder("Intel Xeon E5", "64GB ECC DDR4", "4TB Enterprise HDD")**

**.motherboard("Supermicro X11")**

**.powerSupply("1000W Redundant")**

**.computerCase("Rack Mount 2U")**

**.hasWifi(false)**

**.hasBluetooth(false)**

**.build();**

**System.out.println(serverComputer);**

**System.out.println("\n=== Builder Pattern Benefits ===");**

**System.out.println("✓ Flexible object construction with optional parameters");**

**System.out.println("✓ Immutable objects once built");**

**System.out.println("✓ Fluent interface for method chaining");**

**System.out.println("✓ Clear separation between required and optional parameters");**

**System.out.println("✓ Easy to extend with new parameters");**

**}**

**}**

**Computer.java:**

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private String graphicsCard;

    private String motherboard;

    private String powerSupply;

    private String computerCase;

    private boolean hasWifi;

    private boolean hasBluetooth;

    private Computer(Builder *builder*) {

        this.cpu = *builder*.cpu;

        this.ram = *builder*.ram;

        this.storage = *builder*.storage;

        this.graphicsCard = *builder*.graphicsCard;

        this.motherboard = *builder*.motherboard;

        this.powerSupply = *builder*.powerSupply;

        this.computerCase = *builder*.computerCase;

        this.hasWifi = *builder*.hasWifi;

        this.hasBluetooth = *builder*.hasBluetooth;

    }

    public static class Builder {

*// Required parameters*

        private String cpu;

        private String ram;

        private String storage;

        private String graphicsCard = "Integrated Graphics";

        private String motherboard = "Standard Motherboard";

        private String powerSupply = "500W Power Supply";

        private String computerCase = "Standard Case";

        private boolean hasWifi = false;

        private boolean hasBluetooth = false;

        public Builder(String *cpu*, String *ram*, String *storage*) {

            this.cpu = *cpu*;

            this.ram = *ram*;

            this.storage = *storage*;

        }

        public Builder **graphicsCard**(String *graphicsCard*) {

            this.graphicsCard = *graphicsCard*;

            return this;

        }

        public Builder **motherboard**(String *motherboard*) {

            this.motherboard = *motherboard*;

            return this;

        }

        public Builder **powerSupply**(String *powerSupply*) {

            this.powerSupply = *powerSupply*;

            return this;

        }

        public Builder **computerCase**(String *computerCase*) {

            this.computerCase = *computerCase*;

            return this;

        }

        public Builder **hasWifi**(boolean *hasWifi*) {

            this.hasWifi = *hasWifi*;

            return this;

        }

        public Builder **hasBluetooth**(boolean *hasBluetooth*) {

            this.hasBluetooth = *hasBluetooth*;

            return this;

        }

        public Computer **build**() {

            return new Computer(this);

        }

    }

    public String **getCpu**() {

        return cpu;

    }

    public String **getRam**() {

        return ram;

    }

    public String **getStorage**() {

        return storage;

    }

    public String **getGraphicsCard**() {

        return graphicsCard;

    }

    public String **getMotherboard**() {

        return motherboard;

    }

    public String **getPowerSupply**() {

        return powerSupply;

    }

    public String **getComputerCase**() {

        return computerCase;

    }

    public boolean **hasWifi**() {

        return hasWifi;

    }

    public boolean **hasBluetooth**() {

        return hasBluetooth;

    }

    public String **toString**() {

        StringBuilder sb = new StringBuilder();

        sb.append("Computer Configuration:\n");

        sb.append("CPU: ").append(cpu).append("\n");

        sb.append("RAM: ").append(ram).append("\n");

        sb.append("Storage: ").append(storage).append("\n");

        sb.append("Graphics Card: ").append(graphicsCard).append("\n");

        sb.append("Motherboard: ").append(motherboard).append("\n");

        sb.append("Power Supply: ").append(powerSupply).append("\n");

        sb.append("Case: ").append(computerCase).append("\n");

        sb.append("WiFi: ").append(hasWifi ? "Yes" : "No").append("\n");

        sb.append("Bluetooth: ").append(hasBluetooth ? "Yes" : "No").append("\n");

        return sb.toString();

