Digital Nurture 3.0 I Deep Skilling (WEEK 1 SOLUTIONS)

Superset ID:5021661

Name: Priya Hazra

Module 1 - Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern

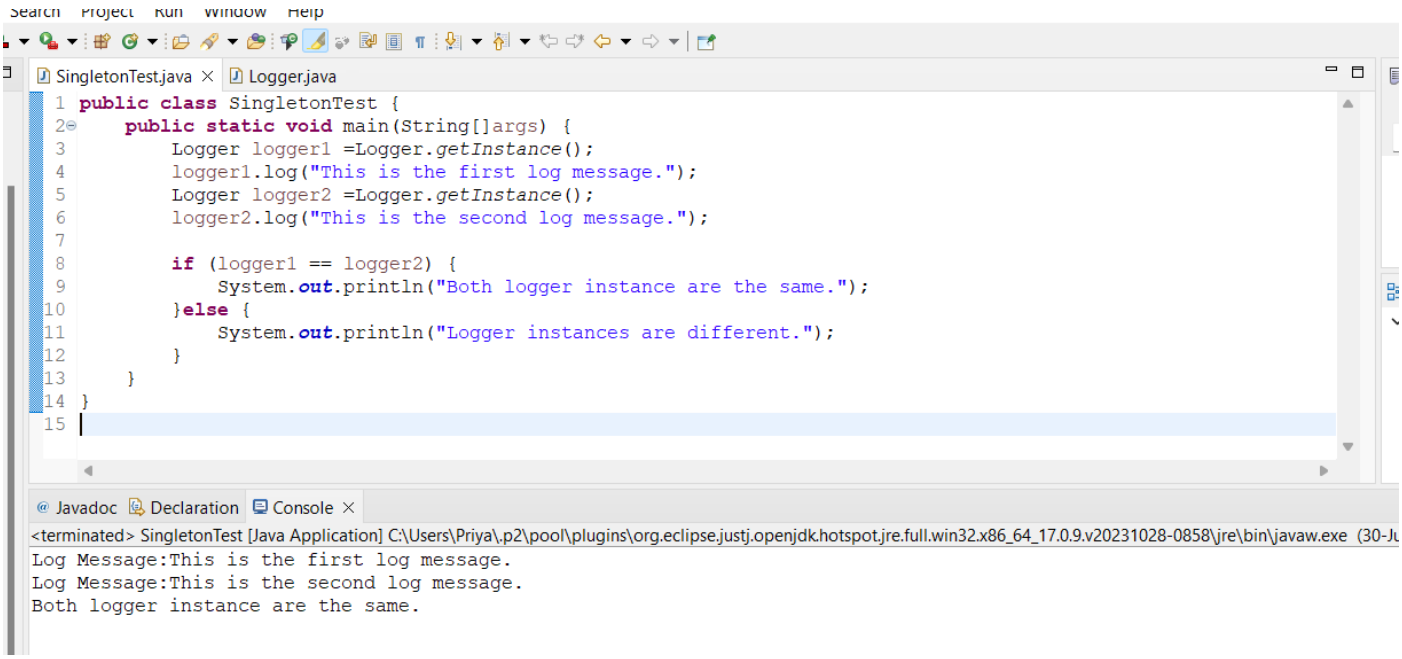
CODE:

SingletonTest.java

```
public class SingletonTest {
	public static void main(String[]args) {
		Logger logger1 =Logger.getInstance();
		logger1.log("This is the first log message.");
		Logger logger2 =Logger.getInstance();
		logger2.log("This is the second log message.");

		if (logger1 == logger2) {
			System.out.println("Both logger instance are the same.");
		}else {
			System.out.println("Logger instances are different.");
		}
	}
}
```

Logger.java

```
public class Logger {
	private static Logger instance;
	private Logger() {
	}
	public static Logger getInstance() {
		if (instance == null) {
			instance = new Logger();
		}
		return instance;
	}
	public void log(String message) {
		System.out.println("Log Message:" + message);
	}
}
```



Exercise 2: Implementing the Factory Method Pattern

Code:

Document.java

```
public interface Document {
   void open();
   void close();
}
```

DocumentFactory.java

```
public abstract class DocumentFactory {
   public abstract Document createDocument();
}
```

ExcelDocument.java

```
public class ExcelDocument implements Document {
   @Override
   public void open() {
      System.out.println("Opening Excel document.");
   }
   @Override
   public void close() {
      System.out.println("Closing Excel document.");
   }
}
```

ExcelDocumentFactory.java

```
public class ExcelDocumentFactory extends DocumentFactory {
   @Override
   public Document createDocument() {
      return new ExcelDocument();
   }
}
```

FactoryMethodTest.java

```
public class FactoryMethodTest {
   public static void main(String[] args) {
```

```
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();
        wordDoc.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();
        pdfDoc.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
        excelDoc.close();
    }
}
```

PdfDocument.java

```
public class PdfDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening PDF document.");
    }

    @Override
    public void close() {
        System.out.println("Closing PDF document.");
    }
}
```

PdfDocumentFactory.java

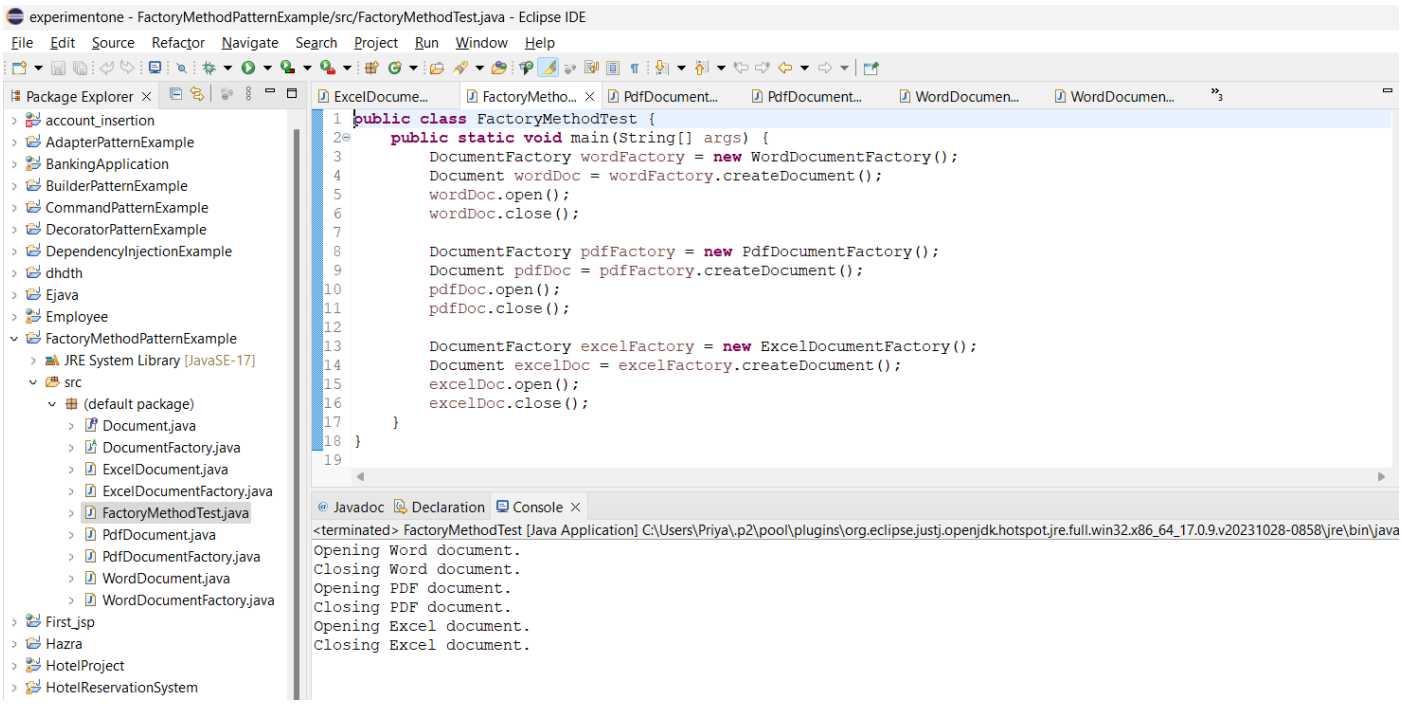
```
public class PdfDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new PdfDocument();
    }
}
```

WordDocument.java

```
public class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Word document.");
    }
    @Override
    public void close() {
        System.out.println("Closing Word document.");
    }
}
```

WordDocumentFactory.java

```
public class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}
```



Exercise 3: Implementing the Builder Pattern

Computer.java

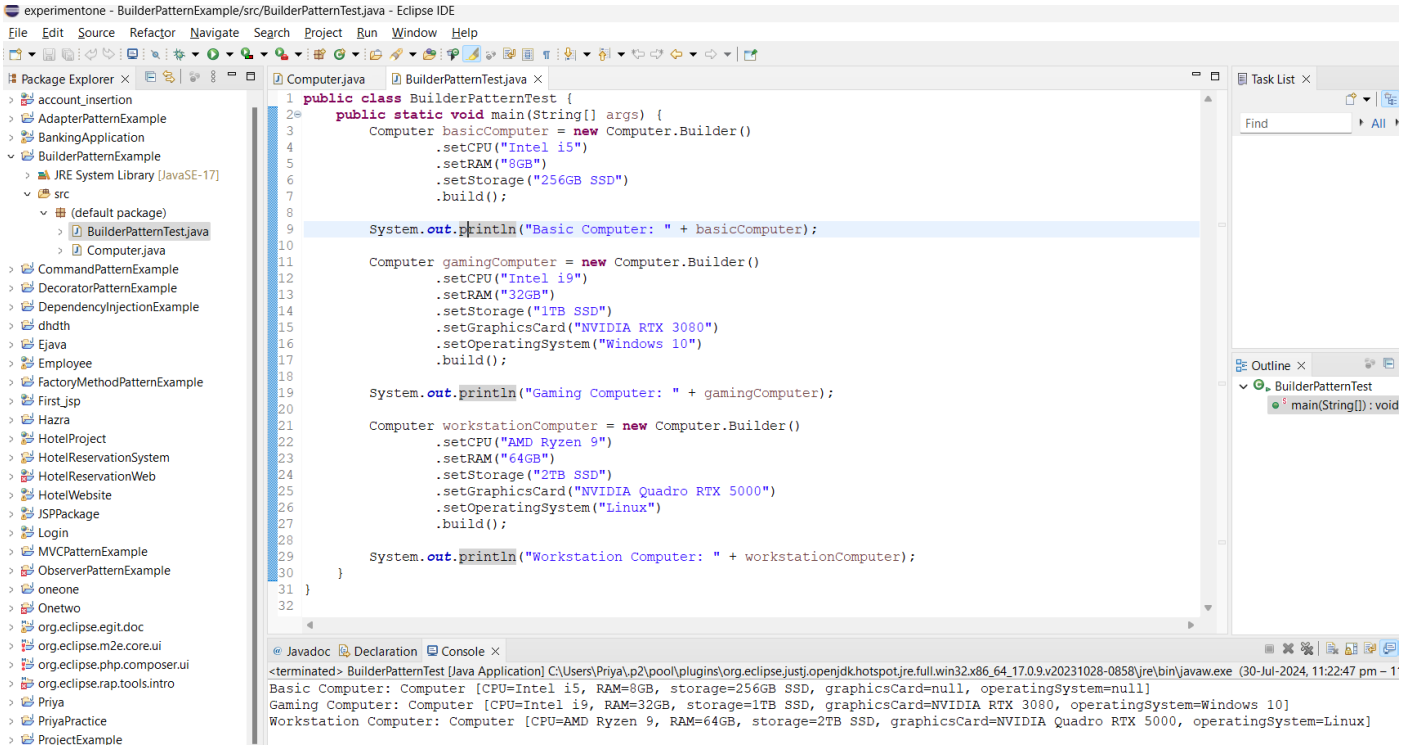
```
public class Computer {
   private String CPU;
   private String RAM;
   private String storage;
   private String graphicsCard;
   private String operatingSystem;
   private Computer(Builder builder) {
      this.CPU = builder.CPU;
      this.RAM = builder.RAM;
      this.storage = builder.storage;
      this.graphicsCard = builder.graphicsCard;
      this.operatingSystem = builder.operatingSystem;
   }
   @Override
   public String toString() {
      return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" + storage +
            ", graphicsCard=" + graphicsCard + ", operatingSystem=" + operatingSystem + "]";
   }
   public static class Builder {
      private String CPU;
      private String RAM;
      private String storage;
      private String graphicsCard;
      private String operatingSystem;

      public Builder setCPU(String CPU) {
         this.CPU = CPU;
```

```
        return this;
    }
    public Builder setRAM(String RAM) {
        this.RAM = RAM;
        return this;
    }
    public Builder setStorage(String storage) {
        this.storage = storage;
        return this;
    }
    public Builder setGraphicsCard(String graphicsCard) {
        this.graphicsCard = graphicsCard;
        return this;
    }
    public Builder setOperatingSystem(String operatingSystem) {
        this.operatingSystem = operatingSystem;
        return this;
    }
    public Computer build() {
        return new Computer(this);
    }
  }
}
```

