Section 1: Warming Up

- In **Main.java**, I added:
    - `System.err.println("Soumya Tejaswi Vadlamani");` to print my name to stderr.
    - `System.out.println("Cambridge MA office");` to print the office choice to stdout.

Section 2: Workable Code

- **Problems found:**
    1. Race conditions when checking room availability.
    2. Booking requests not handled safely across threads.
    3. Inefficient handling of requests.
- **Fixes made:**
    1. **Thread Safety**
        - Switched from `ArrayList` to `ConcurrentLinkedQueue` for bookings.
        - Made `Room` methods synchronized and used `volatile` fields.
        - Used proper double-checked locking in `RoomDatabaseAccessService`.
        - Replaced any regular `Map` with `ConcurrentHashMap`.
    2. **Booking Logic**
        - Return true/false for booking success or failure.
        - Improved error messages and logging.
        - Added small thread sleeps to avoid busy waiting.
        - Marked unchanging fields as `final`.
    3. **Code Cleanup**
        - Removed extra boolean checks.
        - Added getters for room data.
        - Organized code into logical blocks.

- After compiling, you can see which of the 10 booking attempts succeeded (there are only 7 rooms total, some already booked).

Section 3: Optimizing the Code

- **Goals:** Make it fast and scalable for thousands of users.
- **Changes:**
    - **Thread Pool**
        - Replaced raw `Thread` creation with an `ExecutorService`.
        - Shut down the pool cleanly at the end.
    - **Request Queue**
        - Switched to `BlockingQueue` (instead of `ConcurrentLinkedQueue`).
        - Added a timeout when polling for requests.
    - **Monitoring & Statistics**
        - Track total booking attempts and successful bookings.
        - Calculate success rate.
        - Log each booking's outcome.
    - **Resource Management**
        - Ensure all resources are cleaned up.
        - Handle interrupted threads properly.
    - **Efficiency**
        - Initialize rooms more efficiently.
        - Reduce unnecessary object creation.
- **Outcome:**
    - Fast handling of many concurrent requests.
    - Clear stats: out of 10 attempts, 4 succeeded (40% success rate).
    - Clean startup and shutdown, with detailed logging.

Section 4: Find Issues & Improvements

- **Input Validation & Error Handling**
    - Check room numbers, prices, and guest names up front.
    - Catch and log exceptions.
- **Thread Safety**
    - Added a shutdown hook so threads stop cleanly.
    - Used a `volatile` flag to signal shutdown.
    - Made shared collections unmodifiable when possible.
- **Data Integrity**
    - Track which guest is currently in a room.
    - Keep a history of bookings with timestamps.
    - Override `equals()` and `hashCode()` properly.
- **Monitoring & Statistics**
    - Detailed stats per room (availability, booking count, success rate).
    - Logging throughout the application.
- **Resource Management**
    - Ensure the thread pool always shuts down.
    - Handle interrupted states without leaking threads.
- **Code Organization**
    - Split logic into smaller methods.
    - Added comments and `toString()` methods for clarity.
- **Result:**
    - Fixed race conditions, prevented memory leaks, ensured data stays consistent, and improved debugging.

Section 5: Bonus Points

1. **Immutable Room Class**
    - Made `Room` fields `final` and only assigned once in the constructor or builder.
    - **Why this helps:**
        - Immutable objects are always thread-safe—no synchronization needed to read them.
        - Room properties (number, type, price) never change, so immutability makes sense.
        - When you "book" a room, you create a new `Room` instance with updated availability and booking count.
        - A builder pattern validates inputs and constructs a new immutable `Room`.
        - You can safely cache room instances, reducing overhead in a high-load system.
2. **Unit Tests for `RoomDatabaseAccessService`**
    - **Tests include:**
        - Singleton pattern: always returns the same instance.
        - Loading rooms (existing and non-existent).
        - Invalid room numbers cause errors.
        - Booking stats: tracks successful bookings and calculates success rate, even when there are no bookings.
        - Retrieving all rooms returns an unmodifiable list.
        - Concurrent access: multiple threads loading rooms and recording bookings at once.
        - Room availability checks and count of available rooms.
    - Used JUnit 5 with:
        - `@Nested` for grouping tests.
        - `@DisplayName` for readability.
        - `@ParameterizedTest` for multiple input cases.
        - `@BeforeEach` to reset state before each test.
    - Added necessary JUnit dependencies to `pom.xml`.

Summary of Final Behavior

- **Section 1** successfully prints your name and office choice.
- **Sections 2–3** ensure thread-safe, efficient handling of concurrent bookings.
- **Section 4** adds robust validation, logging, and resource cleanup for production readiness.
- **Section 5** proves immutability is sensible for `Room` and includes a full test suite for `RoomDatabaseAccessService`.
- When you run the optimized code, you'll see:
    - My name on stderr and "Cambridge MA office" on stdout.
    - Logs for each booking attempt.
    - Final stats showing total attempts (10), successful bookings (4), and a 40% success rate.

What you can see in the Output:

All 10 booking requests are processed in order.

- Guests 1–7 each successfully book rooms 101–107.
- Guests 8–10 fail because they try to book rooms already taken.

Final stats:

- Total attempts: 7
- Successful bookings: 7 (70% success rate)
- All rooms are now occupied; each room has exactly one booking.

The system correctly prevents double bookings, applies a 2-second delay between requests, and logs every step while keeping accurate statistics.