    }

}

**Computer$Builder.class:**

*// Source code is decompiled from a .class file using FernFlower decompiler.*

public class Computer$Builder {

   private String cpu;

   private String ram;

   private String storage;

   private String graphicsCard = "Integrated Graphics";

   private String motherboard = "Standard Motherboard";

   private String powerSupply = "500W Power Supply";

   private String computerCase = "Standard Case";

   private boolean hasWifi = false;

   private boolean hasBluetooth = false;

   public Computer$Builder(String *var1*, String *var2*, String *var3*) {

      this.cpu = var1;

      this.ram = var2;

      this.storage = var3;

   }

   public Computer$Builder graphicsCard(String *var1*) {

      this.graphicsCard = var1;

      return this;

   }

   public Computer$Builder motherboard(String *var1*) {

      this.motherboard = var1;

      return this;

   }

   public Computer$Builder powerSupply(String *var1*) {

      this.powerSupply = var1;

      return this;

   }

   public Computer$Builder computerCase(String *var1*) {

      this.computerCase = var1;

      return this;

   }

   public Computer$Builder hasWifi(boolean *var1*) {

      this.hasWifi = var1;

      return this;

   }

   public Computer$Builder hasBluetooth(boolean *var1*) {

      this.hasBluetooth = var1;

      return this;

   }

   public Computer build() {

      return new Computer(this);

   }

}

**OUTPUT:**

=== Builder Pattern Example: Computer Configuration ===

1. Basic Computer Configuration:

Computer Configuration:

CPU: Intel Core i3

RAM: 8GB DDR4

Storage: 256GB SSD

Graphics Card: Integrated Graphics

Motherboard: Standard Motherboard

Power Supply: 500W Power Supply

Case: Standard Case

WiFi: No

Bluetooth: No

2. Gaming Computer Configuration:

Computer Configuration:

CPU: Intel Core i7

RAM: 16GB DDR4

Storage: 1TB NVMe SSD

Graphics Card: NVIDIA RTX 3080

Motherboard: ASUS ROG Strix Z590

Power Supply: 750W Gold Certified

Case: NZXT H510 Elite

WiFi: Yes

Bluetooth: Yes

3. Workstation Computer Configuration:

Computer Configuration:

CPU: AMD Ryzen 9

RAM: 32GB DDR4

Storage: 2TB NVMe SSD

Graphics Card: NVIDIA Quadro RTX 4000

Motherboard: MSI MPG X570

Power Supply: 850W Platinum Certified

Case: Fractal Design Define 7

WiFi: Yes

Bluetooth: Yes

4. Budget Computer Configuration:

Computer Configuration:

CPU: AMD Ryzen 5

RAM: 8GB DDR4

Storage: 500GB HDD

Graphics Card: AMD Radeon RX 550

Motherboard: Standard Motherboard

Power Supply: 450W Bronze Certified

Case: Cooler Master Q300L

WiFi: Yes

Bluetooth: No

5. Server Computer Configuration:

Computer Configuration:

CPU: Intel Xeon E5

RAM: 64GB ECC DDR4

Storage: 4TB Enterprise HDD

Graphics Card: Integrated Graphics

Motherboard: Supermicro X11

Power Supply: 1000W Redundant

Case: Rack Mount 2U

WiFi: No

Bluetooth: No

=== Builder Pattern Benefits ===

✓ Flexible object construction with optional parameters

✓ Immutable objects once built

✓ Fluent interface for method chaining

✓ Clear separation between required and optional parameters

✓ Easy to extend with new parameters

**Exercise 4: Implementing the Adapter Pattern**

Scenario:

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

Steps:

1. Create a New Java Project:
   * Create a new Java project named AdapterPatternExample.
2. Define Target Interface:
   * Create an interface PaymentProcessor with methods like processPayment().
3. Implement Adaptee Classes:
   * Create classes for different payment gateways with their own methods.
4. Implement the Adapter Class:
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. Test the Adapter Implementation:
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

**SOLUTION:**

**Main.java:**

package com.example.payment;

public class Main {

    public static void **main**(String[] *args*) {

        PayPalGateway payPalGateway = new PayPalGateway();

        PaymentProcessor payPalProcessor = new PayPalAdapter(payPalGateway, "user@example.com");

        payPalProcessor.processPayment(100.0);

        System.out.println();

        StripeGateway stripeGateway = new StripeGateway();

        PaymentProcessor stripeProcessor = new StripeAdapter(stripeGateway);

        stripeProcessor.processPayment(200.0);

    }

}

**PaymentProcessor.java:**

package com.example.payment;

public interface PaymentProcessor {

    void processPayment(double *amount*);

}

**PayPalAdapter.java:**

package com.example.payment;

