Collection Exercise

Here's a simple example of using a TreeMap in Java to simulate a shopping cart. The TreeMap will store items as keys and their quantities as values, sorted by item names in alphabetical order.

import java.util.\*;

public class ShoppingCart {

public static void main(String[] args) {

// Create a TreeMap to store items and quantities

TreeMap<String, Integer> cart = new TreeMap<>();

// Adding items to the cart

addItemToCart(cart, "Apple", 3);

addItemToCart(cart, "Banana", 2);

addItemToCart(cart, "Orange", 5);

addItemToCart(cart, "Mango", 1);

// Display the cart

displayCart(cart);

// Remove an item from the cart

removeItemFromCart(cart, "Banana");

// Display the cart after removal

System.out.println("\nAfter removing Banana:");

displayCart(cart);

// Update the quantity of an item

updateItemQuantity(cart, "Apple", 5);

// Display the cart after updating quantity

System.out.println("\nAfter updating Apple quantity:");

displayCart(cart);

}

// Method to add items to the cart

private static void addItemToCart(TreeMap<String, Integer> cart, String item, int quantity) {

if (cart.containsKey(item)) {

cart.put(item, cart.get(item) + quantity); // Increase quantity if item exists

} else {

cart.put(item, quantity); // Add new item with its quantity

}

}

// Method to remove an item from the cart

private static void removeItemFromCart(TreeMap<String, Integer> cart, String item) {

cart.remove(item);

}

// Method to update the quantity of an existing item

private static void updateItemQuantity(TreeMap<String, Integer> cart, String item, int newQuantity) {

if (cart.containsKey(item)) {

cart.put(item, newQuantity);

} else {

System.out.println("Item not found in cart.");

}

}

// Method to display the cart

private static void displayCart(TreeMap<String, Integer> cart) {

System.out.println("Shopping Cart:");

for (Map.Entry<String, Integer> entry : cart.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

**Explanation:**

* **TreeMap** is used to store items as keys (e.g., "Apple", "Banana", etc.) and their quantities as values (e.g., 3 for 3 apples).
* The map is sorted alphabetically by default (based on item names).
* Methods like addItemToCart, removeItemFromCart, and updateItemQuantity are used to manipulate the cart.
* When displaying the cart, it automatically lists items in alphabetical order due to the behavior of TreeMap.

**Sample Output:**

Shopping Cart:

Apple: 3

Banana: 2

Mango: 1

Orange: 5

After removing Banana:

Shopping Cart:

Apple: 3

Mango: 1

Orange: 5

After updating Apple quantity:

Shopping Cart:

Apple: 5

Mango: 1

Orange: 5

In this example, we:

* Added items with quantities to the cart.
* Removed an item from the cart.
* Updated the quantity of an existing item.

2) simple banking system that uses a HashMap and an ArrayList to perform CRUD (Create, Read, Update, Delete) operations. In this example, the HashMap stores account IDs as keys and ArrayList stores bank account details like name, balance, etc. For simplicity, let's assume each account has a name and balance.

**Bank System with HashMap and ArrayList**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

class BankAccount {

String accountHolderName;

double balance;

// Constructor

public BankAccount(String accountHolderName, double balance) {

this.accountHolderName = accountHolderName;

this.balance = balance;

}

// Getters and Setters

public String getAccountHolderName() {

return accountHolderName;

}

public void setAccountHolderName(String accountHolderName) {

this.accountHolderName = accountHolderName;

}

public double getBalance() {

return balance;

}

public void setBalance(double balance) {

this.balance = balance;

}

@Override

public String toString() {

return "AccountHolder: " + accountHolderName + ", Balance: " + balance;

}

}

public class BankSystem {

private static Map<Integer, ArrayList<BankAccount>> bankAccounts = new HashMap<>();

public static void main(String[] args) {

// Create accounts

createAccount(1, "John Doe", 1000.00);

createAccount(2, "Jane Smith", 2000.00);

createAccount(3, "Tom Brown", 1500.00);

// Read accounts

System.out.println("Account 1 Details: " + readAccount(1));

System.out.println("Account 2 Details: " + readAccount(2));

// Update an account balance

updateAccountBalance(1, 1200.00);

System.out.println("Updated Account 1 Details: " + readAccount(1));

// Delete an account

deleteAccount(3);

System.out.println("Account 3 after deletion: " + readAccount(3));

}

// Create account - Add an account to the HashMap

public static void createAccount(int accountId, String accountHolderName, double initialBalance) {

