Multithreading in Java - Restaurant Simulation

Objective:

Implement a multithreaded restaurant simulation in Java. The simulation should involve customers placing orders, chefs preparing food, waiters serving dishes, and a receptionist managing table assignments.

Requirements:

1. Customer Class:

- Create a `Customer` class that extends the `Thread` class.
- Each customer should have a unique identifier (e.g., customer ID).
- Customers should randomly order a combination of dishes (e.g., Burger, Pizza, Pasta).
- Customers' orders should go to the order queue.
- Display messages indicating when a customer is placing an order.
- Customer will be waiting (not thread sleep) after ordering until the food is served.
- The customer should be assigned a table by the receptionist using `Thread.join()`.

2. Chef Class:

- Create a `Chef` class that extends the `Thread` class.
- Each chef should have a unique identifier.
- The chef should take orders from the order queue and simulate cooking time for each dish.
- Display messages indicating when a chef is preparing a dish.
- After preparing food, it should be queued in the cooked food queue.

3. Waiter Class:

- Create a `Waiter` class that extends the `Thread` class.
- Each waiter should have a unique identifier.
- The waiter should serve dishes from the cooked food queue to customers.
- Display messages indicating when a waiter is serving a dish.
- Use `Thread.join()` to synchronize with the chef thread before serving the dish.

4. Receptionist Class:

- Create a `Receptionist` class that manages table assignments for customers.
- The receptionist should keep track of available tables and assign a table to each customer.
- Display messages indicating when a customer is assigned a table by the receptionist using `Thread.join()`.

5. Order Queue and Cooked Food Queue:

- Use shared data structures (e.g., `ArrayList` or `Queue`) to represent the order queue and the cooked food queue.
- You may use `Thread.join()` to coordinate the termination of threads accessing and modifying these queues as necessary.

6. Simulation:

- Implement a main class (`RestaurantSimulation`) that creates instances of customers, chefs, waiters, and the receptionist.
- Start all threads to simulate the concurrent activities in the restaurant.
- Specify the number of tables in the restaurant along with the number of chefs and waiters.
- Determine the duration of the simulation based on the number of customers served (i.e. the simulation will stop if "N" number of customers are served. Set an appropriate value to "N" as you wish).
- Use `Thread.join()` to ensure the correct termination of threads at the end of the simulation.

7. Output:

- Display relevant messages to show the flow of activities (e.g., customer placing an order, chef preparing food, waiter serving dishes, receptionist assigning tables).
- Use `Thread.join()` to ensure the correct termination of threads.

Submission:

Submit the Java source code files along **with a brief document explaining** the design choices, challenges faced, and how `Thread.join()` was implemented in the multithreaded simulation.

Note:

- You may use the `Thread.sleep()` method to simulate processing time for orders, cooking, serving, food consuming, and table assignment. The duration of sleep should be random.
- You may use `Thread.join()` to synchronize threads.
- Pay attention to synchronization issues to avoid data corruption or race conditions.