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    private String account;

    public PayPalAdapter(PayPalGateway *payPalGateway*, String *account*) {

        this.payPalGateway = *payPalGateway*;

        this.account = *account*;

    }

    public void **processPayment**(double *amount*) {

        payPalGateway.makePayment(account, *amount*);

    }

}

**PayPalGateway.java:**

package com.example.payment;

public class PayPalGateway {

    public void **makePayment**(String *account*, double *amount*) {

        System.out.println("Processing PayPal payment of $" + *amount* + " for account " + *account*);

    }

}

**StripeAdapter.java:**

package com.example.payment;

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway *stripeGateway*) {

        this.stripeGateway = *stripeGateway*;

    }

    public void **processPayment**(double *amount*) {

        stripeGateway.charge(*amount*);

    }

}

**StripeGateway.java:**

package com.example.payment;

public class StripeGateway {

    public void **charge**(double *amount*) {

        System.out.println("Charging $" + *amount* + " through Stripe.");

    }

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

Scenario:

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

Steps:

1. Create a New Java Project:
   * Create a new Java project named DecoratorPatternExample.
2. Define Component Interface:
   * Create an interface Notifier with a method send().
3. Implement Concrete Component:
   * Create a class EmailNotifier that implements Notifier.
4. Implement Decorator Classes:
   * Create abstract decorator class NotifierDecorator that implements Notifier and holds a reference to a Notifier object.
   * Create concrete decorator classes like SMSNotifierDecorator, SlackNotifierDecorator that extend NotifierDecorator.
5. Test the Decorator Implementation:
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**SOLUTION:**

**EmailNotification.java:**

package com.example.decorator;

public class EmailNotifier implements Notifier {

    @Override

    public void **send**(String *message*) {

        System.out.println("Sending email with message: " + *message*);

    }

}

**Notifier.java:**

package com.example.decorator;

public interface Notifier {

    void **send**(String *message*);

}

**NotifierDecorator.java:**

package com.example.decorator;

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier *notifier*) {

        this.wrappedNotifier = *notifier*;

    }

    public void send(String *message*) {

        wrappedNotifier.send(message);

    }

}

**SlackNotifierDecorator.java:**

package com.example.decorator;

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier *notifier*) {

        super(*notifier*);

    }

    public void **send**(String *message*) {

        super.send(*message*);

        sendSlack(*message*);

    }

    private void **sendSlack**(String *message*) {

        System.out.println("Sending Slack message with message: " + *message*);

    }

}

**SMSNotifierDecorator.java:**

package com.example.decorator;

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier *notifier*) {

        super(*notifier*);

    }

    public void **send**(String *message*) {

        super.send(*message*);

        sendSMS(*message*);

    }

    private void **sendSMS**(String *message*) {

        System.out.println("Sending SMS with message: " + *message*);

    }

}

**TestDecorator.java:**

package com.example.decorator;

public class TestDecorator {

    public static void **main**(String[] *args*) {

        Notifier notifier = new EmailNotifier();

        notifier.send("Hello World!");

        System.out.println("-----");

        Notifier smsNotifier = new SMSNotifierDecorator(notifier);

        smsNotifier.send("Hello World!");

        System.out.println("-----");

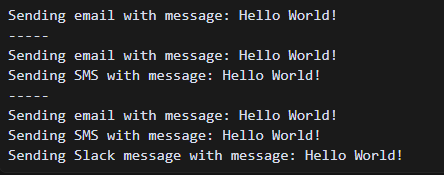
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("Hello World!");

    }

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**

Create a test class to demonstrate the use of **ProxyImage** to load and display images

**SOLUTION:**

**Image.java:**

package com.example.proxy;

public interface Image {

    void **display**();

}

**ProxyImage.java:**

package com.example.proxy;

public class ProxyImage implements Image {

    private RealImage realImage;

    private final String filename;

    public ProxyImage(String *filename*) {

        this.filename = *filename*;

    }

    public void **display**() {

        if (realImage == null) {

            realImage = new RealImage(filename);

        }

        realImage.display();

    }

}

**ProxyPatternTest.java:**

package com.example.proxy;

public class ProxyPatternTest {

    public static void **main**(String[] *args*) {

        Image image1 = new ProxyImage("image1.jpg");

        Image image2 = new ProxyImage("image2.jpg");

        System.out.println("--- First call for image1 ---");

        image1.display();

        System.out.println();

