

Table of Contents

1. [Question 1 2](#)
 1. [Part A: 2](#)
 2. [Part B: 2](#)
 3. [Part C: 2](#)
 4. [Part D: 2](#)
2. [Question 3 3](#)
3. [Question 4 9](#)
4. [Question 5 11](#)
5. [Question 6 13](#)

Question 1

Part A:

The error in the code is the usage of **‘this’ inside a static method**. In Java, this refers to the current instance of the class, but static methods do not belong to any instance. Therefore, using this inside a static method is not allowed and will cause a compilation error.

Part B:

To determine which of the provided class definitions fits the definition of an immutable class, let's first review the properties of an immutable class in Java. An immutable class:

- Does not provide "setter" methods (methods that modify fields or objects referred to by fields).
- Ensures that fields are private and final.
- Prevents subclassing by declaring the class as final.
- Initializes all fields via a constructor performing a deep copy if necessary.
- Ensures that no methods return mutable objects.

None of the provided class definitions strictly follow all rules for creating an immutable class.

Part C:

- **public void** addABookinStore(**String bookName**, **int bookPrice**) {}
- **public boolean** isFullyOccupied() {}
- **public boolean** validateData() {}
- **public void** sellABook(**String bookName**) {}
- **public int** findCostofTotalBooksInShop() {}

‘compareTwoBookStores’ method can be written in three ways

- **public int** compareTwoBookStores(**BookStore otherBookStore**) {}
- **public static boolean** compareTwoBookStores(**BookStore bs1**, **BookStore bs2**) {}
- **public static int** compareTwoBookStores(**BookStore store1**, **BookStore store2**) {}

Part D:

```
7 10
15 30
4 20
15 30
=====
22 40
15 30
```

Question 3

```
import java.util.ArrayList;
```

```
// Item class
```

```
public abstract class Item {
    protected String name;
    protected double originalPrice;
    public Item(String name, double originalPrice) {
        this.name = name;
        this.originalPrice = originalPrice;
    }
    // Abstract method to be implemented by subclasses
    public abstract double calculatePrice();
    // Getters
    public String getName() {
        return name;
    }
    public double getOriginalPrice() {
        return originalPrice;
    }
}
```

```
// Perishable class
```

```
public class Perishable extends Item {
    private Date expiryDate;
    private int quantity;
    public Perishable(String name, double originalPrice, Date expiryDate, int quantity) {
        super(name, originalPrice);
    }
}
```

```

this.expiryDate = expiryDate;
this.quantity = quantity;
}
@Override
public double calculatePrice() {
    if (expiryDate.after(new Date())) {
        return originalPrice * quantity;
    } else {
        return 0;
    }
}
}

```

// Permanent class

```

public class Permanent extends Item {
    private Date entryDate;

    public Permanent(String name, double originalPrice, Date entryDate) {
        super(name, originalPrice);
        this.entryDate = entryDate;
    }

    public double calculatePrice() {
        // Calculate days since entryDate
        long daysSinceEntry = daysBetween(entryDate, new Date());

        // Initialize depreciated price to original price
        double depreciatedPrice = originalPrice;

        // Apply daily depreciation rate for each day since entryDate
        double depreciated = price;

        for (int i = 0; i < daysSinceEntry; i++) {
            depreciated = depreciated - (depreciated * 0.002);
        }

        return depreciated;
    }
}

```

// Helper method to calculate days between two dates

```

private long daysBetween(Date startDate, Date endDate) {

```

```
long millisecondsPerDay = 1000 * 60 * 60 * 24; // Number of milliseconds in a day
long difference = endDate.getTime() - startDate.getTime();
return difference / millisecondsPerDay;
}
```

// Bulk class

```
public class Bulk extends Permanent {
    private String description;
    private boolean inPackage;

    public Bulk(String name, double originalPrice, Date entryDate, String description, boolean inPackage) {
        super(name, originalPrice, entryDate);
        this.description = description;
        this.inPackage = inPackage;
    }

    @Override
    public double calculatePrice() {
        if (inPackage)
            return originalPrice; // Price remains the same if inPackage
        else
            return super.calculatePrice(); // Depreciate if not inPackage (same as Permanent items)
    }
}
```