BuilderPatternTest.java

```
public class BuilderPatternTest {
  public static void main(String[] args) {
    Computer basicComputer = new Computer.Builder()
        .setCPU("Intel i5")
        .setRAM("8GB")
        .setStorage("256GB SSD")
        .build();

    System.out.println("Basic Computer: " + basicComputer);

    Computer gamingComputer = new Computer.Builder()
        .setCPU("Intel i9")
        .setRAM("32GB")
        .setStorage("1TB SSD")
        .setGraphicsCard("NVIDIA RTX 3080")
        .setOperatingSystem("Windows 10")
        .build();

    System.out.println("Gaming Computer: " + gamingComputer);

    Computer workstationComputer = new Computer.Builder()
        .setCPU("AMD Ryzen 9")
        .setRAM("64GB")
        .setStorage("2TB SSD")
        .setGraphicsCard("NVIDIA Quadro RTX 5000")
        .setOperatingSystem("Linux")
        .build();
```

```
        System.out.println("Workstation Computer: " + workstationComputer);
    }}
```



Exercise 4: Implementing the Adapter Pattern

CODE:

AdapterPatternTest.java

```
public class AdapterPatternTest {
    public static void main(String[] args) {
        PayPal payPal = new PayPal();
        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);
        payPalProcessor.processPayment(100.00);
        Stripe stripe = new Stripe();
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
        stripeProcessor.processPayment(200.00);
        Paytm paytm = new Paytm();
        PaymentProcessor squareProcessor = new PaytmAdapter(paytm);
        squareProcessor.processPayment(300.00);
    }
}
```

PaymentProcessor.java

```
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

PayPal.java

```
public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of Rs" + amount + " through PayPal.");
    }
}
```

PayPalAdapter.java

```
public class PayPalAdapter implements PaymentProcessor {
   private PayPal payPal;
   public PayPalAdapter(PayPal payPal) {
      this.payPal = payPal;
   }
   @Override
   public void processPayment(double amount) {
      payPal.sendPayment(amount);
   }
}
```

Paytm.java

```
public class Paytm {
   public void pay(double amount) {
      System.out.println("Processing payment of Rs" + amount + " through Paytm.");
   }
}
```

PaytmAdapter.java

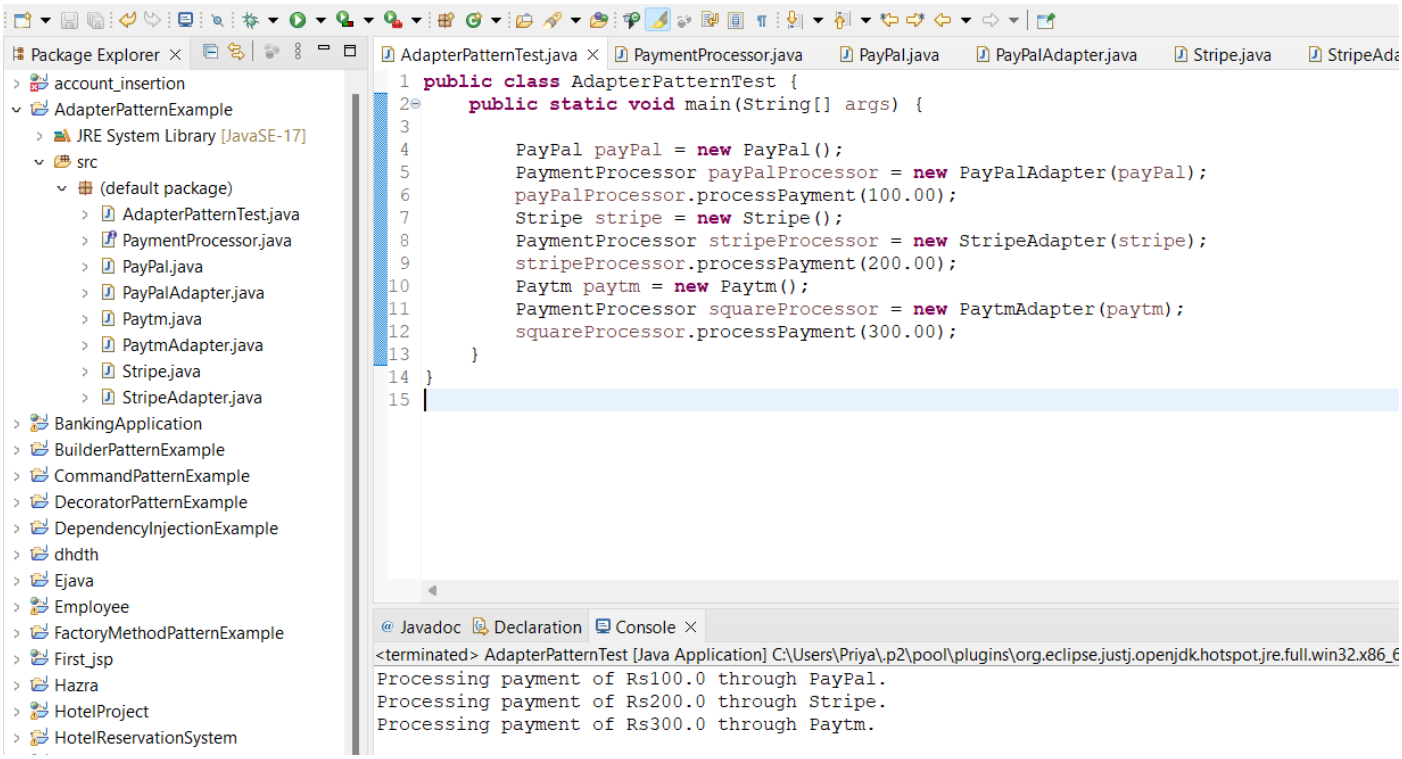
```
public class PaytmAdapter implements PaymentProcessor {
   private Paytm paytm;
   public PaytmAdapter(Paytm paytm) {
      this.paytm = paytm;
   }
   @Override
   public void processPayment(double amount) {
         paytm.pay(amount);
   }
}
```

Stripe.java

```
public class Stripe {
   public void makePayment(double amount) {
      System.out.println("Processing payment of Rs" + amount + " through Stripe.");
   }
}
```

StripeAdapter.java

```
public class StripeAdapter implements PaymentProcessor {
   private Stripe stripe;
   public StripeAdapter(Stripe stripe) {
      this.stripe = stripe;
   }
   @Override
   public void processPayment(double amount) {
      stripe.makePayment(amount);
   }
}
```



Exercise 5: Implementing the Decorator Pattern

CODE:

EmailNotifier.java

```
public class EmailNotifier implements Notifier {
   @Override
   public void send(String message) {
      System.out.println("Sending Email: " + message);
   }
}
```

Notifier.java

```
public interface Notifier {
   void send(String message);
}
```

NotifierDecorator.java

```
public abstract class NotifierDecorator implements Notifier {
   protected Notifier wrappedNotifier;

   public NotifierDecorator(Notifier notifier) {
      this.wrappedNotifier = notifier;
   }
   @Override
   public void send(String message) {
      wrappedNotifier.send(message);
   }
}
```

SlackNotifierDecorator.java

```
public class SlackNotifierDecorator extends NotifierDecorator {
   public SlackNotifierDecorator(Notifier notifier) {
      super(notifier);
   }
   @Override
```

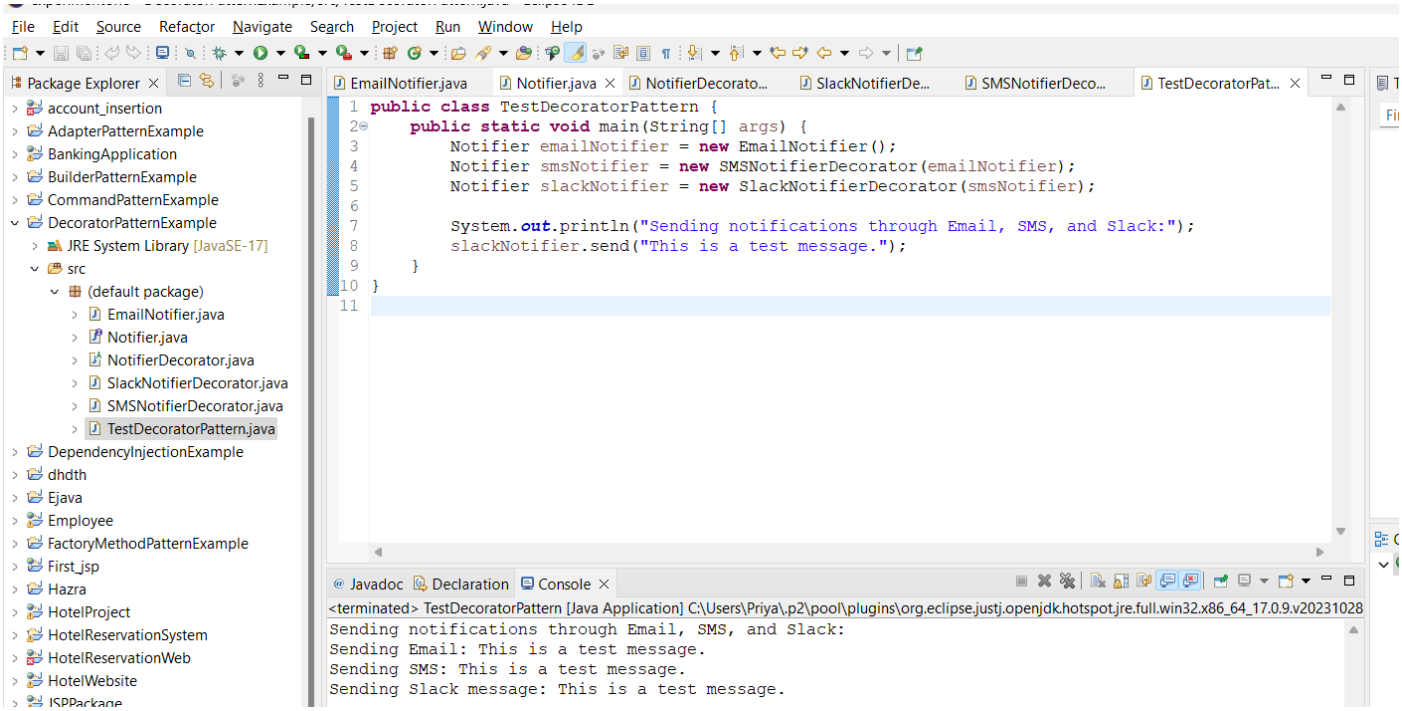
```
    public void send(String message) {
        super.send(message);
        sendSlackMessage(message);
    }
    private void sendSlackMessage(String message) {
        System.out.println("Sending Slack message: " + message);
    }
}
```

SmsNotifierDecorator.java