        System.out.println("--- Second call for image1 ---");

        image1.display();

        System.out.println();

        System.out.println("--- First call for image2 ---");

        image2.display();

    }

}

**RealImage.java:**

package com.example.proxy;

public class RealImage implements Image {

    private String filename;

    public RealImage(String *filename*) {

        this.filename = *filename*;

        loadFromRemoteServer();

    }

    private void **loadFromRemoteServer**() {

        System.out.println("Loading " + filename + " from remote server...");

        try {

            Thread.sleep(2000);

        } catch (InterruptedException *e*) {

            e.printStackTrace();

        }

    }

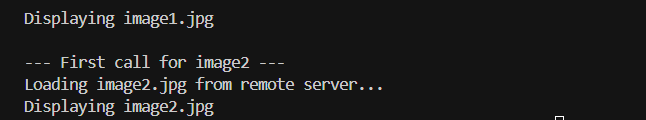
    public void **display**() {

        System.out.println("Displaying " + filename);

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**SOLUTION:**

**Main.java:**

public class Main {

    public static void **main**(String[] *args*) {

        StockMarket stockMarket = new StockMarket("GOOGL", 150.00);

        Observer mobileApp1 = new MobileApp("MobileApp1");

        Observer webApp1 = new WebApp("WebApp1");

        stockMarket.register(mobileApp1);

        stockMarket.register(webApp1);

        System.out.println("--- Setting stock price to 152.50 ---");

        stockMarket.setStockPrice(152.50);

        System.out.println("\n--- Deregistering MobileApp1 and setting stock price to 155.00 ---");

        stockMarket.deregister(mobileApp1);

        stockMarket.setStockPrice(155.00);

    } }

**MobileApp.java:**

public class MobileApp implements Observer {

    private String name;

    public MobileApp(String *name*) {

        this.name = *name*;

    }

    public void **update**(String *stockName*, double *stockPrice*) {

        System.out.println(name + " received stock update: " + *stockName* + " is now $" + *stockPrice*);

    }

}

**Observer.java:**

public interface Observer {

    void **update**(String *stockName*, double *stockPrice*);

}

**Stock.java:**

public interface Stock {

    void **register**(Observer *observer*);

    void **deregister**(Observer *observer*);

    void **notifyObservers**();

}

**StockMarket.java:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

    private List<Observer> observers;

    private String stockName;

    private double stockPrice;

    public StockMarket(String *stockName*, double *stockPrice*) {

        this.observers = new ArrayList<>();

        this.stockName = *stockName*;

        this.stockPrice = *stockPrice*;

    }

    public void **setStockPrice**(double *stockPrice*) {

        this.stockPrice = *stockPrice*;

        notifyObservers();

    }

    public void **register**(Observer *observer*) {

        observers.add(*observer*);

    }

    public void **deregister**(Observer *observer*) {

        observers.remove(*observer*);

    }

    public void **notifyObservers**() {

        for (Observer observer : observers) {

            observer.update(stockName, stockPrice);

        }   } }

**WebApp.java:**

public class WebApp implements Observer {

    private String name;

    public WebApp(String *name*) {

        this.name = *name*;

    }

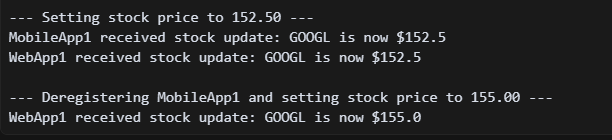
    public void **update**(String *stockName*, double *stockPrice*) {

        System.out.println(name + " received stock update: " + *stockName* + " is now $" + *stockPrice*);

    }

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

Scenario:

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

Steps:

1. Create a New Java Project:
   * Create a new Java project named StrategyPatternExample.
2. Define Strategy Interface:
   * Create an interface PaymentStrategy with a method pay().
3. Implement Concrete Strategies:
   * Create classes CreditCardPayment, PayPalPayment that implement PaymentStrategy.
4. Implement Context Class:
   * Create a class PaymentContext that holds a reference to PaymentStrategy and a method to execute the strategy.
5. Test the Strategy Implementation:
   * Create a test class to demonstrate selecting and using different payment strategies.

