Map - Ques on 1: Movie Rental Tracker with Rental Limits Using HashMap

You are building a system to track movies rented from a video rental store. The system should store the tles of movies along with the number of mes they have been rented. Users should be able to add, update, or remove movie records, check if a movie exists, and retrieve specific rental counts. Addi onally, the system should enforce a rental limit per movie, such that no movie can be rented more than a certain number of mes. If a movie exceeds this limit, it should be flagged as "over-rented."

If any opera on a empts to access a movie that does not exist in the system, a custom excep on, MovieNotFoundExcep on, should be thrown and handled appropriately.

Opera ons to be Performed:

1. Add Movie Records: Input the number of movies and their respec ve tles, rental counts, and rental limits.
2. Retrieve Rental Count: Look up the total number of mes a specific movie has been rented by its tle.
3. Remove Movie Records: Remove a movie's record from the system.
4. Check if a Movie Exists: Verify if a specific movie is in the system based on its tle.
5. Calculate Total Rental Count: Display the total number of rentals for all movies.
6. Check for Over-Rented Movies: Display all movies that have been rented beyond their rental limit.

Input Format

Adding Movies:

* + - The first line contains an integer n, represen ng the number of movies to add.
    - For each movie, input:
      * The first line contains the movie’s tle (String). o The second line contains the rental count (Integer).
      * The third line contains the rental limit (Integer). Perform Opera ons: • A line containing the tle of a movie to retrieve its rental count.
  + A line containing the tle of a movie to remove from the system.
  + A line containing the tle of a movie to check if it exists in the system.
  + A line to check for movies exceeding the rental limit.

Output Format

Retrieving Rental Count for Movie:

* + If found, print: [Movie tle] has been rented [count] mes.
  + If not found, throw and catch the MovieNotFoundExcep on and print: Movie [Movie tle] not found.

Removing a Movie:

* + If found and removed, print: Movie [Movie tle] has been removed from the store.
  + If not found, throw and catch the MovieNotFoundExcep on and print: Movie [Movie tle] not found for removal.

Checking if a Movie Exists:

* + If the movie exists, print: Movie [Movie tle] exists in the store.
  + If the movie doesn’t exist, throw and catch the MovieNotFoundExcep on and print: Movie [Movie tle] does not exist in the store.

Calcula ng Total Rental Count:

* + Print the total rental count of all movies: Total Movies Rented: [total count]

Checking for Over-Rented Movies:

* + If any movie has been rented beyond its limit, print: Movie [Movie tle] has been over-rented.

Sample Input 1

3

Incep on

100

150

The Dark Knight

120

200

Interstellar

80

100

Incep on

The Dark Knight

Interstellar

Check Over-Rented

Sample Output 1

Incep on has been rented 100 mes.

Movie The Dark Knight has been removed from the store.

Movie Interstellar exists in the store.

Movie Incep on has been over-rented.

Total Movies Rented: 300

Sample Input 2

2

The Godfather

250

300

Pulp Fic on

300

350

The Shawshank Redemp on

The Matrix

Check Over-Rented

Sample Output 2

Movie The Godfather has been rented 250 mes.

Movie Pulp Fic on has been rented 300 mes.

Movie The Shawshank Redemp on not found.

Movie The Matrix does not exist in the store.

Movie The Godfather has been over-rented.

Total Movies Rented: 550

Map - Ques on 2: Library Book Borrowing Tracker with Enhanced Features

You are building a library system to track borrowed books. The system should store the tles of books along with the number of mes they have been borrowed. Users should be able to add, update, or remove book records, check if a book exists, retrieve specific borrow counts, and display all books that have been borrowed more than a certain number of mes.

Addi onally, the system should handle custom excep ons for invalid book access and allow users to retrieve the total borrow count of books that exceed a specified threshold.

Opera ons to be Performed:

1. Add Book Records: Input the number of books and their respec ve tles and borrow counts.
2. Retrieve Borrow Count: Look up the total number of mes a specific book has been borrowed by its tle.
3. Remove Book Records: Remove a book's record from the system.
4. Check if a Book Exists: Verify if a specific book is in the system based on its tle.
5. Retrieve Books Above Threshold: Display all books that have been borrowed more than a certain number of mes.
6. Calculate Total Borrow Count: Display the total number of borrow counts for all books.

