Task: Inventory Management System.

Milestone 1:Implement Inventory Item Hierarchy

Create a base `Item` class and specialized subclasses for an inventory management system.

Step 1: Create Item Class

- Define `Item` class with fields: id, name, price, quantity
- Add constructor and getter/setter methods
- Implement `Comparable<Item>` interface
- Override `compareTo` method to sort by price in ascending order

Step 2: Create Subclasses

Create three classes that extend `Item`:

Book Class:

- Add `author` field

Clothing Class:

- Add `size` field

Electronics Class:

- Add `warranty` field (integer)

Solution

`Item.java`:

```
package InventoryManagementSystem;

public class Item implements Comparable<Item>{
    private String id;
    private String name;
    private double price;
    private int quantity;
```

```
    public Item(String id, String name, double price, int
quantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

//    to be compared according to asc order of price
```

```
    @Override
    public int compareTo(Item o) {
//         obj which is calling is smaller  : return -ve
        if(this.price < o.price){
            return -1;
        } else if(this.price > o.price){
            return 1;
        }
        return 0;
    }
}
```

Book.java

```
package InventoryManagementSystem;

public class Book extends Item {
    private String author;

    public Book(String id , String name, double price , int
quantity, String author) {
        super(id, name, price, quantity);
        this.author = author;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

Clothing.java

```
package InventoryManagementSystem;

public class Clothing extends Item{
    private String size;
```

```
    public Clothing(String id, String name, double price , int
quantity ,  String size){
        super(id, name, price, quantity);
        this.size = size;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String size) {
        this.size = size;
    }
}
```

Electronics.java

```
package InventoryManagementSystem;

public class Electronics extends Item{
    private int warranty;

    public Electronics(String id , String name, double price,
int quantity, int warranty) {
        super(id, name, price, quantity);
        this.warranty = warranty;
    }

    public int getWarranty() {
        return warranty;
    }

    public void setWarranty(int warranty) {
        this.warranty = warranty;
    }
}
```

Milestone 2: Implement Generic Inventory Class

Create a generic `Inventory` class that can manage different types of items (Books, Clothing, Electronics) with validation and exception handling.

Step 1: Design Generic Class

- Create `Inventory<T extends Item>` class using bounded generics
- Use `HashMap<String, T>` to store items with ID as key
- Initialize HashMap in constructor

Step 2: Implement Add Method

- Create `add(T item)` method
- Check if quantity is negative and throw `InvalidQuantityException` exception with appropriate message
- Check for duplicate IDs using `containsKey()` and throw `DuplicateItemException`
- Add item to HashMap if validations pass

Step 3: Implement Utility Methods

Remove Method:

- Create `remove(T item)` method
- Remove item from HashMap using item ID

Get Methods:

- `get(String id)` - return single item by ID
- `getAll()` - return `List<T>` containing all items using `new ArrayList<>(items.values())`

The generic approach allows one `Inventory` class to work with `Book`, `Clothing`, and `Electronics` objects while maintaining type safety.

NOTE : Throw RuntimeException if custom exception creation is not known yet

Solution

Inventory.java

```
package InventoryManagementSystem;

import java.util.*;

public class Inventory<T extends Item> {
    private HashMap<String, T> items;

    public Inventory() {
        items = new HashMap<>();
    }

    public void add(T item) throws InvalidQuantityException {
        if(item.getQuantity() < 0){
            throw new InvalidQuantityException("Quantity is less
than equal to zero which is invalid");
        }
        if(items.containsKey(item.getId())) {
//            System.out.println("Item " + item.getId() + "
already exists");
            throw new DuplicateItemException("Item " +
item.getId() + " already exists");
        }
        items.put(item.getId(), item);
    }