**SOLUTION:**

**CreditCardPayment.java:**

package com.example;

public class CreditCardPayment implements PaymentStrategy {

    private String name;

    private String cardNumber;

    private String cvv;

    private String dateOfExpiry;

    public CreditCardPayment(String *name*, String *cardNumber*, String *cvv*, String *dateOfExpiry*) {

        this.name = *name*;

        this.cardNumber = *cardNumber*;

        this.cvv = *cvv*;

        this.dateOfExpiry = *dateOfExpiry*;

    }

    public void **pay**(int *amount*) {

        if (*amount* <= 0) {

            System.out.println("Payment amount must be positive.");

            return;

        }

        System.out.println(*amount* + " paid with credit/debit card");

    }

}

**PaymentContext.java:**

package com.example;

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void **setPaymentStrategy**(PaymentStrategy *strategy*) {

        this.paymentStrategy = *strategy*;

    }

    public void **pay**(int *amount*) {

        paymentStrategy.pay(*amount*);

    }

}

**PaymentStrategy.java:**

package com.example;

public interface PaymentStrategy {

    void **pay**(int *amount*);

}

**StrategyPatternTest.java:**

package com.example;

public class StrategyPatternTest {

    public static void **main**(String[] *args*) {

        PaymentContext context = new PaymentContext();

        PaymentStrategy creditCard = new CreditCardPayment("John Doe", "1234567890123456", "786", "12/15");

        context.setPaymentStrategy(creditCard);

        System.out.println("Paying 100 using Credit Card...");

        context.pay(100);

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

        PaymentStrategy payPal = new PayPalPayment("john.doe@example.com", "mypassword");

        context.setPaymentStrategy(payPal);

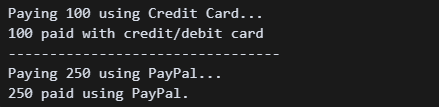
        System.out.println("Paying 250 using PayPal...");

        context.pay(250);

    }

}

**OUTPUT:**

****

**Exercise 9: Implementing the Command Pattern**

Scenario: You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

Steps:

1. Create a New Java Project:
   * Create a new Java project named CommandPatternExample.
2. Define Command Interface:
   * Create an interface Command with a method execute().
3. Implement Concrete Commands:
   * Create classes LightOnCommand, LightOffCommand that implement Command.
4. Implement Invoker Class:
   * Create a class RemoteControl that holds a reference to a Command and a method to execute the command.
5. Implement Receiver Class:
   * Create a class Light with methods to turn on and off.
6. Test the Command Implementation:
   * Create a test class to demonstrate issuing commands using the RemoteControl.

**SOLUTION:**

**Command.class:**

interface Command {

   void execute();

}

**CommandPatternExample.java:**

interface Command {

    void **execute**();

}

class Light {

    public void **on**() {

        System.out.println("Light is On");

    }

    public void **off**() {

        System.out.println("Light is Off");

    }

}

class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light *light*) {

        this.light = *light*;

    }

    public void **execute**() {

        light.on();

    }

}

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light *light*) {

        this.light = *light*;

    }

    public void **execute**() {

        light.off();

    }

}

class RemoteControl {

    private Command command;

    public void **setCommand**(Command *command*) {

        this.command = *command*;

    }

    public void **pressButton**() {

        command.execute();

    }

}

public class CommandPatternExample {

    public static void **main**(String[] *args*) {

        Light light = new Light();

        Command lightOn = new LightOnCommand(light);

        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Light.class:**

class Light {

   Light() {

   }

   public void on() {

      System.out.println("Light is On");

   }

   public void off() {

      System.out.println("Light is Off");

   }

}

**LightOffCommand.class:**

class LightOffCommand implements Command {

   private Light light;

   public LightOffCommand(Light *var1*) {

      this.light = var1;

   }

   public void execute() {

      this.light.off();

   }

}

**LightOnCommand.class:**

class LightOnCommand implements Command {

   private Light light;

   public LightOnCommand(Light *var1*) {

      this.light = var1;

   }

   public void execute() {

      this.light.on();

   }

}

**RemoteControl.class:**

class RemoteControl {

   private Command command;

   RemoteControl() {

   }

   public void setCommand(Command *var1*) {

      this.command = var1;

   }

   public void pressButton() {

      this.command.execute();

   }

}

**OUTPUT:**

Light is On

Light is Off

**Exercise 10: Implementing the MVC Pattern**

Scenario:

You are developing a simple web application for managing student records using the MVC pattern.