Input Format:

Adding Books:

* + - The first line contains an integer n, represen ng the number of books to add.
    - For each book, input:
      * The first line contains the book’s tle (String).
      * The second line contains the borrow count (Integer).

Perform Opera ons:

* + A line containing the tle of a book to retrieve its borrow count.
  + A line containing the tle of a book to remove from the system.
  + A line containing the tle of a book to check if it exists in the system.
  + A line containing the threshold to check for books borrowed more than that number.

Output Format:

Retrieving Borrow Count:

* + If found, print: [Book tle] has been borrowed [count] mes.
  + If not found, throw and catch the BookNotFoundExcep on and print: Book [Book tle] not found.

Removing a Book:

* + If found and removed, print: Book [Book tle] has been removed from the library.
  + If not found, throw and catch the BookNotFoundExcep on and print: Book [Book tle] not found for removal.

Checking if a Book Exists:

* + If the book exists, print: Book [Book tle] exists in the library.
  + If the book doesn’t exist, throw and catch the BookNotFoundExcep on and print: Book [Book tle] does not exist in the library.

Retrieve Books Above Threshold:

* + Print all books that have been borrowed more than the given threshold.

Sample Input 1:

3

The Alchemist

150

1984

100

To Kill a Mockingbird

200

The Alchemist

1984

Check Borrow Threshold 120

Sample Output 1:

The Alchemist has been borrowed 150 mes.

Book 1984 has been removed from the library.

Book To Kill a Mockingbird exists in the library.

Books above the threshold of 120:

The Alchemist has been borrowed 150 mes.

To Kill a Mockingbird has been borrowed 200 mes.

Set - Example 1: Event Ticket Management System Using HashSet

You are building a system to manage event ckets for a concert. Each cket has a unique ID, a name (which represents the type of seat or VIP package), and a status to indicate whether the cket has been used or not. The system needs to:

1. Add new ckets to the inventory.
2. Mark ckets as used by their ID.
3. Remove ckets from the inventory by their ID.
4. Track how many ckets are s ll unused a er each opera on.

Task:

Implement an EventTicketSystem class using a HashSet to manage the list of ckets. Each cket should be represented by a Ticket class with the following a ributes:

id (int): A unique iden fier for the cket, star ng from 2001 and increasing sequen ally for each new cket added. name (String): The name of the cket (e.g., VIP, General Admission, etc.).

used (boolean): Status indica ng whether the cket has been used or not (default = false).

The EventTicketSystem class should include the following methods:

Func onali es

Add Ticket:

* Implement the method public void addTicket(String name) to add a new cket to the inventory list.
* The cket's ID should be assigned automa cally, star ng from 2001 and incremen ng for each new cket.
* This method should take the cket name as input and create a new Ticket object with a unique ID and the given name, then add it to the list of ckets.

Use Ticket:

* Implement the method public void useTicket(int id) to mark a cket as used by its ID.
* If the cket is found, mark it as used and print a message indica ng that the cket with the given ID has been used.
* If the cket is not found, print a message indica ng that the cket with the given ID is not found.

Remove Ticket:

* Implement the method public void removeTicket(int id) to remove a cket from the inventory by its ID.
* Print whether the cket was successfully removed or if it was not found.

Track Unused Tickets:

* Implement the method public int countUnusedTickets() to count and return the number of ckets that are s ll unused.
* This method will be used to display how many ckets are s ll unused a er performing the use and remove opera ons.

Excep on Handling:

* In case a cket is not found during the Use or Remove opera ons, a custom excep on TicketNotFoundExcep on should be thrown.
* This excep on should be caught in the methods where the opera on is performed and should print a message indica ng that the cket with the given ID could not be found.

Input Format

Adding Tickets:

* The first line contains an integer n, represen ng the number of ckets to add to the inventory.
* For each cket, input:
  + The name of the cket (String).

Perform Opera ons:

* A line containing the ID of the cket to mark as used.
* A line containing the ID of the cket to remove from the inventory.

