Electric Car Charging Station Management System

Description:
You are tasked with implementing a management system for an electric car charging station using Java Collections and Generics. The system keeps track of car owners, their charging sessions, and manages charging ports with a waiting queue.

Tasks:

- In CarOwner class:
  - Implement chargingHistory using an appropriate List collection
  - Complete the addChargingSession method
- In ChargingManager class:
  - Implement carOwners storage using an appropriate Map collection
  - Complete the process arrival and departure methods for both new and existing car owners appropriately
- In ChargingStation class:
  - Implement chargingPorts storage using an appropriate List collection
  - Implement waiting queue using an appropriate Queue collection
  - Complete the handleCarOwnerArrival and handleCarOwnerDeparture methods
  - Make sure print statements are correct
  - Implement the queue management logic in checkWaitingQueue

[**No changes are needed in classes other than these**]

Requirements:

- All Collection class objects created must be genericised suitably
- Use appropriate Java Collections for each data structure
- ChargingStation should manage a first-come-first-served waiting queue
- CarOwner should maintain an ordered history of charging sessions
- ChargingManager should provide fast lookup of car owners by their names

The provided Main class will test your implementation with various scenarios. Your implementation should handle:

- New car owner registration and charging
- Existing car owner charging
- Waiting queue management when all ports are occupied
- Proper charging session recording and history display

Input Format

The program reads input in the following format:

- First line: An integer N representing the number of charging ports in the station
- Second line: An integer M representing the number of commands to process

- Following M lines: Commands in one of these formats:
    - For new car owner: ARRIVE name carMake carModel batteryCapacity
        - Example: ARRIVE John Tesla Model3 75.0
        - name: String representing owner's name
        - carMake: String representing car manufacturer
        - carModel: String representing car model
        - batteryCapacity: Double representing battery capacity in kWh
    - For existing car owner: ARRIVE name
        - Example: ARRIVE John
    - For departure: DEPART name
        - Example: DEPART John

Output Format

The system must output status messages in real-time as commands are processed. These messages must follow these exact formats:

1. Port Assignment Messages

When a car owner is assigned to a charging port:

```
[name] started charging at port [port_number]
```

Example: John started charging at port 0

2. Queue Messages

When all ports are occupied and a car owner must wait:

```
All ports are occupied. Adding [name] to the waiting queue.
```

Example: All ports are occupied. Adding Sarah to the waiting queue.

3. Departure Messages

When a car owner finishes charging and leaves:

```
[name] finished charging and left.
```

Example: John finished charging and left.

4. Error Messages

For various error conditions:

Error: [name] is not a registered car owner.

Error: [name] is not currently charging at any port.

5. Charging History Display

After processing all commands, the system must display the charging history for ALL car owners who have registered in the system, even if they haven't charged. The format must be:

```
[name]'s charging history:
Charging session: Energy charged: [energy] kWh
```

Where:

- Each owner's history starts on a new line
- Energy values must be formatted to 2 decimal places
- Each session must be on a new line
- Owners must be displayed even if they have no charging sessions
- Sessions must be displayed in chronological order

Example:

```
John's charging history:
Charging session: Energy charged: 60.00 kWh
Charging session: Energy charged: 60.00 kWh
Sarah's charging history:
Charging session: Energy charged: 64.40 kWh
```

Complete Example

Here's a complete input/output example:

Input:

```
2          // 2 charging ports
6          // 5 commands
ARRIVE John Tesla Model3 75.0
ARRIVE Sarah BMW i4 80.5
ARRIVE Mike Nissan Leaf 62.0
DEPART John
DEPART Sarah
DEPART Mike
```

Expected Output:

```
John started charging at port 0
Sarah started charging at port 1
All ports are occupied. Adding Mike to the waiting queue.
John finished charging and left.
Mike started charging at port 0
Sarah finished charging and left.
Mike finished charging and left.
Mike's charging history:
Charging session: Energy charged: 49.60 kWh
Sarah's charging history:
Charging session: Energy charged: 64.40 kWh
John's charging history:
Charging session: Energy charged: 60.00 kWh
```

Important Output Rules

- **Exact Formatting**

    - Use exact spacing and punctuation as shown
    - No extra blank lines between real-time messages
    - One blank line between each owner's charging history
    - Numbers must use exactly 2 decimal places
    - No trailing spaces at the end of lines
- **Numerical Formatting**
    - Energy values must be displayed with exactly 2 decimal places
    - Use dot (.) as decimal separator
    - Do not use thousands separators
    - Example: 60.00 kWh (correct), 60 kWh (incorrect), 60.0 kWh (incorrect)
- **Port Numbers**
    - Port numbers start from 0
    - Must be displayed as integers without padding
    - Example: port 0 (correct), port 00 (incorrect)

This output format will be used to automatically test your implementation, so precise adherence to the specification is required. Any deviation from this format, including extra spaces, missing newlines, or incorrect decimal places, will be considered incorrect.

Instructions:

Follow the steps given below to complete the OOP lab problem.

Step 1: Read the Lab Question

Read and understand the Lab problem given above.

Step 2: Give execute permission to the executable files

Use the following command to give execute permission to the executables.

```
:~$ chmod +x RunTestCase CreateSubmission
```

Step 3: See the input and the expected output

Use the following command to see the input and the expected output for test case T1. Note that L12 refers to Lab 12 , Q1 refers to Question 1and T1 refers to test case 1.

```
:~$ ./RunTestCase L12 Q1 YourBITSId T1
```

Use your own (13 character) BITS Id in place of YourBITSId in the above command. Type your BITS Id in upper case (capital letters). Ensure that you enter your BITS Id in 202XA7PSXXXXG format.

Run the command from within the folder containing the executables and the java files.

Step 4: Modify the solution java file

Modify the solution Java file(s) to solve the question given above. Repeat Step 3 until all the test cases are passed. The test cases are numbered T1, T2, etc.

Step 5: Passing evaluative test cases

There are evaluative test cases whose expected output is hidden. The evaluative test cases are numbered ET1, ET2, etc. The lab marks will be based on the evaluative test cases. Use the following command to check whether your solution passes the evaluative test cases.

```
:~$ ./RunTestCase L12 Q1 YourBITSId ET1
```

Ensure that your *final* solution passes all the evaluative test cases ET1, ET2, etc. The expected output of the evaluative test cases are *hidden*.

Step 6: Create submission zip file

Use the following command to create the submission zip file.

```
:~$ ./CreateSubmission L12 YourBITSId
```

The first command line argument must correspond to the lab number and the second command line argument must be your 13 character BITS id.

The above command will create a zip file. The command will also list the evaluative test cases that your solution program has passed.

Step 7: Upload submission zip file

Upload the submission zip file created in Step 6. Do not change the name of the zip file or modify any file inside the zip file. You will not be awarded marks if the zip file is tampered with in any manner.