```
public class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    @Override
    public void send(String message) {
        super.send(message);
        sendSMS(message);
    }
    private void sendSMS(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
```

TestDecoratorPattern.java

```
public class TestDecoratorPattern {
    public static void main(String[] args) {
        Notifier emailNotifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
        System.out.println("Sending notifications through Email, SMS, and Slack:");
        slackNotifier.send("This is a test message.");
    }
}
```



Exercise 6: Implementing the Proxy Pattern

CODE:

Image.java

```
public interface Image {
   void display();
}
```

ProxyImage.java

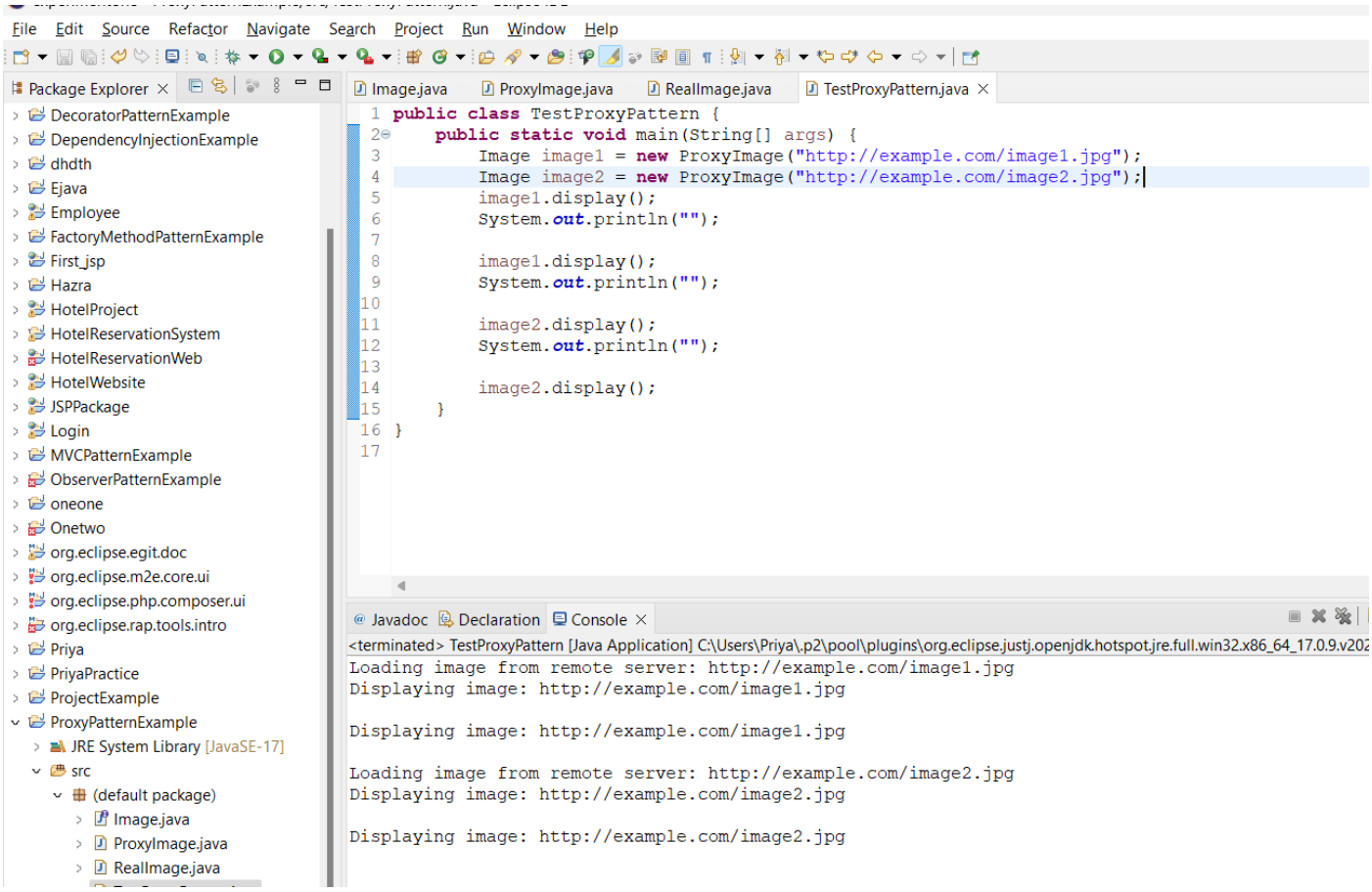
```
public class ProxyImage implements Image {
   private String imageUrl;
   private RealImage realImage;
   public ProxyImage(String imageUrl) {
      this.imageUrl = imageUrl;
   }
   @Override
   public void display() {
      if (realImage == null) {
         realImage = new RealImage(imageUrl);
      }
      realImage.display();
   }
}
```

RealImage.java

```
public class RealImage implements Image {
   private String imageUrl;
   public RealImage(String imageUrl) {
      this.imageUrl = imageUrl;
      loadImageFromRemoteServer();
   }
   private void loadImageFromRemoteServer() {
      System.out.println("Loading image from remote server: " + imageUrl);
   }
   @Override
   public void display() {
      System.out.println("Displaying image: " + imageUrl);
   }
}
```

TestProxyPattern.java

```
public class TestProxyPattern {
   public static void main(String[] args) {
      Image image1 = new ProxyImage("http://example.com/image1.jpg");
      Image image2 = new ProxyImage("http://example.com/image2.jpg");
      image1.display();
      System.out.println("");
      image1.display();
      System.out.println("");
      image2.display();
      System.out.println("");
      image2.display();
   }
}
```



Exercise 7: Implementing the Observer Pattern

Code:

MobileApp.java

```
public class MobileApp implements Observer {
   private String name;
   public MobileApp(String name) {
      this.name = name;
   }
   @Override
   public void update(double stockPrice) {
      System.out.println(name + " received stock price update: " + stockPrice);
   }
}
```

Observer.java

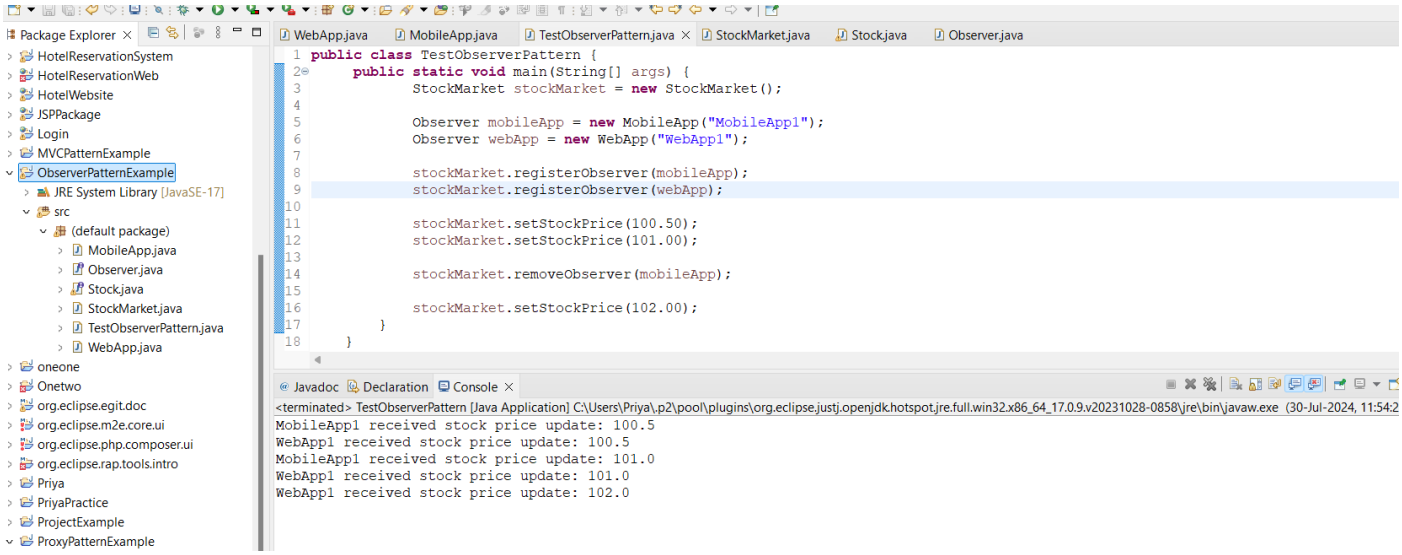
```
public interface Observer {
   void update(double stockPrice);
}
```

Stock.java

```
import java.util.ArrayList;
import java.util.List;
public interface Stock {
   void registerObserver(Observer observer);
   void removeObserver(Observer observer);
```

```
    void notifyObservers();
}
```
StockMarket.java
```
import java.util.ArrayList;
import java.util.List;
public class StockMarket implements Stock {
    private List<Observer> observers;
    private double stockPrice;
    public StockMarket() {
        observers = new ArrayList<>();
    }
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(stockPrice);
        }
    }
    public void setStockPrice(double stockPrice) {
        this.stockPrice = stockPrice;
        notifyObservers();
    }
}
```
TestObserverPattern.java
```
public class TestObserverPattern {
            public static void main(String[] args) {
                StockMarket stockMarket = new StockMarket();
                Observer mobileApp = new MobileApp("MobileApp1");
                Observer webApp = new WebApp("WebApp1");
                stockMarket.registerObserver(mobileApp);
                stockMarket.registerObserver(webApp);
                stockMarket.setStockPrice(100.50);
                stockMarket.setStockPrice(101.00);
                stockMarket.removeObserver(mobileApp);
                stockMarket.setStockPrice(102.00);
              }
            }
```
WebApp.java
```
public class WebApp implements Observer {
    private String name;
    public WebApp(String name) {
        this.name = name;
    }
    @Override
    public void update(double stockPrice) {
        System.out.println(name + " received stock price update: " + stockPrice);
```

```
    }
}
```



Exercise 8: Implementing the Strategy Pattern

Code:

CreditCardPayment.java

```
public class CreditCardPayment implements PaymentStrategy {
    private String name;
    private String cardNumber;

    public CreditCardPayment(String name, String cardNumber) {
        this.name = name;
        this.cardNumber = cardNumber;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}
```

PaymentContent.java

```
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public PaymentContext(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void executePayment(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