ArrayList<BankAccount> accounts = bankAccounts.getOrDefault(accountId, new ArrayList<>());

accounts.add(new BankAccount(accountHolderName, initialBalance));

bankAccounts.put(accountId, accounts);

System.out.println("Account created for: " + accountHolderName);

}

// Read account - Get details of an account

public static String readAccount(int accountId) {

ArrayList<BankAccount> accounts = bankAccounts.get(accountId);

if (accounts == null || accounts.isEmpty()) {

return "Account not found.";

}

return accounts.get(0).toString(); // Assuming one account per ID

}

// Update account balance

public static void updateAccountBalance(int accountId, double newBalance) {

ArrayList<BankAccount> accounts = bankAccounts.get(accountId);

if (accounts != null && !accounts.isEmpty()) {

accounts.get(0).setBalance(newBalance);

System.out.println("Account " + accountId + " balance updated to " + newBalance);

} else {

System.out.println("Account not found.");

}

}

// Delete account - Remove an account from the HashMap

public static void deleteAccount(int accountId) {

if (bankAccounts.containsKey(accountId)) {

bankAccounts.remove(accountId);

System.out.println("Account " + accountId + " deleted.");

} else {

System.out.println("Account not found.");

}

}

}

**Explanation:**

1. **BankAccount Class**:
   * This class represents a bank account with fields for the account holder's name and balance.
   * It includes a constructor and getter/setter methods, as well as a toString() method to print the account details.
2. **BankSystem Class**:
   * The bankAccounts map uses account IDs as keys, and the ArrayList stores the account information (in this case, just one account per ID).
   * **CRUD Operations**:
     + **Create**: The createAccount() method creates a new bank account and adds it to the map.
     + **Read**: The readAccount() method retrieves the details of a bank account by ID.
     + **Update**: The updateAccountBalance() method updates the balance of a specific account.
     + **Delete**: The deleteAccount() method removes an account by ID.

**Example Output:**

Account created for: John Doe

Account created for: Jane Smith

Account created for: Tom Brown

Account 1 Details: AccountHolder: John Doe, Balance: 1000.0

Account 2 Details: AccountHolder: Jane Smith, Balance: 2000.0

Updated Account 1 Details: AccountHolder: John Doe, Balance: 1200.0

Account 3 after deletion: Account not found.

This example demonstrates a basic implementation using HashMap and ArrayList to manage bank account details and supports basic CRUD operations.

**Java Collection Framework Student Exercise Handbook**

**1. ArrayList**

An ArrayList is a resizable array implementation of the List interface. It allows for random access to elements but is not synchronized.

**Key Features:**

* Elements are stored in an array and resized as needed.
* Allows duplicate elements.
* Maintains insertion order.

**Exercise 1: Basic Operations with ArrayList**

* **Task**: Create an ArrayList of String to store student names.
  + Add names to the list.
  + Retrieve and print the name at a specific index.
  + Remove a name by index.
  + Display the size of the list.

import java.util.ArrayList;

public class ArrayListExample {

public static void main(String[] args) {

ArrayList<String> students = new ArrayList<>();

// Adding elements

students.add("John");

students.add("Sarah");

students.add("David");

// Retrieving an element

System.out.println("Student at index 1: " + students.get(1));

// Removing an element

students.remove(0); // Remove "John"

// Size of the ArrayList

System.out.println("Size of list: " + students.size());

// Displaying all elements

System.out.println("Students: " + students);

}

}

**Exercise 2: Sorting an ArrayList**

* **Task**: Sort the ArrayList of student names in alphabetical order and print the list.

import java.util.ArrayList;

import java.util.Collections;

public class SortArrayList {

public static void main(String[] args) {

ArrayList<String> students = new ArrayList<>();

students.add("John");

students.add("Sarah");

students.add("David");

// Sorting the list

Collections.sort(students);

// Displaying sorted list

System.out.println("Sorted Students: " + students);

}

}

**2. LinkedList**

A LinkedList is a doubly-linked list implementation of the List and Deque interfaces. It provides more efficient insertions and deletions compared to ArrayList.

**Key Features:**

* Supports fast insertions and deletions at both ends.
* Can be used as a queue or stack.