// Pharmacy class

```
public class Pharmacy {
    private ArrayList<Item> items;

    public Pharmacy() {
        items = new ArrayList<>();
    }

    public void addItem(Item item) {
        items.add(item);
    }

    public double calculateTotalCost() {
        double totalCost = 0;
        for (Item item : items) {
```

```

totalCost += item.calculatePrice();
}
return totalCost;
}
public static void main(String[] args) {
// Example usage
Pharmacy pharmacy = new Pharmacy();
// Adding items
pharmacy.addItem(new Perishable("Medicine A", 100.0, new Date(2024 - 1900, 6, 1), 10));
pharmacy.addItem(new Permanent("Medicine B", 200.0, new Date(2023 - 1900, 0, 1)));
pharmacy.addItem(new Bulk("Medicine C", 300.0, new Date(2022 - 1900, 11, 1), "Bulk medicine",
true));
// Calculating total cost
double totalCost = pharmacy.calculateTotalCost();
System.out.println("Total cost of items in the pharmacy: $" + totalCost);
}
}

```

Question 4

```

public interface DocumentFormatter {
String format(String text);
}

public class PlainTextFormatter implements DocumentFormatter {
@Override
public String format(String text) {
return "PlainText: " + text;
}
}

public class HtmlFormatter implements DocumentFormatter {
@Override
public String format(String text) {
return "<html><body>" + text + "</body></html>";
}
}

```

```

}

public class Document {
    private DocumentFormatter formatter;
    private String content;
    public Document(DocumentFormatter formatter, String content) {
        this.formatter = formatter;
        this.content = content;
    }
    public String getFormattedContent() {
        return formatter.format(content);
    }
}

public class Main {
    public static void main(String[] args) {
        // Create formatters
        DocumentFormatter plainTextFormatter = new PlainTextFormatter();
        DocumentFormatter htmlFormatter = new HtmlFormatter();
        // Create documents with different formatters
        Document plainTextDocument = new Document(plainTextFormatter, "This is a plain text document.");
        Document htmlDocument = new Document(htmlFormatter, "This is an HTML document.");
        // Display formatted content
        System.out.println(plainTextDocument.getFormattedContent());
        System.out.println(htmlDocument.getFormattedContent());
    }
}

```

Question 5

```

public class OrderedPair<T> {
    private T first;
    private T second;
    public OrderedPair(T first, T second) {
        this.first = first;
        this.second = second;
    }
}

```

```

}
public T getFirst() {
return first;
}
public void setFirst(T first) {
this.first = first;
}
public T getSecond() {
return second;
}
public void setSecond(T second) {
this.second = second;
}
@Override
// can be written in formatting
public String toString() {
return "(" + first + ", " + second + ")";
}
}
public class Runner {
public static void main(String[] args) {
// Example with Strings (key-value pair)
OrderedPair<String> pair1 = new OrderedPair<>("Name", "Alice");
System.out.println("String pair: " + pair1);
// Example with Integers
OrderedPair<Integer> pair2 = new OrderedPair<>(10, 20);
System.out.println("Integer pair: " + pair2);
// Using getters
System.out.println("First of pair1: " + pair1.getFirst());
System.out.println("Second of pair2: " + pair2.getSecond());

}
}

```

Question 6

```
public void searchRecentCars() {
    ShowRoom sr = readFromFile();

    Car[] recentCars = sr.getCarsArray();

    int currYear = Calender.getInstance().get(Calendar.YEAR);

    for (Car car : carsArray) {
        if (currYear - car.getDateOfManufacture().getYear() <= 5){
            System.out.println(car.toString());
        }
    }

    return recentCars;
}

// Method to delete all black cars
public void deleteBlackCars() {
    ShowRoom sr = readFromFile();

    Car[] cars = sr.getCarsArray();

    for (int i = 0; i < cars.length; i++) {
        if (cars[i].getColor().equals("Black")){
            cars[i] = null;
        }
    }

    sr.setCarsArray(cars);
    writeToFile(sr);
}

// Method to read ShowRoom object from file
public ShowRoom readFromFile() {
    ShowRoom showRoom = null;

    try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("ShowRoom.ser"))) {
        showRoom = (ShowRoom) in.readObject();
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }

    return showRoom;
}
```

```
}  
  
// Method to write ShowRoom object to file  
public void writeToFile>ShowRoom sr) {  
    try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("ShowRoom.ser"))) {  
        out.writeObject(sr);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```