PaymentStrategy.java

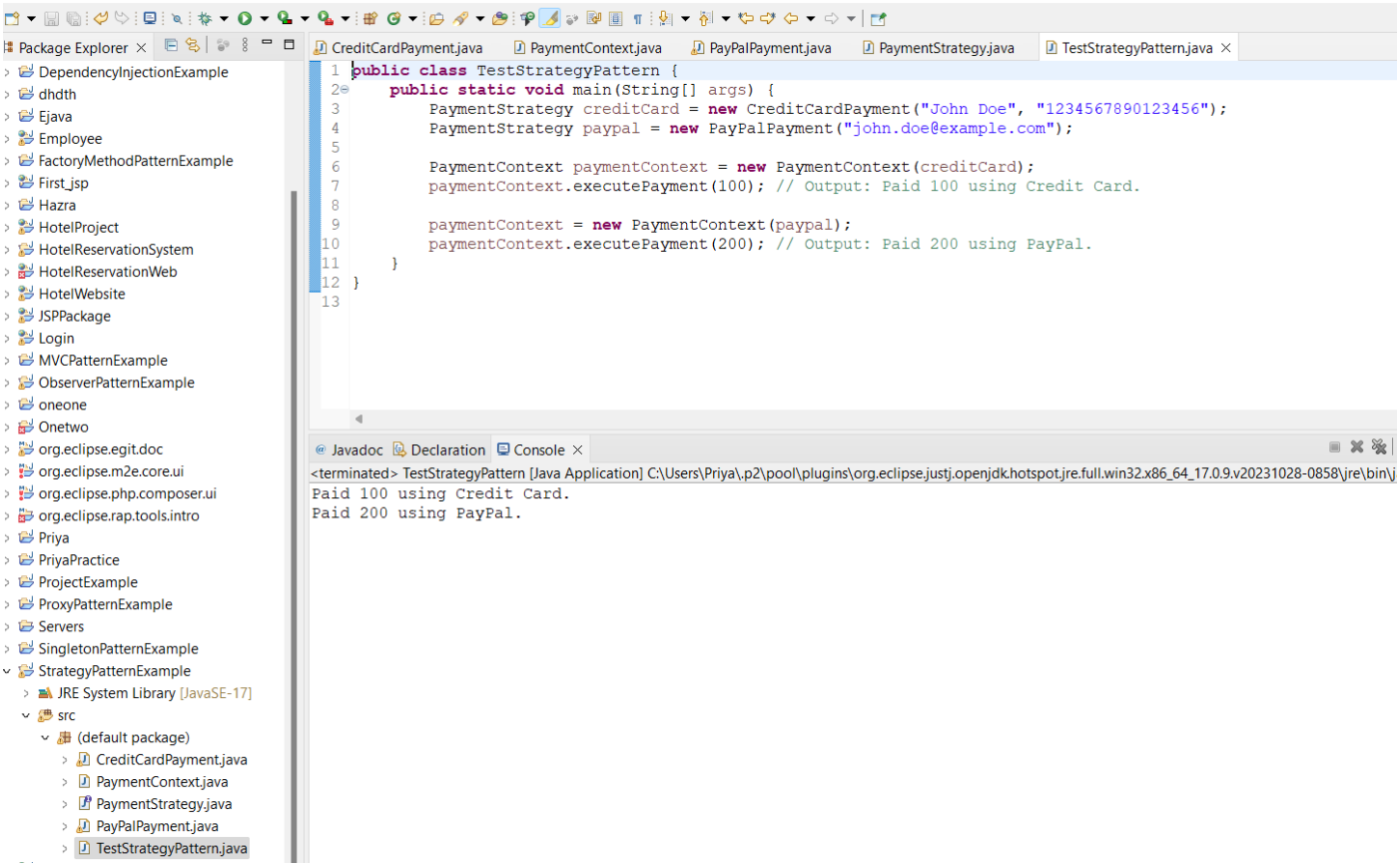
```
public interface PaymentStrategy {
    void pay(int amount);
}
```

PayPalPayment.java

```
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

TestStrategyPattern.java

```
public class TestStrategyPattern {
    public static void main(String[] args) {
        PaymentStrategy creditCard = new CreditCardPayment("John Doe", "1234567890123456");
        PaymentStrategy paypal = new PayPalPayment("john.doe@example.com");

        PaymentContext paymentContext = new PaymentContext(creditCard);
        paymentContext.executePayment(100); // Output: Paid 100 using Credit Card.

        paymentContext = new PaymentContext(paypal);
        paymentContext.executePayment(200); // Output: Paid 200 using PayPal.
    }
}
```



Exercise 9: Implementing the Command Pattern

Code:

Command.java

```
public interface Command {
   void execute();
}
```

Light.java

```
public class Light {
   public void turnOn() {
      System.out.println("The light is ON");
   }

   public void turnOff() {
      System.out.println("The light is OFF");
   }
}
```

LightOffCommand.java

```
public class LightOffCommand implements Command {
   private Light light;

   public LightOffCommand(Light light) {
      this.light = light;
   }

   @Override
   public void execute() {
      light.turnOff();
   }
}
```

LightOnCommand.java

```
public class LightOnCommand implements Command {
   private Light light;

   public LightOnCommand(Light light) {
      this.light = light;
   }

   @Override
   public void execute() {
      light.turnOn();
   }
}
```

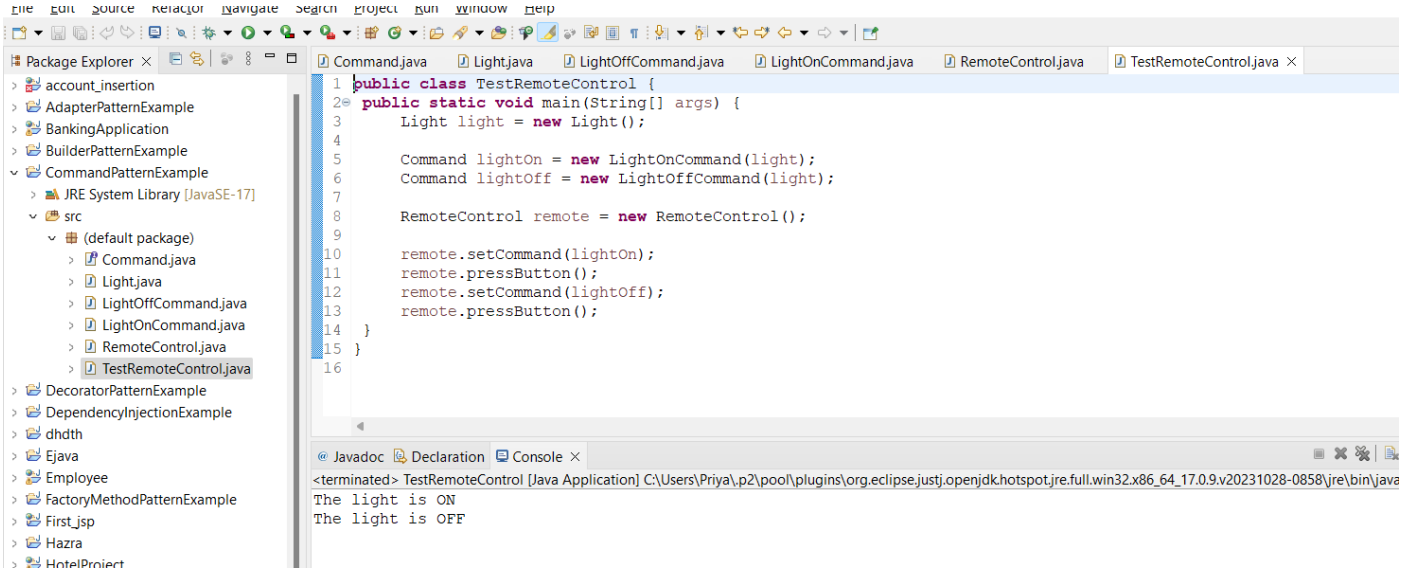
RemoteControl.java

```
public class RemoteControl {
   private Command command;
```

```
    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

TestRemoteControl.java

```
public class TestRemoteControl {
 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();
 }
}
```



Exercise 10: Implementing the MVC Pattern

Code:

Main.java

```
public class Main {
    public static void main(String[] args) {
        Student student = new Student("PRIYA HAZRA", "2101020271", "A");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(student, view);
        controller.updateView();
        controller.setStudentName("PRIYA HAZRA");
```

```
        controller.setStudentId("2101020271");
        controller.setStudentGrade("B");

        controller.updateView();
    }
}
```

Student.java

```
public class Student {
    private String name;
    private String id;
    private String grade;
    public Student(String name, String id, String grade) {
        this.name = name;
        this.id = id;
        this.grade = 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;
    }
}
```

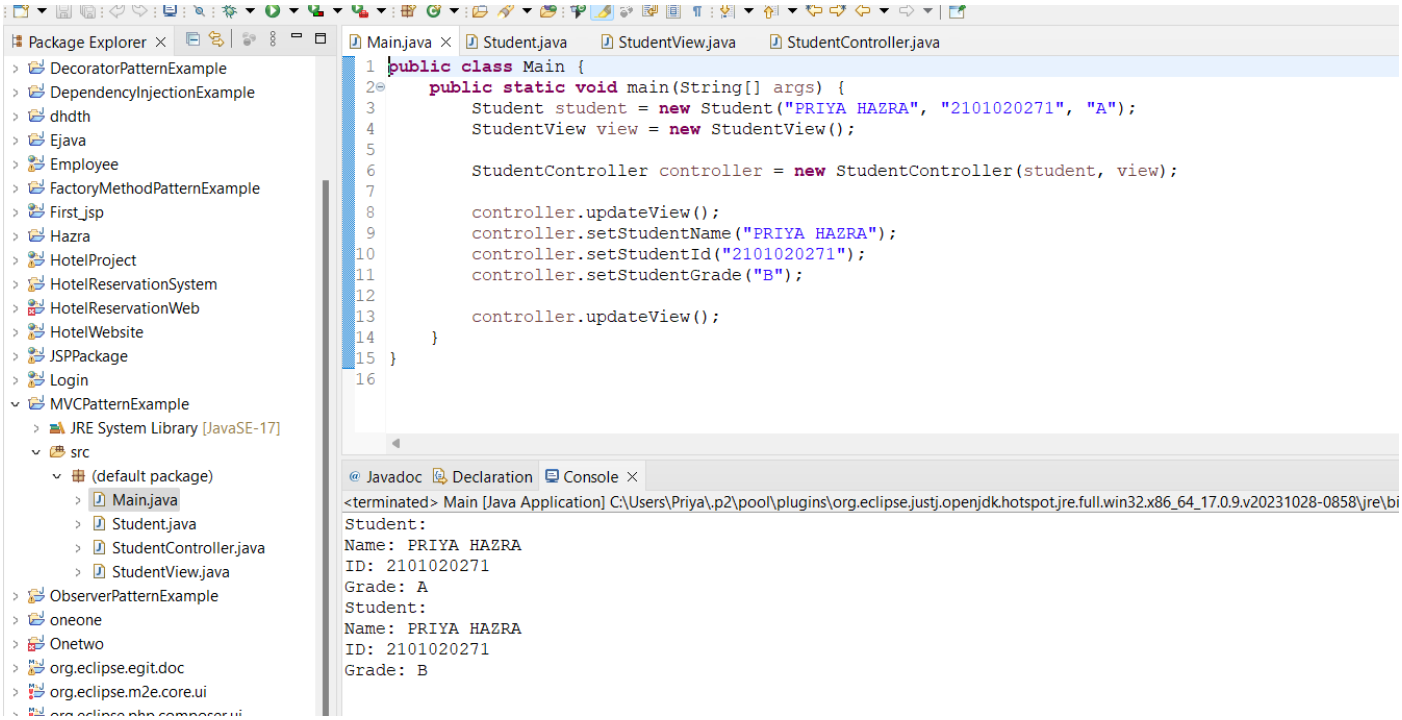
StudentController.java

```
public class StudentController {
    private Student model;
    private StudentView view;
    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }
    public void updateView() {
        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());
    }
    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();
    }
}
```

