**Case Study 1: ConcurrentHashMap:**

**Case Study 1: Managing Real-Time User Sessions in a Multithreaded Environment**

**Background Context:**

In a web application, users log in and interact with the system. Each active user is assigned a session ID. To keep track of these sessions:

* The server needs to store session data, such as the user's last activity timestamp. (Thread 1 – to handle/update the timestamps.
* Sessions should be updated frequently—every time the user clicks, types, or refreshes.
* Sessions that have been inactive for too long (e.g., 10 minutes) should be automatically removed.

**Big Problem:**

“How can we safely store, update, and remove active user sessions in a web system where multiple threads operate concurrently without introducing data corruption or crashes?”

**Relate to Thread:**

Multiple threads are accessing and modifying this session data at the same time:

1. User Request Threads – Whenever a user interacts with the system, a thread updates their session timestamp.
2. Background Cleanup Thread – Periodically scans all sessions to remove any that have been inactive for too long.
3. Monitoring/Logging Threads – May read the session data to report statistics.

If not handled properly, this concurrent access can cause:

* ❌ Data inconsistency – A session might be deleted while it's being updated.
* ❌ Race conditions – Two threads update the same session at the same time, leading to incorrect or lost data.
* ❌ Crashes or exceptions – Accessing the session map while it's being modified can throw exceptions like ConcurrentModificationException.

To update for better performance: used ConcurrentHashMap

* **To** let multiple threads safely update and access session data.
* Cleanup runs without blocking the main application flow. – locking frameworks
* No need for synchronized blocks, avoiding performance bottlenecks.

**Sample Coding**

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class SessionManager {

private static final ConcurrentHashMap<String, Long> sessions = new ConcurrentHashMap<>();

private static final long SESSION\_TIMEOUT = 10\_000; // 10 seconds for demo

// Simulate user activity

public static void updateSession(String sessionId) {

sessions.put(sessionId, System.currentTimeMillis());

System.out.println("Updated session: " + sessionId);

}

// Background task to clean up old sessions

public static void startSessionCleanupTask() {

Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {

long now = System.currentTimeMillis();

for (String sessionId : sessions.keySet()) {

long lastActive = sessions.get(sessionId);

if (now - lastActive > SESSION\_TIMEOUT) {

sessions.remove(sessionId);

System.out.println("Removed expired session: " + sessionId);

}

}

}, 0, 5, TimeUnit.SECONDS);

}

// Monitor active sessions

public static void printSessions() {

System.out.println("Active Sessions:");

sessions.forEach((id, time) -> System.out.println(" - " + id + " (Last Active: " + time + ")"));

}

public static void main(String[] args) throws InterruptedException {

startSessionCleanupTask();

// Simulate user actions in different threads

Thread user1 = new Thread(() -> {

for (int i = 0; i < 5; i++) {

updateSession("user1");

try { Thread.sleep(2000); } catch (InterruptedException ignored) {}

}

});

Thread user2 = new Thread(() -> {

for (int i = 0; i < 3; i++) {

updateSession("user2");

try { Thread.sleep(4000); } catch (InterruptedException ignored) {}

}

});

user1.start();

user2.start();

user1.join();

user2.join();

// Final state

Thread.sleep(12000); // wait for cleanup to run

printSessions();

}

}