Output Format

Current Inventory:

* Print "Tickets in the Inventory:"
* For each cket in the inventory, print the cket details in the format:
  + Ticket{id=<id>, name='<name>', used=<used>}

Ticket Use Status:

* If the cket is used, print: Ticket with ID <id> has been used.
* If the cket is not found, throw and catch the TicketNotFoundExcep on and print: Ticket with ID <id> not found.

Ticket Removal Status:

* If the cket is removed, print: Ticket with ID <id> removed successfully.
* If the cket is not found, throw and catch the TicketNotFoundExcep on and print: Ticket with ID <id> not found.

Updated Inventory:

* Print "Updated Inventory:"
* For each cket in the updated inventory, print the cket details in the format:
  + Ticket{id=<id>, name='<name>', used=<used>}

Total Unused Tickets:

* Print "Total unused ckets: <number>", where <number> is the total count of ckets that are s ll unused.

Sample Input 1:

3

VIP Front Row

General Admission

Backstage Pass

2002

2003

Sample Output 1:

Tickets in the Inventory:

Ticket{id=2001, name='VIP Front Row', used=false}

Ticket{id=2002, name='General Admission', used=false}

Ticket{id=2003, name='Backstage Pass', used=false}

Ticket with ID 2002 has been used.

Ticket with ID 2003 removed successfully.

Updated Inventory:

Ticket{id=2001, name='VIP Front Row', used=false}

Ticket{id=2002, name='General Admission', used=true}

Total unused ckets: 1

Sample Input 2:

4

Regular Ticket

VIP Package

Early Bird

Group Discount

2005

2004

Sample Output 2:

Tickets in the Inventory:

Ticket{id=2001, name='Regular Ticket', used=false}

Ticket{id=2002, name='VIP Package', used=false}

Ticket{id=2003, name='Early Bird', used=false} Ticket{id=2004, name='Group Discount', used=false}

Ticket with ID 2005 not found.

Ticket with ID 2004 not found.

Updated Inventory:

Ticket{id=2001, name='Regular Ticket', used=false}

Ticket{id=2002, name='VIP Package', used=false}

Ticket{id=2003, name='Early Bird', used=false} Ticket{id=2004, name='Group Discount', used=false}

Total unused ckets: 4

Set - Example 2: Event Registra on System Using HashSet

You are developing an Event Registra on System that helps organize registra ons for a community event. The system uses a HashSet to manage par cipant IDs. Each par cipant ID is unique and is stored in the HashSet.

Your task is to write a Java program that performs the following opera ons:

1. Add Par cipant IDs: Input a number of unique par cipant IDs and store them in the HashSet.
2. Check for a Par cipant ID: A er storing the IDs, check if a specific par cipant ID exists in the HashSet and report whether it is present or not.
3. Delete a Par cipant ID: If the ID exists, it should be removed from the HashSet. If the ID is not found, indicate that the ID is not present.
4. Display the Updated List of Par cipant IDs: A er a emp ng the dele on, display the updated list of par cipant IDs.
5. Count Even Par cipant IDs: Count how many par cipant IDs are even and display the count.

Input Format:

* The first line contains an integer n represen ng the number of par cipant IDs.
* The next n lines each contain a unique integer represen ng a par cipant ID.
* The following line contains an integer represen ng the par cipant ID to be checked.
* The final line contains an integer represen ng the par cipant ID to be removed.

Output Format:

* Display whether the par cipant ID to be checked is present in the HashSet in the format: o "The par cipant ID [ID] is present in the HashSet." if the ID is found.
  + "The par cipant ID [ID] is not present in the HashSet." if the ID is not found.
* Display whether the par cipant ID to be removed was found and removed or not in the format:
  + "The par cipant ID [ID] was removed from the HashSet." if the ID was found and removed.
  + "The par cipant ID [ID] was not found in the HashSet." if the ID was not found.
* Display the updated list of par cipant IDs in the HashSet in the format:
  + "Updated list of par cipant IDs:" o For each ID, print "Par cipant ID: [ID]" in the order they were iterated.
* Count Even Par cipant IDs:
  + Print: "Number of even par cipant IDs: [count]" where [count] is the number of even par cipant IDs.