StudentView.java

```
public class StudentView {
    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("ID: " + studentId);
        System.out.println("Grade: " + studentGrade);
    }
}
```



Exercise 11: Implementing Dependency Injection

Code:

CustomerRepository.java

```
public interface CustomerRepository {
    String findCustomerById(int id);
}
```

CustomerRepositoryIMPL.java

```
public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
```

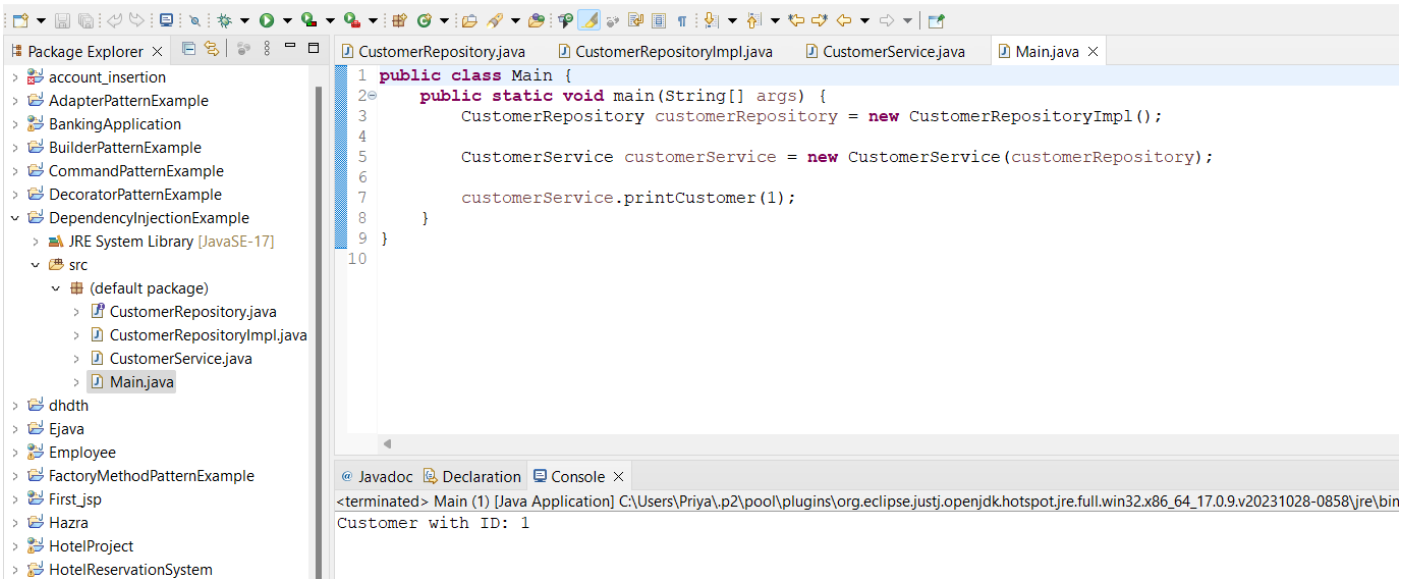
```
    public String findCustomerById(int id) {
        return "Customer with ID: " + id;
    }
}
```

CustomerService.java

```
public class CustomerService {
    private CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public void printCustomer(int id) {
        String customer = customerRepository.findCustomerById(id);
        System.out.println(customer);
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        CustomerService customerService = new CustomerService(customerRepository);

        customerService.printCustomer(1);
    }
}
```

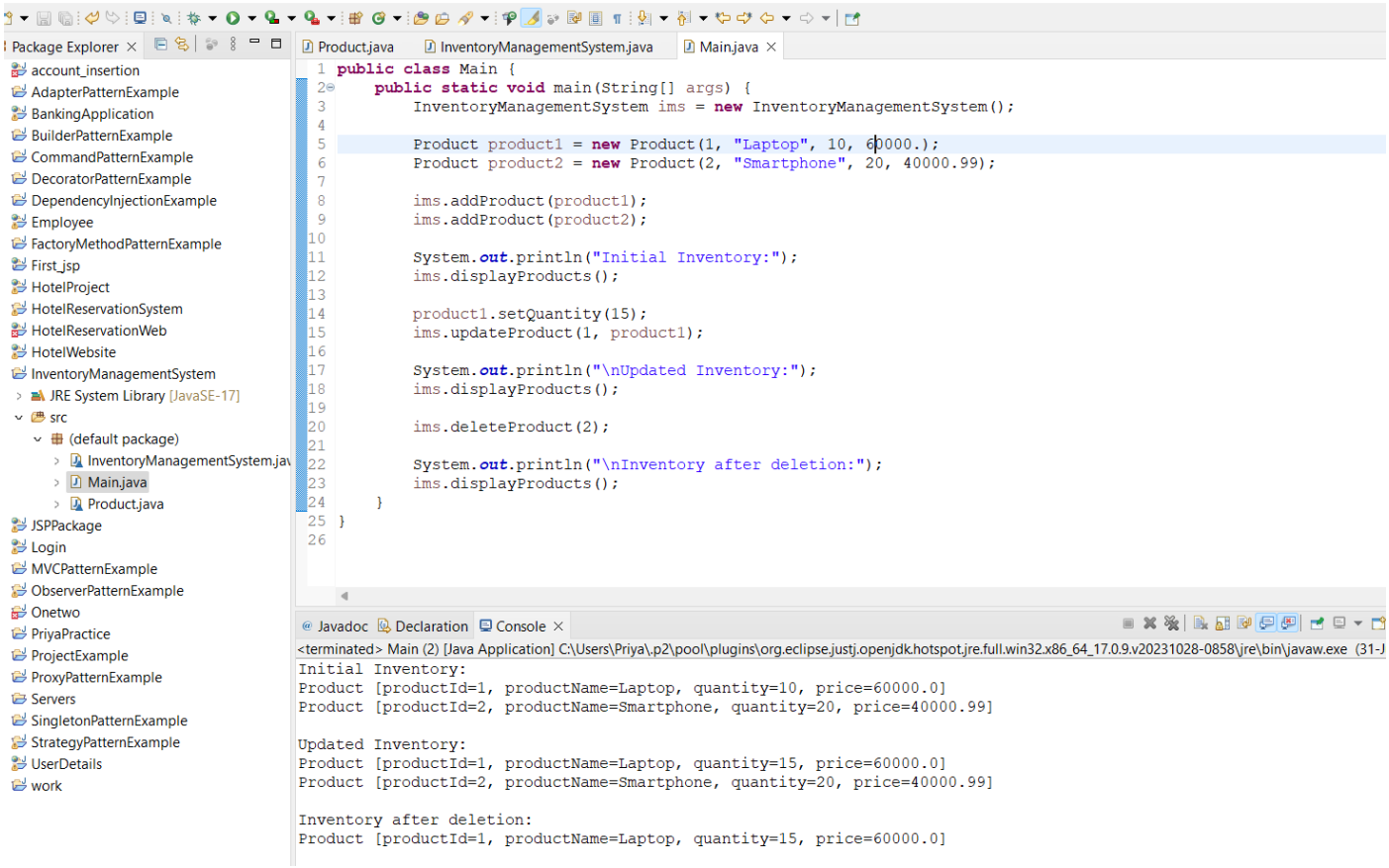


Week 1_Algorthms_Data Structures

Exercise 1: Inventory Management System

Data structures and algorithms improve performance, scalability, and resource management in inventory systems.

Key structures include **ArrayList** for fast indexed access, **HashMap** for quick retrieval with unique keys, and **TreeMap** for sorted keys with log-time operations.



CODE:

Product.java

```
class Product {
   private int productId;
   private String productName;
   private int quantity;
   private double price;

   public Product(int productId, String productName, int quantity, double price) {
      this.productId = productId;
      this.productName = productName;
      this.quantity = quantity;
      this.price = price;
   }
   public int getProductId() {
      return productId;
   }
   public void setProductId(int productId) {
      this.productId = productId;
   }
   public String getProductName() {
```

```
    return productName;
  }
  public void setProductName(String productName) {
    this.productName = productName;
  }
  public int getQuantity() {
    return quantity;
  }
  public void setQuantity(int quantity) {
    this.quantity = quantity;
  }
  public double getPrice() {
    return price;
  }
  public void setPrice(double price) {
    this.price = price;
  }
  @Override
  public String toString() {
    return "Product [productId=" + productId + ", productName=" + productName + ", quantity=" + quantity + ",
price=" + price + "]";
  }
}
```

InventoryManagementSystem.java

```
import java.util.HashMap;
import java.util.Map;

class InventoryManagementSystem {
  private Map<Integer, Product> inventory;

  public InventoryManagementSystem() {
    this.inventory = new HashMap<>();
  }
  public void addProduct(Product product) {
    inventory.put(product.getProductId(), product);
  }
  public void updateProduct(int productId, Product updatedProduct) {
    if (inventory.containsKey(productId)) {
      inventory.put(productId, updatedProduct);
    } else {
      System.out.println("Product not found!");
    }
  }
  public void deleteProduct(int productId) {
    if (inventory.containsKey(productId)) {
      inventory.remove(productId);
    } else {
      System.out.println("Product not found!");
    }
  }
  public void displayProducts() {
    for (Product product : inventory.values()) {
```

```
        System.out.println(product);
      }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        InventoryManagementSystem ims = new InventoryManagementSystem();

        Product product1 = new Product(1, "Laptop", 10, 60000.);
        Product product2 = new Product(2, "Smartphone", 20, 40000.99);

        ims.addProduct(product1);
        ims.addProduct(product2);

        System.out.println("Initial Inventory:");
        ims.displayProducts();

        product1.setQuantity(15);
        ims.updateProduct(1, product1);

        System.out.println("\nUpdated Inventory:");
        ims.displayProducts();

        ims.deleteProduct(2);

        System.out.println("\nInventory after deletion:");
        ims.displayProducts();
    }
}
```

Time Complexity

Using a **HashMap**:

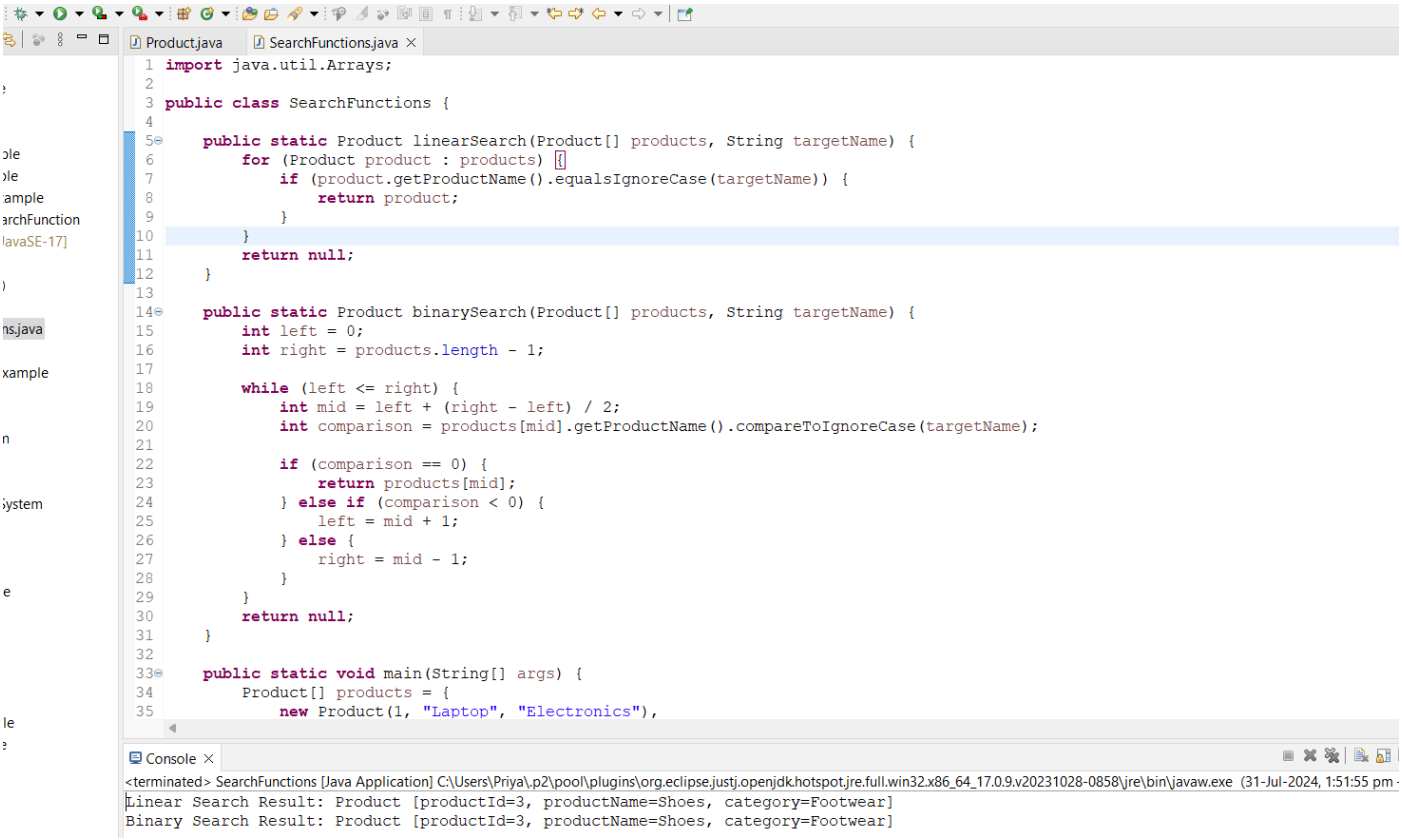
- **Adding a Product**: O(1) on average
- **Updating a Product**: O(1) on average
- **Deleting a Product**: O(1) on average

Optimization: Adjusting the load factor, implementing batch operations, and using indexing methods like B-trees for large datasets.

Exercise 2: E-commerce Platform Search Function

Asymptotic Notation:

- **Big O Notation**: Describes algorithm efficiency based on input size.
    - O(1): Constant time
    - O(n): Linear time
    - O(log n): Logarithmic time
    - O(n^2): Quadratic time



CODE:

SearchFunctions.java

```
import java.util.Arrays;

public class SearchFunctions {

    public static Product linearSearch(Product[] products, String targetName) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(targetName)) {
                return product;
            }
        }
        return null;
    }

    public static Product binarySearch(Product[] products, String targetName) {
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = products[mid].getProductName().compareToIgnoreCase(targetName);

            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
```

```
        return null;
    }

    public static void main(String[] args) {
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Smartphone", "Electronics"),
            new Product(3, "Shoes", "Footwear"),
            new Product(4, "T-shirt", "Clothing")
        };

        Product foundProduct = linearSearch(products, "Shoes");
        System.out.println("Linear Search Result: " + foundProduct);

        Arrays.sort(products, (p1, p2) -> p1.getProductName().compareToIgnoreCase(p2.getProductName()));
        foundProduct = binarySearch(products, "Shoes");
        System.out.println("Binary Search Result: " + foundProduct);
    }
}
```

Product.java

```
public class Product {
    private int productId;
    private String productName;
    private String category;
    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }
    public int getProductId() {
        return productId;
    }
    public String getProductName() {
        return productName;
    }
    public String getCategory() {
        return category;
    }
    @Override
    public String toString() {
        return "Product [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";
    }
}
```

Time Complexity:

- Linear Search:
    - Best Case: O(1) (target is the first element)
    - Average Case: O(n) (target is in the middle)
    - Worst Case: O(n) (target is last or not found)
- Binary Search:
    - Best Case: O(1) (target is the middle element)

- Average Case: O(log n)
- Worst Case: O(log n)

Algorithm Suitability:

- Linear Search: Simple but inefficient for large datasets.
- Binary Search: Efficient for large, sorted datasets; requires sorting (O(n log n)) if not pre-sorted.

Exercise 3: Sorting Customer Orders

Sorting Algorithms:

- Bubble Sort:
  - Description: Compares and swaps adjacent elements repeatedly.
  - Complexity: Best: O(n), Average/Worst: O(n^2)
- Insertion Sort:
  - Description: Builds sorted array by inserting each element into its correct position.
  - Complexity: Best: O(n), Average/Worst: O(n^2)
- Quick Sort:
  - Description: Divides and sorts based on a pivot element.
  - Complexity: Best/Average: O(n log n), Worst: O(n^2)
- Merge Sort:
  - Description: Divides, sorts, and merges arrays.
  - Complexity: Best/Average/Worst: O(n log n)

Order.java SortingAlgorithms.java Main.java

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Order[] orders = {
            new Order(1, "Alice", 300.50),
            new Order(2, "Bob", 150.75),
            new Order(3, "Charlie", 250.00),
            new Order(4, "Daisy", 100.00)
        };

        System.out.println("Original Orders:");
        Arrays.stream(orders).forEach(System.out::println);

        // Bubble Sort
        SortingAlgorithms.bubbleSort(orders);
        System.out.println("\nOrders after Bubble Sort:");
        Arrays.stream(orders).forEach(System.out::println);

        // Quick Sort
```

Javadoc Declaration Console

<terminated> Main (3) [Java Application] C:\Users\Priya\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre

```
Original Orders:
Order [orderId=1, customerName=Alice, totalPrice=300.5]
Order [orderId=2, customerName=Bob, totalPrice=150.75]
Order [orderId=3, customerName=Charlie, totalPrice=250.0]
Order [orderId=4, customerName=Daisy, totalPrice=100.0]

Orders after Bubble Sort:
Order [orderId=4, customerName=Daisy, totalPrice=100.0]
Order [orderId=2, customerName=Bob, totalPrice=150.75]
Order [orderId=3, customerName=Charlie, totalPrice=250.0]
Order [orderId=1, customerName=Alice, totalPrice=300.5]

Orders after Quick Sort:
Order [orderId=4, customerName=Daisy, totalPrice=100.0]
Order [orderId=2, customerName=Bob, totalPrice=150.75]
Order [orderId=3, customerName=Charlie, totalPrice=250.0]
Order [orderId=1, customerName=Alice, totalPrice=300.5]
```

CODE:

Order.java

```
public class Order {
   private int orderId;
   private String customerName;
   private double totalPrice;

   public Order(int orderId, String customerName, double totalPrice) {
      this.orderId = orderId;
      this.customerName = customerName;
      this.totalPrice = totalPrice;
   }

   public int getOrderId() {
      return orderId;
   }

   public String getCustomerName() {
      return customerName;
   }

   public double getTotalPrice() {
      return totalPrice;
   }

   @Override
   public String toString() {
      return "Order [orderId=" + orderId + ", customerName=" + customerName + ", totalPrice=" + totalPrice + "]";
   }
}
```

SortingAlogorithms.java

```
public class SortingAlgorithms {
        public static void bubbleSort(Order[] orders) {
            int n = orders.length;
            for (int i = 0; i < n - 1; i++) {
               for (int j = 0; j < n - i - 1; j++) {
                  if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

                     Order temp = orders[j];
                     orders[j] = orders[j + 1];
                     orders[j + 1] = temp;
                  }
               }
            }
         }

   public static void quickSort(Order[] orders, int low, int high) {
      if (low < high) {
         int pi = partition(orders, low, high);
         quickSort(orders, low, pi - 1);
         quickSort(orders, pi + 1, high);
      }
```

```
    }

    private static int partition(Order[] orders, int low, int high) {
        double pivot = orders[high].getTotalPrice();
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (orders[j].getTotalPrice() <= pivot) {
                i++;
                Order temp = orders[i];
                orders[i] = orders[j];
                orders[j] = temp;
            }
        }
        Order temp = orders[i + 1];
        orders[i + 1] = orders[high];
        orders[high] = temp;

        return i + 1;
    }
}
```

Main.java

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Order[] orders = {
            new Order(1, "Alice", 300.50),
            new Order(2, "Bob", 150.75),
            new Order(3, "Charlie", 250.00),
            new Order(4, "Daisy", 100.00)
        };

        System.out.println("Original Orders:");
        Arrays.stream(orders).forEach(System.out::println);

        // Bubble Sort
        SortingAlgorithms.bubbleSort(orders);
        System.out.println("\nOrders after Bubble Sort:");
        Arrays.stream(orders).forEach(System.out::println);

        // Quick Sort
        Order[] ordersForQuickSort = {
            new Order(1, "Alice", 300.50),
            new Order(2, "Bob", 150.75),
            new Order(3, "Charlie", 250.00),
            new Order(4, "Daisy", 100.00)
        };

        SortingAlgorithms.quickSort(ordersForQuickSort, 0, ordersForQuickSort.length - 1);
        System.out.println("\nOrders after Quick Sort:");
        Arrays.stream(ordersForQuickSort).forEach(System.out::println);
    }
```

}

Time Complexity:

- Bubble Sort:
  - Best: O(n), Average/Worst: O(n^2)
- Quick Sort:
  - Best/Average: O(n log n), Worst: O(n^2) (with poor pivot choice)

Why Quick Sort is Preferred:

- Efficiency: Faster average performance (O(n log n)).
- Space Usage: More space-efficient (in-place sort).
- Practical Performance: Better for large datasets despite worst-case scenario.

