**1. Match - Generic Class**

The Match class is designed with the following fields:

* **homeTeam, awayTeam**:  
  Represented as String types for simplicity, given the straightforward nature of the task. In real case , these would probably need to be classes so they could store further information( form yellow cards players etc).
* **homeScore, awayScore**:  
  Defined as int to store goals, as they cannot be null and must be whole number
* **startedAt**:  
  Uses Instant to capture the exact timestamp of match creation, providing precision for sorting.
* **matchId**:  
  A unique identifier (int) for each match. This is required for efficient data manipulation in the scoreboard and resolves edge cases where two matches start at roughly the same time (see MatchComparator below).

**2. MatchComparator - Custom Comparator**

The MatchComparator sorts matches based on the following criteria (in order of priority):

1. **Total Goals**: Compares the sum of homeScore and awayScore across matches.
2. **Start Time**: Uses startedAt to break ties when total goals are equal.
3. **Match ID**: Acts as deal breaker when matches have same score and started t the same time, ensuring consistent sorting in rare edge cases.

This approach guarantees a deterministic sort order, even under concurrent match creation.

**3. Scoreboard - Core Data Structure**

The Scoreboard class includes the following components:

**Required Fields:**

* **private AtomicInteger matchCounter**:  
  Tracks the total number of matches and generates unique matchId values.
* **private Map<Integer, Match> matchesMap**:  
  Hashmap where we store all matches.Actually the only structure really needed for this task.Key is matchId whereas value is the Match. We could have went with approach where key is something like homeTeamvsawayTeam and then we would not need private TreeSet<Match> matches. However hashing of strings is much slower comparing to hashing of numbers and we need numbers (matchId) anyway for cases where 2 matches start at exactly the same time finish with the same result so we have something based on what we could perform sort
* **private TreeSet<Match> matches**:  
  A TreeSet maintaining a sorted collection of matches based on MatchComparator.
  + **Trade-offs**:
    - **Pros**: Enables efficient retrieval of sorted matches in getSummary (presumably the most-called method). The small memory expense that is worth of performance gains.
    - **Cons**: Updates in updateMatch are slower (requires removal, modification, and re-insertion). If updates were more frequent than getSummary calls, I’d consider using only matchesMap and sorting on demand.
  + **Alternative**: Sorting matchesMap values each time getSummary is called was an option, but this would sacrifice efficiency for a commonly used operation.

**Additional Fields:**

* **Lock**:  
  Included to ensure thread safety in a multithreaded environment, even though this wasn’t explicitly required in the task. In case if it was some competition with more matches at the same time I would have probably have a list of locks so I can enable parallel writing of updates at the same time.
* **cachedSummary**:  
  Added since I presumed getSummary would be most called method. Therefore instead of iterating through treeSet all the time when method is called we have a fairly simple cachedList. Even though in case of world cup this would be unsignificant improvement since there would not be many matches anyway

**Design Considerations**

* **Efficiency vs. Memory**:  
   I prioritized efficiency comparing to memory
* **Scalability**:  
  The matchId-based HashMap and TreeSet combination scales well for large datasets, avoiding the overhead of string-based keys or repeated sorting.
* **Flexibility**:  
  The structure allows for easy adaptation—e.g., removing TreeSet if update-heavy workloads dominate.