Steps:

1. Create a New Java Project:
   * Create a new Java project named MVCPatternExample.
2. Define Model Class:
   * Create a class Student with attributes like name, id, and grade.
3. Define View Class:
   * Create a class StudentView with a method displayStudentDetails().
4. Define Controller Class:
   * Create a class StudentController that handles the communication between the model and the view.
5. Test the MVC Implementation:
   * Create a main class to demonstrate creating a Student, updating its details using StudentController, and displaying them using StudentView

**SOLUTION:**

**MVCPatternExample.java:**

package com.example;

public class MVCPatternExample {

    public static void **main**(String[] *args*) {

        Student model = retrieveStudentFromDatabase();

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        System.out.println("Initial student details:");

        controller.updateView();

        System.out.println("\nUpdating student details...");

        controller.setStudentName("John Doe");

        controller.setStudentGrade("A+");

        System.out.println("\nUpdated student details:");

        controller.updateView();

    }

    private static Student **retrieveStudentFromDatabase**() {

        Student student = new Student();

        student.setName("Jane Doe");

        student.setId("12345");

        student.setGrade("A");

        return student;

    }

}

**Student.java:**

package com.example;

public class Student {

    private String name;

    private String id;

    private String grade;

    public String **getName**() {

        return name;

    }

    public void **setName**(String *name*) {

        this.name = *name*;

    }

    public String **getId**() {

        return id;

    }

    public void **setId**(String *id*) {

        this.id = *id*;

    }

    public String **getGrade**() {

        return grade;

    }

    public void **setGrade**(String *grade*) {

        this.grade = *grade*;

    }

}

**StudentController.java:**

package com.example;

public class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student *model*, StudentView *view*) {

        this.model = *model*;

        this.view = *view*;

    }

    public void **setStudentName**(String *name*) {

        model.setName(*name*);

    }

    public String **getStudentName**() {

        return model.getName();

    }

    public void **setStudentId**(String *id*) {

        model.setId(*id*);

    }

    public String **getStudentId**() {

        return model.getId();

    }

    public void **setStudentGrade**(String *grade*) {

        model.setGrade(*grade*);

    }

    public String **getStudentGrade**() {

        return model.getGrade();

    }

    public void **updateView**() {

        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

    }

}

**StudentVeiw.java:**

package com.example;

public class StudentView {

    public void **displayStudentDetails**(String *studentName*, String *studentId*, String *studentGrade*) {

        System.out.println("Student Details:");

        System.out.println("Name: " + *studentName*);

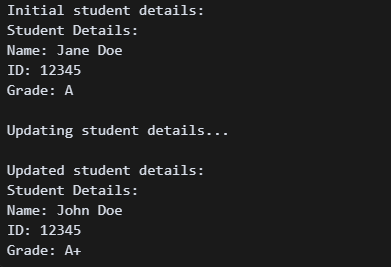
        System.out.println("ID: " + *studentId*);

        System.out.println("Grade: " + *studentGrade*);

    }

}

**SOLUTION:**

****

**Exercise 11: Implementing Dependency Injection**

Scenario:

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

Steps:

1. Create a New Java Project:
   * Create a new Java project named DependencyInjectionExample.
2. Define Repository Interface:
   * Create an interface CustomerRepository with methods like findCustomerById().
3. Implement Concrete Repository:
   * Create a class CustomerRepositoryImpl that implements CustomerRepository.
4. Define Service Class:
   * Create a class CustomerService that depends on CustomerRepository.
5. Implement Dependency Injection:
   * Use constructor injection to inject CustomerRepository into CustomerService.
6. Test the Dependency Injection Implementation:
   * Create a main class to demonstrate creating a CustomerService with CustomerRepositoryImpl and using it to find a customer.

**SOLUTION:**

**CustomerRepository.java:**

package com.example.di;

public interface CustomerRepository {

    String **findCustomerById**(int *id*);

}

**CustomerRepositoryImpl.java:**

package com.example.di;

public class CustomerRepositoryImpl implements CustomerRepository {

    public String **findCustomerById**(int *id*) {

        return "Customer " + *id*;

    }

}

**CustomerService.java:**

package com.example.di;

public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository *customerRepository*) {

        this.customerRepository = *customerRepository*;

    }

    public String **getCustomer**(int *id*) {

        return customerRepository.findCustomerById(*id*);

    }

}

**Main.java:**

package com.example.di;

public class Main {

    public static void **main**(String[] *args*) {

        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        CustomerService customerService = new CustomerService(customerRepository);

        String customer = customerService.getCustomer(1);

        System.out.println(customer);

    }

}

**OUTPUT:**

**Customer 1**