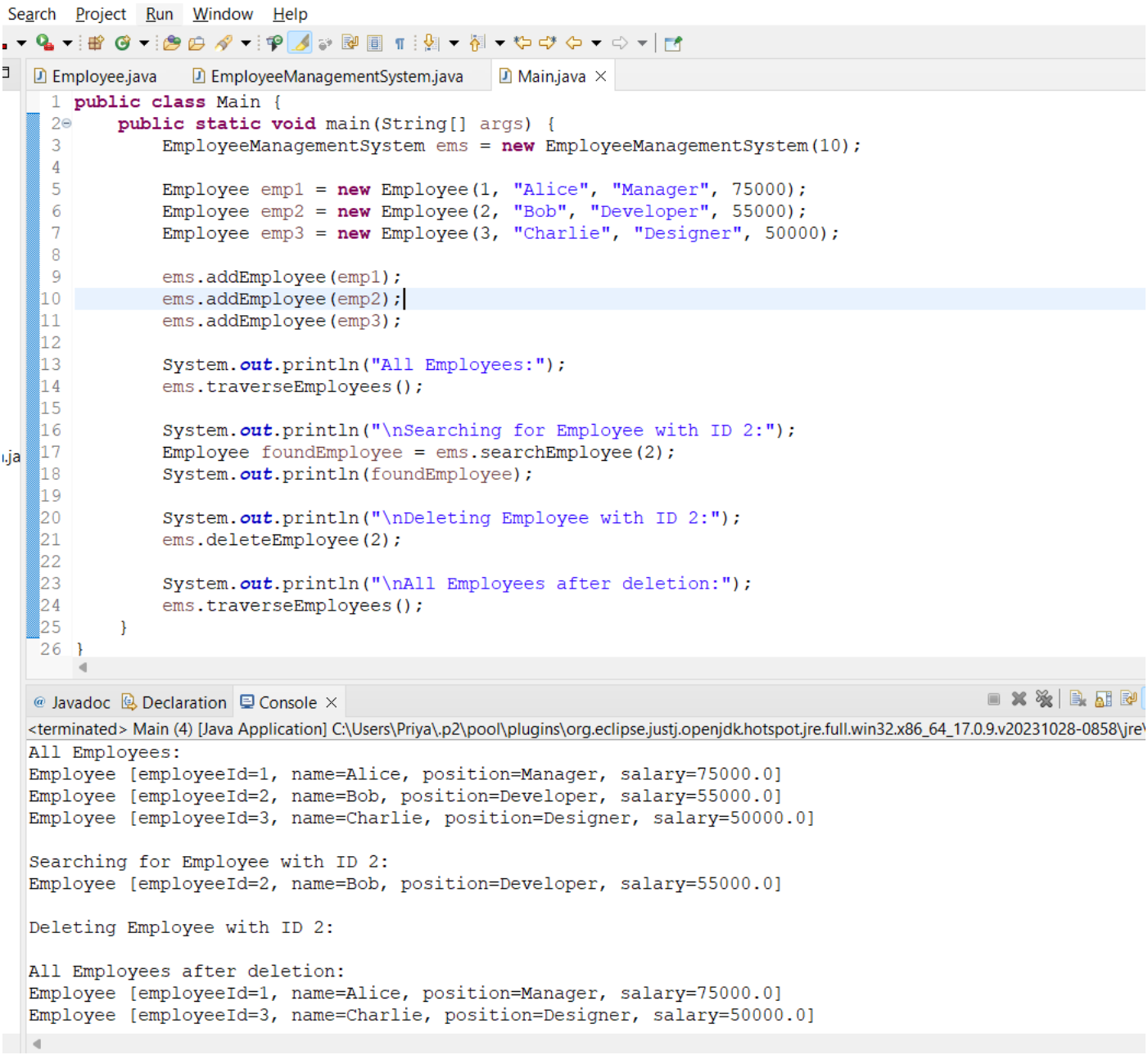
Exercise 4: Employee Management System

Array Representation in Memory:

- Storage: Elements are stored in contiguous memory locations.
- Address Calculation: Uses base address and element size for address computation.

Advantages:

- Random Access: O(1) time for direct access.
- Cache Friendliness: Contiguous memory improves performance.
- Ease of Iteration: Efficient traversal.



CODE:

Employee.java

```
public class Employee {
   private int employeeId;
   private String name;
   private String position;
   private double salary;

   public Employee(int employeeId, String name, String position, double salary) {
      this.employeeId = employeeId;
      this.name = name;
      this.position = position;
      this.salary = salary;
   }

   // Getters and setters
   public int getEmployeeId() {
      return employeeId;
```

```
    }

    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee [employeeId=" + employeeId + ", name=" + name + ", position=" + position + ", salary=" +
salary + "]";
    }
}
```

EmployeeManagementSystem.java

```
public class EmployeeManagementSystem {
    private Employee[] employees;
    private int size;
    private int capacity;

    public EmployeeManagementSystem(int capacity) {
        this.capacity = capacity;
        this.employees = new Employee[capacity];
        this.size = 0;
    }

    public void addEmployee(Employee employee) {
        if (size < capacity) {
            employees[size] = employee;
            size++;
```

```
        } else {
            System.out.println("Cannot add employee, array is full.");
        }
    }

    public Employee searchEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                return employees[i];
            }
        }
        return null;
    }

    public void traverseEmployees() {
        for (int i = 0; i < size; i++) {
            System.out.println(employees[i]);
        }
    }

    public void deleteEmployee(int employeeId) {
        int index = -1;
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                index = i;
                break;
            }
        }

        if (index != -1) {
            for (int i = index; i < size - 1; i++) {
                employees[i] = employees[i + 1];
            }
            employees[size - 1] = null;
            size--;
        } else {
            System.out.println("Employee not found.");
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        EmployeeManagementSystem ems = new EmployeeManagementSystem(10);

        Employee emp1 = new Employee(1, "Alice", "Manager", 75000);
        Employee emp2 = new Employee(2, "Bob", "Developer", 55000);
        Employee emp3 = new Employee(3, "Charlie", "Designer", 50000);

        ems.addEmployee(emp1);
        ems.addEmployee(emp2);
        ems.addEmployee(emp3);
```

```
        System.out.println("All Employees:");
        ems.traverseEmployees();

        System.out.println("\nSearching for Employee with ID 2:");
        Employee foundEmployee = ems.searchEmployee(2);
        System.out.println(foundEmployee);

        System.out.println("\nDeleting Employee with ID 2:");
        ems.deleteEmployee(2);

        System.out.println("\nAll Employees after deletion:");
        ems.traverseEmployees();
    }
}
```

Time Complexity:

- Add Operation: O(1) (at the end)
- Search Operation: O(n) (worst case)
- Traverse Operation: O(n)
- Delete Operation: O(n) (due to element shifting)

Limitations:

- Fixed Size: Costly resizing.
- Inefficient Deletion: Element shifting is inefficient.
- Memory Allocation: Contiguous memory can be problematic.

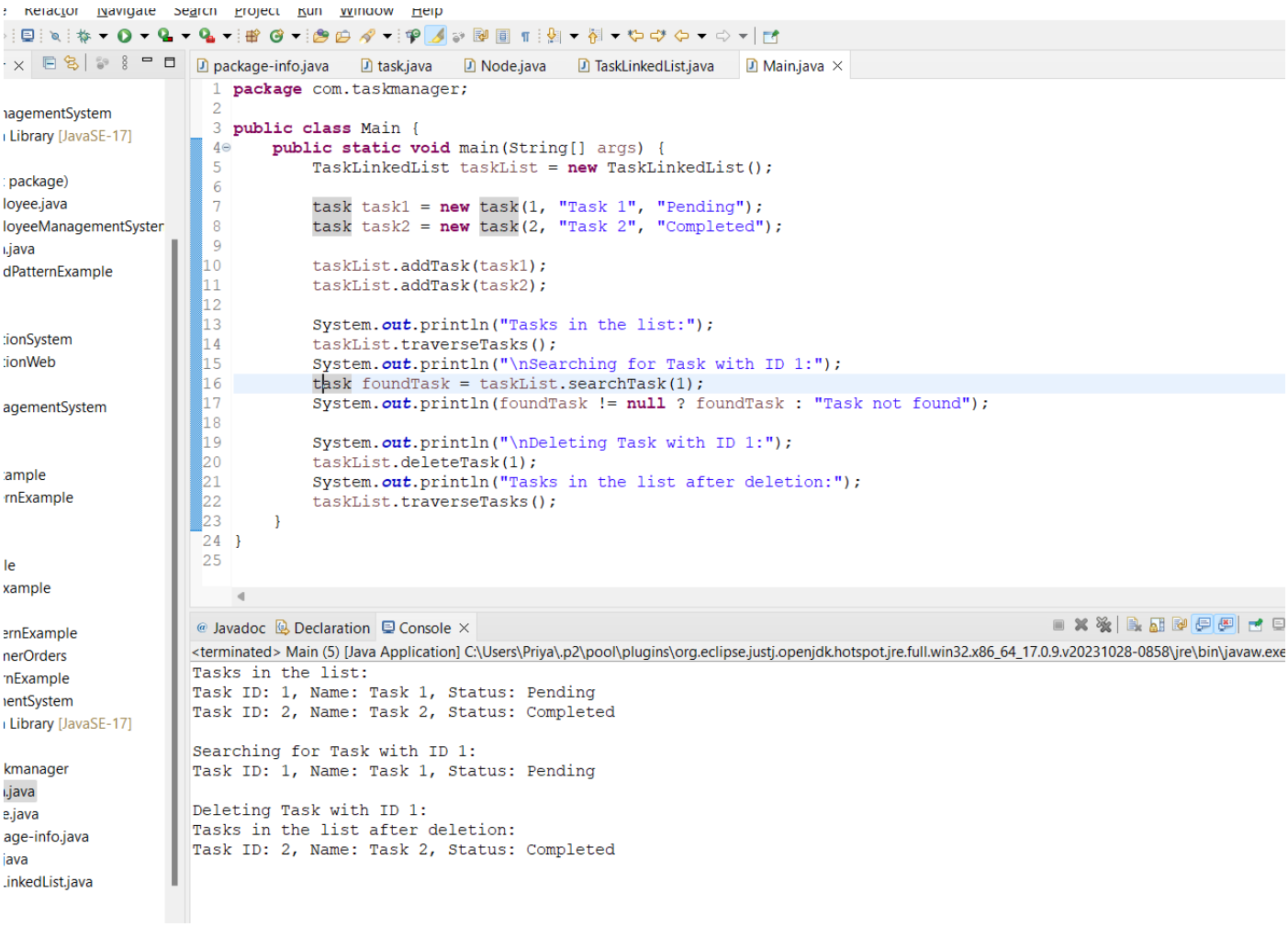
When to Use Arrays:

- Small to Medium Datasets: Efficient with size constraints.
- Random Access: Ideal for direct access.
- Simple Iteration: Suitable for efficient traversal.

Exercise 5: Task Management System

A singly linked list allows one-way traversal with each node pointing to the next, while a doubly linked list allows two-way traversal with nodes pointing to both the next and previous nodes.

In a linked list, adding, searching, traversing, and deleting tasks each have a worst-case time complexity of O(n) due to traversal requirements. Linked lists offer advantages over arrays with dynamic sizing, more efficient insertions/deletions, and better memory utilization by allocating memory only for needed nodes.