Sample Input 1:

4

1011

2022

3033

4044

3033

2022

Sample Output 1:

The par cipant ID 3033 is present in the HashSet.

The par cipant ID 2022 was removed from the HashSet.

Updated list of par cipant IDs:

Par cipant ID: 1011

Par cipant ID: 3033

Par cipant ID: 4044

Number of even par cipant IDs: 1

Sample Input 2:

5

100

200

300

400

500

150

350

Sample Output 2:

The par cipant ID 150 is not present in the HashSet.

The par cipant ID 350 was not found in the HashSet.

Updated list of par cipant IDs:

Par cipant ID: 100

Par cipant ID: 200

Par cipant ID: 300

Par cipant ID: 400

Par cipant ID: 500

Number of even par cipant IDs: 5

List - Ques on 1: Help Desk Ticket Management System with Sor ng by Hours Spent (Comparable) and Problem Descrip on (Comparator)

You are developing a so ware applica on for a Help Desk to manage and track user support ckets. The applica on allows users to input mul ple help ckets and sorts them based on either the total me spent on each cket or the problem descrip on. Each cket consists of a Ticket ID, a problem descrip on, and the number of hours spent resolving the issue, which must be entered on separate lines.

The applica on should:

1. Input Valida on:
   * The Ticket ID is a posi ve integer between 1 and 10 and must be unique across all ckets.
   * The number of hours spent is an integer between 1 and 100.
2. Sor ng Requirements:
   * Sort the ckets based on:
     + By Default: The number of hours spent, in descending order, using the Comparable interface.
     + Alterna vely, allow sor ng the ckets by the problem descrip on in alphabe cal order, using the Comparator interface.
3. Average Hours Calcula on:
   * Calculate the average number of hours spent across all ckets.
4. Func onality:
   * Sort and display the ckets according to the selected criteria (by hours spent or by problem descrip on).
   * Display each cket’s details a er sor ng.
   * Display the average hours spent across all ckets.

Task:

1. Ticket Class:
   * Implement a Ticket class that represents a cket with the following a ributes:
     + cketID (int): A unique iden fier for the cket, between 1 and 10.
     + descrip on (String): A brief descrip on of the problem. o hoursSpent (int): The number of hours spent on the cket, between 1 and 100.
2. Comparable Interface:
   * Implement the Comparable<Ticket> interface in the Ticket class to sort ckets by the number of hours spent in descending order.
3. Comparator Interface:
   * Implement a ProblemDescrip onComparator class that implements the Comparator<Ticket> interface to compare ckets based on their problem descrip ons in alphabe cal order.
4. Main Applica on Flow:
   * Input all cket records in a single batch with each cket’s details on separate lines: o First Line: Ticket ID (unique integer between 1 and 10).
     + Second Line: Problem Descrip on (string).
     + Third Line: Number of Hours Spent (integer between 1 and 100). o Enter "stop" to signal that the input is complete.
   * A er input, display the sorted ckets based on the chosen criteria (either by hours spent or by problem descrip on).
   * Calculate and display the average hours spent across all ckets.

Input Format:

* + Input all cket records in a single batch, with each cket’s details on separate lines:
    - First Line: Ticket ID (unique integer between 1 and 10).
    - Second Line: Problem Descrip on (string). o Third Line: Number of Hours Spent (integer between 1 and 100).
    - Enter "stop" to signal that the input is complete.
  + A er inpu ng the cket details, choose the sor ng criteria by inpu ng:
    - 1 to sort ckets by the number of hours spent (default, using Comparable).
    - 2 to sort ckets by problem descrip on (alphabe cal, using Comparator).

Output Format:

* + Display each cket in the following format:
    - Ticket ID: [ cket ID] o Problem Descrip on: [problem descrip on] o Hours Spent: [hours spent]
  + At the end, display the average number of hours spent across all ckets in the following format:
    - Average Hours Spent: [average number of hours spent] (forma ed to two decimal places).

Error Handling:

* + NumberFormatExcep on:
    - Error Message for Non-Integer Input: "Please enter a valid integer."
  + Number of Hours Spent:
    - Error Message for Out-of-Range Values: "Hours spent must be between 1 and 100."  ID Valida on:
    - Error Message for Invalid ID: "Ticket ID must be a posi ve integer between 1 and 10 and must be unique."