    public void remove(T item) {
        items.remove(item.getId());
    }

    public T get(String id) {
        return items.get(id);
    }

    public List<T> getAll() {
        return new ArrayList<>(items.values());
    }
```

Milestone 3: Implement Recently Viewed Items Tracker

Create a system to track the most recent items viewed with a fixed capacity, handling re-visits efficiently.

Step 1: Choose Data Structure

- Use `LinkedList<Item>` instead of Queue for flexibility
- LinkedList allows efficient removal from middle, addition at front, and removal from end
- Queue alone cannot handle re-visiting existing items properly

Step 2: Initialize Class

- Create `RecentlyViewItems` class with `LinkedList<Item>` field
- Set `MAX_SIZE` constant (e.g., 3 for testing)
- Initialize empty LinkedList in constructor

Step 3: Implement Add Method

Create `addRecentlyViewedItem(Item item)` method:

- First remove the item if already present using `items.remove(item)`
- Add item to front using `addFirst(item)`
- Check if size exceeds MAX_SIZE and remove last item using `removeLast()`

Step 4: Implement Get Method

- Create `getRecentlyViewedItems()` method
- Return new `ArrayList<>(items)` to provide safe copy of current list

Why LinkedList?

- Handles duplicate views: removes existing item and moves it to front
- Efficient operations: O(1) for add/remove at ends, O(n) for middle removal
- Maintains chronological order with most recent first
- Simple capacity management by removing from tail

RecentlyViewedItems.java

```
package InventoryManagementSystem;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class RecentlyViewItems {
    private LinkedList<Item> items;
    private Integer MAX_SIZE = 3;

    public RecentlyViewItems() {
        items = new LinkedList<>();
    }

    public void addRecentlyViewedItem(Item item) {
        // remove the item if it is already present so that it
can be moved at the front
        items.remove(item);

        items.addFirst(item);
        if(items.size() > MAX_SIZE) {
            items.removeLast();
        }
    }

    public List<Item> getRecentlyViewedItems() {
        return new ArrayList<>(items);
    }
}
```

Milestone 4: Implement Priority-Based Order Processing

Create an order processing system that handles Express and Regular orders with priority-based processing.

Step 1: Design Order Class

- Create `Order` class with fields: `orderId` (String) and `isExpress` (boolean)
- Add constructor and getter/setter methods
- Implement `Comparable<Order>` interface for priority sorting

Step 2: Implement Priority Logic

Override `compareTo(Order o)` method with following priority rules:

- If both orders have same type (both express or both regular), compare by `orderId` alphabetically
- Else express order has higher priority

Step 3: Create OrderProcessor Class

- Use `PriorityQueue<Order>` instead of regular Queue for automatic priority sorting
- Initialize PriorityQueue in constructor

Step 4: Implement Processing Methods

Add Order:

- Create `addOrder(Order order)`
- Create `processOrder()`
- Add `getSize()` method to check remaining orders

`Solution:`

`Order.java`

```
package InventoryManagementSystem;

import Generics.Pair;
```

```
public class Order implements Comparable<Order>{
    private String orderId;
    private boolean isExpress;

    public Order(String orderId, boolean isExpress) {
        this.orderId = orderId;
        this.isExpress = isExpress;
    }

    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public boolean isExpress() {
        return isExpress;
    }

    public void setExpress(boolean express) {
        isExpress = express;
    }

    @Override
    public int compareTo(Order o) {
//
        if(this.isExpress == o.isExpress){
//            we need to decide on the basis of orderId
            return this.orderId.compareTo(o.orderId);
        } else if(this.isExpress){
            return -1;
        } else {
            return 1;
        }
    }
}
```

OrderProcessor.java

```
package InventoryManagementSystem;

import java.util.PriorityQueue;
import java.util.Queue;

public class OrderProcessor {
//    we need to process the order according to some priority
    private Queue<Order> orders;

    public OrderProcessor() {
//        orders = new LinkedList<>();
        orders = new PriorityQueue<>();
    }

    public void addOrder(Order order) {
        orders.offer(order);
//        supports add , offer
    }

    public Order processOrder() {
        return orders.poll();
    }

    public int getSize(){
        return orders.size();
    }
}
```

Milestone 5 : Implement Custom Comparators and Inventory Filtering

Create custom sorting comparators and add filtering/sorting functionality to the Inventory class.

Step 1: Create Name Comparator

- Create `ItemNameComparator` class implementing `Comparator<Item>`
- Compare names lexicographically

Step 2: Create Quantity Comparator

- Create `ItemQuantityComparator` class implementing `Comparator<Item>`
- Sort by quantity in descending order (highest quantity first)

Step 3: Add Filtering Methods to Inventory

Price Filter:

- Create `filterByPrice(double minPrice, double maxPrice)` method

Availability Filter:

- Create `filterByAvailability()` method

Step 4: Add Sorting Method

- Create `sortItems(Comparator<T> comparator)` method