**Exercise 1: Basic Operations with LinkedList**

* **Task**: Create a LinkedList of integers and perform the following operations:
  + Add elements to the beginning and end.
  + Remove elements from the beginning and end.
  + Display the elements.

import java.util.LinkedList;

public class LinkedListExample {

public static void main(String[] args) {

LinkedList<Integer> numbers = new LinkedList<>();

// Adding elements

numbers.add(10); // Add to end

numbers.addFirst(5); // Add to beginning

numbers.addLast(15); // Add to end

// Removing elements

numbers.removeFirst(); // Remove from beginning

numbers.removeLast(); // Remove from end

// Displaying elements

System.out.println("LinkedList: " + numbers);

}

}

**Exercise 2: Implementing a Stack using LinkedList**

* **Task**: Use LinkedList to implement a basic stack that supports push, pop, and peek operations.

import java.util.LinkedList;

public class StackUsingLinkedList {

private LinkedList<Integer> stack = new LinkedList<>();

public void push(int value) {

stack.addFirst(value);

}

public int pop() {

return stack.removeFirst();

}

public int peek() {

return stack.getFirst();

}

public boolean isEmpty() {

return stack.isEmpty();

}

public static void main(String[] args) {

StackUsingLinkedList stack = new StackUsingLinkedList();

stack.push(10);

stack.push(20);

stack.push(30);

System.out.println("Top element: " + stack.peek()); // Output: 30

System.out.println("Popped element: " + stack.pop()); // Output: 30

System.out.println("Is stack empty? " + stack.isEmpty());

}

}

**3. TreeMap**

A TreeMap is a NavigableMap implementation that uses a Red-Black tree for sorting the keys. The keys are ordered in their natural order, or by a comparator provided at map creation.

**Key Features:**

* Sorted by keys.
* No duplicate keys.
* Navigable operations like lower(), ceiling().

**Exercise 1: Basic Operations with TreeMap**

* **Task**: Create a TreeMap to store student names and their grades.
  + Add students and their grades.
  + Retrieve the grade of a specific student.
  + Display all students sorted by name.

import java.util.TreeMap;

public class TreeMapExample {

public static void main(String[] args) {

TreeMap<String, Integer> studentGrades = new TreeMap<>();

// Adding elements (student name as key, grade as value)

studentGrades.put("John", 85);

studentGrades.put("Sarah", 92);

studentGrades.put("David", 78);

// Retrieving an element

System.out.println("Grade of Sarah: " + studentGrades.get("Sarah"));

// Displaying all elements (sorted by key)

System.out.println("All students and their grades: " + studentGrades);

}

}

**Exercise 2: Using NavigableMap Methods**

* **Task**: Use TreeMap to find the closest student in terms of grade using higherEntry() and lowerEntry().

import java.util.TreeMap;

public class TreeMapNavigableExample {

public static void main(String[] args) {

TreeMap<String, Integer> studentGrades = new TreeMap<>();

studentGrades.put("John", 85);

studentGrades.put("Sarah", 92);

studentGrades.put("David", 78);

// Find closest higher entry

System.out.println("Student with higher grade than 80: " + studentGrades.higherEntry("John"));

// Find closest lower entry

System.out.println("Student with lower grade than 80: " + studentGrades.lowerEntry("Sarah"));

}

}

**4. HashSet**

A HashSet is an implementation of the Set interface that does not allow duplicates and does not guarantee any specific order of the elements.

**Key Features:**

* No duplicate elements.
* Unordered collection.
* Fast operations for adding, removing, and checking for the existence of elements.

**Exercise 1: Basic Operations with HashSet**

* **Task**: Create a HashSet to store unique student IDs.
  + Add student IDs.
  + Try to add duplicate IDs.
  + Display the set and check if a specific ID exists.

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

HashSet<Integer> studentIDs = new HashSet<>();

// Adding elements

studentIDs.add(101);

studentIDs.add(102);

studentIDs.add(103);

// Attempting to add duplicate

studentIDs.add(101); // Duplicate element, won't be added

// Checking for an element

System.out.println("Does student with ID 102 exist? " + studentIDs.contains(102));

// Displaying elements

System.out.println("Student IDs: " + studentIDs);

}

}

**Exercise 2: Set Operations**

* **Task**: Use HashSet to find the union, intersection, and difference of two sets of student IDs.

import java.util.HashSet;

public class HashSetOperations {

public static void main(String[] args) {

HashSet<Integer> set1 = new HashSet<>();

HashSet<Integer> set2 = new HashSet<>();

set1.add(101);

set1.add(102);

set1.add(103);

set2.add(102);

set2.add(104);

set2.add(105);

// Union

HashSet<Integer> union = new HashSet<>(set1);

union.addAll(set2);

System.out.println("Union: " + union);

// Intersection

HashSet<Integer> intersection = new HashSet<>(set1);

intersection.retainAll(set2);

System.out.println("Intersection: " + intersection);

// Difference

HashSet<Integer> difference = new HashSet<>(set1);

difference.removeAll(set2);

System.out.println("Difference: " + difference);

}

}