Sample Input 1:

1

So ware Installa on

10

2

Password Reset

5

3

Network Issue

20

4

System Crash

18

5

Backup Restore 15 stop 1

Sample Output 1 (Sorted by Hours Spent):

# Ticket ID: 3

Problem Descrip on: Network Issue

# Hours Spent: 20

# Ticket ID: 4

Problem Descrip on: System Crash

Hours Spent: 18

Ticket ID: 5

Problem Descrip on: Backup Restore

Hours Spent: 15

Ticket ID: 1

Problem Descrip on: So ware Installa on

Hours Spent: 10

Ticket ID: 2

Problem Descrip on: Password Reset

Hours Spent: 5

Average Hours Spent: 13.60

Sample Input 2:

1

So ware Installa on

10

2

Password Reset

5

3

Network Issue

20

4

System Crash

18

5

Backup Restore 15 stop 2

Sample Output 2 (Sorted by Problem Descrip on):

Ticket ID: 5

Problem Descrip on: Backup Restore

Hours Spent: 15

Ticket ID: 4

Problem Descrip on: System Crash

Hours Spent: 18

Ticket ID: 3

Problem Descrip on: Network Issue

Hours Spent: 20

Ticket ID: 1

Problem Descrip on: So ware Installa on

Hours Spent: 10

Ticket ID: 2

Problem Descrip on: Password Reset

Hours Spent: 5

Average Hours Spent: 13.60

Sample Input 3 (Invalid Input):

one

Hard Drive Failure

12

2

Printer Error

7

3

System Down me

30

10

Data Corrup on

40

12

Connec on Error 25 stop

1

Sample Output 3:

Please enter a valid integer.

Ticket ID must be a posi ve integer between 1 and 10 and must be unique.

Sample Input 4 (Invalid Hour Value):

1

Account Setup

15

2

Email Setup

10

3

So ware Crash

30

5

Printer Jam

60

10

Data Loss Recovery 105 stop 1

Sample Output 4:

Hours spent must be between 1 and 100.

JDBC - Example 1: Employee Database Management System (CRUD Opera ons)

You are tasked with developing a Java applica on to manage employee data using JDBC and MySQL. The applica on should allow users to perform CRUD (Create, Read, Update, Delete) opera ons on employee data stored in a database. Addi onally, it should validate each opera on to ensure correct input and data integrity. Users will also be able to query and view the number of ac ve employees (those who have joined the company and are s ll employed).

Table Name: employee

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| employee\_id | int |  |  |  |
| name | varchar(100) |  |  |  |
| age | int |  |  |  |
| gender | varchar(10) |  |  | Gender (Male/Female/Other) |
| join\_date | date |  |  | Date when the employee joined |
| ac ve\_end\_date | date |  |  | Date un l the employee is employed |

Valida on for each opera on:

1. Inser ng Employee:
   * Validate that the employee name is not empty.
   * Age must be a posi ve integer, and within a valid range (18 to 100 years).
   * Gender must be one of the following: Male, Female, or Other.
2. Upda ng Employee:
   * Ensure the employee exists based on the employee\_id before upda ng.
   * Validate that the new age is a posi ve integer.
3. Dele ng Employee:
   * Ensure the employee exists in the database.
   * Delete only if the employee's `ac ve\_end\_date` is `null`.
4. Show All Employees:
   * Display all employees currently in the database, including their details in ascending order of name and descending order of join\_date.
5. Show Ac ve Employees:
   * Display all employees who are currently ac ve, i.e., those whose `join\_date <= current date <= ac ve\_end\_date`.

Opera ons:

1. Insert Employee Details:
   * Input: Employee ID, Name, Age, Gender, Join Date, Ac ve End Date.
   * Valida on: Checks as described above.

1. Update Employee Details:
   * Input: Employee ID, New Age.
   * Valida on: Ensure the employee exists and the new details are valid.

1. Delete Employee Details:
   * Input: Employee ID.
   * Valida on: Ensure the employee exists and is eligible for dele on (ac ve).