CODE:

Package-info.java

```
package com.taskmanager;
```

Node.java

```
package com.taskmanager;

public class Node {
   task task;
   Node next;

   public Node(task task) {
      this.task = task;
      this.next = null;
   }
}
```

task.java

```
package com.taskmanager;

public class task {
   private int taskId;
   private String taskName;
   private String status;

   public task(int taskId, String taskName, String status) {
      this.taskId = taskId;
```

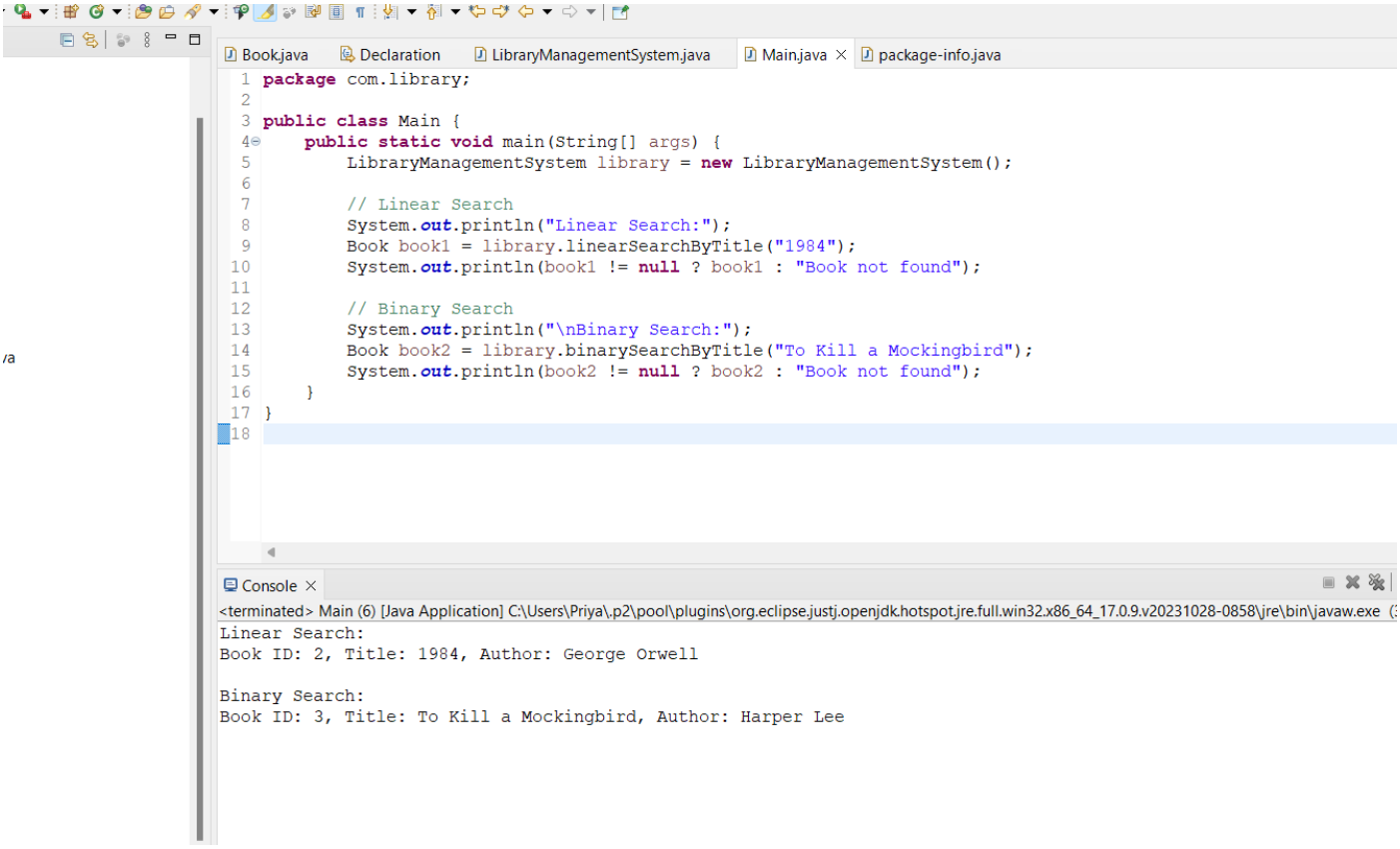
```
        this.taskName = taskName;
        this.status = status;
    }

    public int getTaskId() {
        return taskId;
    }

    public String getTaskName() {
        return taskName;
    }

    public String getStatus() {
        return status;
    }

    @Override
    public String toString() {
        return "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;
    }
}
```

Main.java

```
package com.taskmanager;

public class Main {
    public static void main(String[] args) {
        TaskLinkedList taskList = new TaskLinkedList();

        task task1 = new task(1, "Task 1", "Pending");
        task task2 = new task(2, "Task 2", "Completed");

        taskList.addTask(task1);
        taskList.addTask(task2);

        System.out.println("Tasks in the list:");
        taskList.traverseTasks();
        System.out.println("\nSearching for Task with ID 1:");
        task foundTask = taskList.searchTask(1);
        System.out.println(foundTask != null ? foundTask : "Task not found");

        System.out.println("\nDeleting Task with ID 1:");
        taskList.deleteTask(1);
        System.out.println("Tasks in the list after deletion:");
        taskList.traverseTasks();
    }
}
```

Exercise 6: Library Management System

Linear search checks each element one by one with a time complexity of O(n) and is suitable for unsorted or small datasets. Binary search, with a time complexity of O(log n), efficiently finds elements in a sorted array by dividing the search interval in half, making it ideal for large, sorted datasets.



Code:
Book.java

```
package com.library;

public class Book {
    private int bookId;
    private String title;
    private String author;

    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    @Override
    public String toString() {
        return "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;
    }
```

```
}
```

LibraryManagementSystem.java

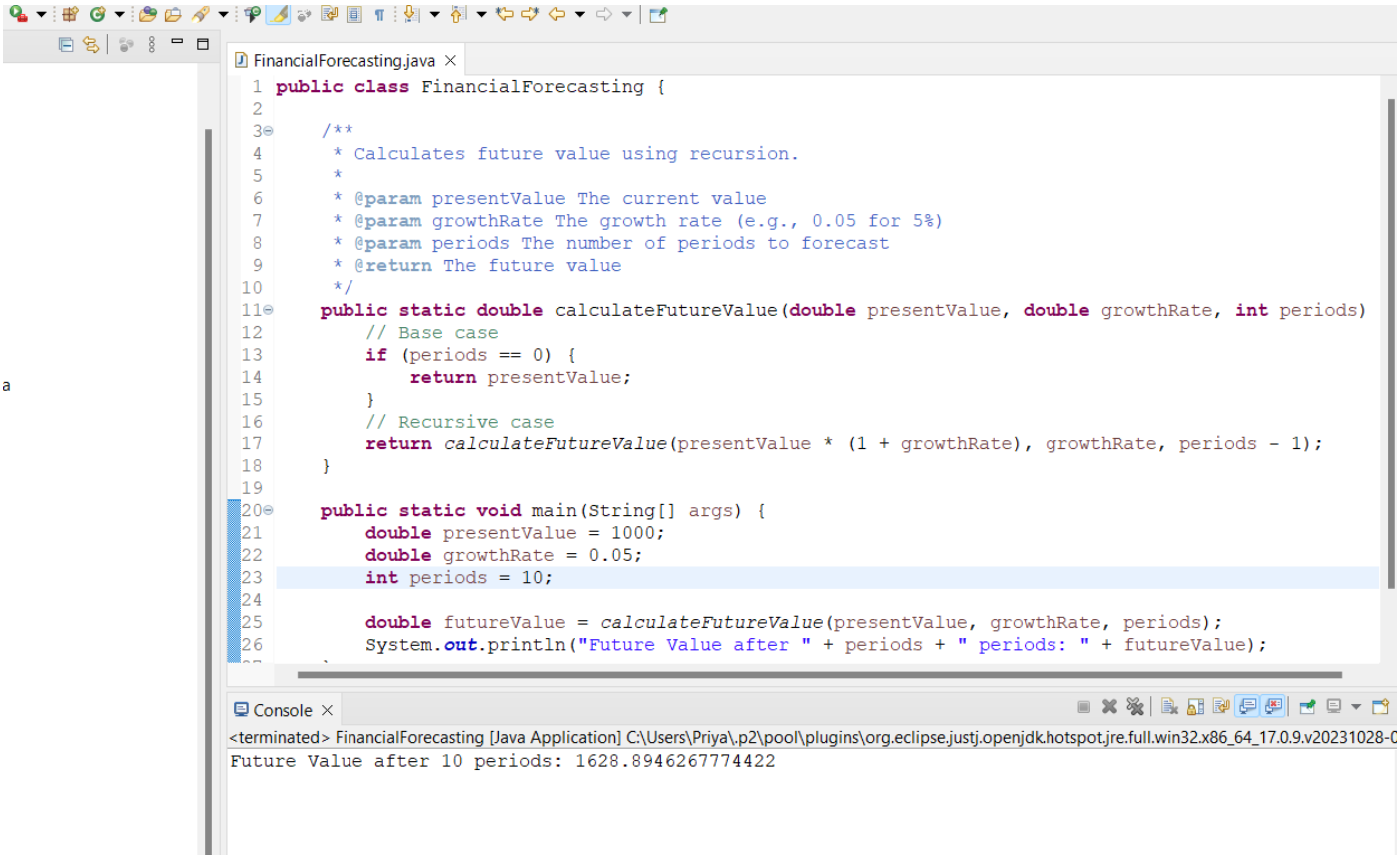
```
package com.library;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LibraryManagementSystem {
    private List<Book> books;

    public LibraryManagementSystem() {
        books = new ArrayList<>();

        books.add(new Book(1, "The Great Gatsby", "F. Scott Fitzgerald"));
        books.add(new Book(2, "1984", "George Orwell"));
        books.add(new Book(3, "To Kill a Mockingbird", "Harper Lee"));
    }

    public Book linearSearchByTitle(String title) {
        for (Book book : books) {
            if (book.getTitle().equalsIgnoreCase(title)) {
                return book;
            }
        }
        return null;
    }

    public Book binarySearchByTitle(String title) {
        Collections.sort(books, Comparator.comparing(Book::getTitle));
        int left = 0;
        int right = books.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            Book midBook = books.get(mid);
            int comparison = midBook.getTitle().compareToIgnoreCase(title);

            if (comparison == 0) {
                return midBook;
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}

package com.library;
```

```
public class Main {
    public static void main(String[] args) {
        LibraryManagementSystem library = new LibraryManagementSystem();

        // Linear Search
        System.out.println("Linear Search:");
        Book book1 = library.linearSearchByTitle("1984");
        System.out.println(book1 != null ? book1 : "Book not found");

        // Binary Search
        System.out.println("\nBinary Search:");
        Book book2 = library.binarySearchByTitle("To Kill a Mockingbird");
        System.out.println(book2 != null ? book2 : "Book not found");
    }
}
```

Linear search has a time complexity of O(n), requiring a check of each element in the worst case, and is best for small or unsorted datasets. Binary search, with a time complexity of O(log n), is faster for large, sorted datasets by repeatedly halving the search interval but requires pre-sorted data.

Exercise 7: Financial Forecasting

Recursion is a technique where a function calls itself to solve smaller instances of a problem, breaking it down into manageable sub-problems. It typically includes a base case (termination condition) and a recursive case (the function calling itself).



CODE:

```
public class FinancialForecasting {
```

```
    /**
     * Calculates future value using recursion.
     *
     * @param presentValue The current value
     * @param growthRate The growth rate (e.g., 0.05 for 5%)
     * @param periods The number of periods to forecast
     * @return The future value
     */
    public static double calculateFutureValue(double presentValue, double growthRate, int periods) {
        // Base case
        if (periods == 0) {
            return presentValue;
        }
        // Recursive case
        return calculateFutureValue(presentValue * (1 + growthRate), growthRate, periods - 1);
    }

    public static void main(String[] args) {
        double presentValue = 1000;
        double growthRate = 0.05;
        int periods = 10;

        double futureValue = calculateFutureValue(presentValue, growthRate, periods);
        System.out.println("Future Value after " + periods + " periods: " + futureValue);
    }
}
```

Recursive algorithms for financial forecasting typically have a time complexity of O(n), with each period requiring a single recursive call until reaching the base case. To optimize and avoid excessive computation, memoization can be used to store and reuse results of sub-problems, reducing overall computation time. The trade-off is increased space complexity due to storing intermediate results, but it enhances time efficiency.