1. Show All Employee Details:
   * Output: Display all employees in the table.

1. Show Ac ve Employees:
   * Output: Display all employees whose employment is s ll valid (based on the current date).

Addi onal Feature:

1. Delete Employees Below Given Age:

* Input: Age.
* Valida on: Ensure the employee exists. If not, display: “No employees available below age [age].”

Ini al Query:

CREATE TABLE employee ( employee\_id INT PRIMARY KEY, name VARCHAR(100),

age INT, gender VARCHAR(10), join\_date DATE, ac ve\_end\_date DATE

);

INSERT INTO employee VALUES(1, 'John Smith', 30, 'Male', '2020-01-01', '2025-01-01');

INSERT INTO employee VALUES(2, 'Jane Doe', 25, 'Female', '2021-02-15', '2023-12-31');

INSERT INTO employee VALUES(3, 'Bob Brown', 40, 'Male', '2019-05-10', '2023-05-10');

INSERT INTO employee VALUES(4, 'Alice White', 50, 'Female', '2018-07-15', '2024-07-15');

INSERT INTO employee VALUES(5, 'Emily Davis', 35, 'Female', '2020-09-01', '2025-09-01');

2. Student Enrollment Management System (CRUD Opera ons)

JDBC - Example 2: Student Enrollment Management System (CRUD Opera ons)

You are tasked with crea ng a Java applica on to manage student enrollment data using JDBC and MySQL. The applica on should allow users to perform CRUD (Create, Read, Update, Delete) opera ons on student data stored in the database. The system should also validate each opera on to ensure correct input and data integrity. Addi onally, users will be able to query and view ac ve students (those who are currently enrolled).

Table Name: student

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type |  |  |
| student\_id | int |  | Primary key |
| name | varchar(100) |  |  |
| age | int |  |  |
| gender | varchar(10) |  | Gender (Male/Female/Other) |
| enrollment\_date | date |  | Date when the student enrolled |
| gradua on\_date | date |  | Date when the student is expected to graduate |

Valida on for each opera on:

1. Inser ng Student:
   * Validate that the student name is not empty.
   * Age must be a posi ve integer, and within a valid range (18 to 80 years).
   * Gender must be one of the following: Male, Female, or Other.
2. Upda ng Student:
   * Ensure the student exists based on the student\_id before upda ng.  Validate that the new age is a posi ve integer.
3. Dele ng Student:
   * Ensure the student exists in the database.
   * Delete only if the student is not graduated (i.e., `gradua on\_date` is `null`).
4. Show All Students:
   * Display all students currently in the database, including their details.
5. Show Ac ve Students:
   * Display all students who are currently ac ve, i.e., those whose `enrollment\_date <= current date <= gradua on\_date`

Opera os

1. Insert Student Details:
   * Input: Student ID, Name, Age, Gender, Enrollment Date, Gradua on Date.  Valida on: Checks as described above.

1. Update Student Details:
   * Input: Student ID, New Age.
   * Valida on: Ensure the student exists and the new details are valid.

1. Delete Student Details:
   * Input: Student ID.
   * Valida on: Ensure the student exists and is eligible for dele on (not graduated).

1. Show All Student Details:
   * Output: Display all students in the table.

1. Show Ac ve Students:
   * Output: Display all students whose enrollment is s ll ac ve (based on the current date).

Addi onal Feature:

1. Delete Students Below Given Age:

* Input: Age.
* Valida on: Ensure the student exists. If not, display: “No students available below age [age].”

Ini al Query:

CREATE TABLE student ( student\_id INT PRIMARY KEY, name VARCHAR(100),

age INT, gender VARCHAR(10), enrollment\_date DATE, gradua on\_date DATE

);

INSERT INTO student VALUES(1, 'Sarah Johnson', 21, 'Female', '2022-01-10', '2026-06-10');

INSERT INTO student VALUES(2, 'James Smith', 23, 'Male', '2021-09-01', '2025-06-30');

INSERT INTO student VALUES(3, 'Linda Green', 19, 'Female', '2023-05-05', NULL);

INSERT INTO student VALUES(4, 'David Brown', 22, 'Male', '2022-08-25', '2024-12-15');

INSERT INTO student VALUES(5, 'Olivia White', 20, 'Female', '2023-03-11', NULL);