Dokumentation – Modul 323 Projekt

Projektübersicht

- Titel: Analyse von Fussballdaten – Spieler- und Teamstatistiken
- Ziel: Aus Rohdaten strukturierte Statistiken für Ligen/Saisons ableiten (Spiele, Teams, Tore).
- Versionen:
  - V1 Imperativ: klassische Schleifen, Maps/Listen.
  - V2 Funktional: Java Streams, Lambdas, Collector-APIs.
- Output: Einheitlich formatiert, konsolentauglich und ASCII-sicher.

Technologien

- Programmiersprache: Java 17
- Build/Run: `javac`, `java`, Skripte (`run.ps1`, `run.sh`, `run.bat`)
- Funktionale Elemente (V2):
  - Lambda Expressions (z. B. `x -> x + 1`)
  - Streams (`filter`, `map`, `flatMap`, `collect`, `reduce`)
  - Method References (z. B. `Integer::sum`)
  - Konzeptionell: Immutability via `final`

Beispiel (eigene Implementierung):

```
List<Integer> xs = List.of(1,2,3,4);
int sumSquares = xs.stream()
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

Datenbasis

- Dateien je Version: `v*/data/matches.csv`, `v*/data/players.csv`
- matches.csv (neues Format, genutzte Felder):
  - `League`, `Season` (z. B. 2012/2013 → Startjahr 2012), `Date`, `Home`, `Away`, `HG`, `AG`
  - Weitere Quotenfelder werden ignoriert.
- players.csv:

    - `player,team,league,season,goals`
    - Auf 2025 erweitert (Teams der Super League), plus ausgewählte 2024-Einträge.

Funktionen

- Filter: Spiele nach Liga und Saison
- Map: Tordifferenz je Spiel
- Reduce:
    - Gesamttore pro Team (alle Teams, sortiert)
    - Durchschnittstore pro Spiel (gesamt)
- Top-Scorer: Top 5 Spieler der gewählten Liga/Saison
- Teamvergleich: Tabelle je Team mit W/D/L, GF/GA, GD, Pkt (sortiert nach Pkt, GD, GF)
- Automatik:
    - Liga-Erkennung bevorzugt „Super League“, sonst meistvertretene Liga
    - Neueste Saison im Datensatz

Output-Format

- Spiele: `Datum | Home vs Away | X:Y`
- Tordifferenz: `Home - Away | Diff: N`
- Gesamttore pro Team: `Team | N Tore`
- Top-Scorer: `# Name | N Tore (Team)`
- Teamvergleich (Kopf):
    - `Team |  W  D  L |  GF  GA  GD | Pkt`
- Hinweis: ASCII-Separators (`|` , `:`) für konsistente Darstellung in verschiedenen Konsolen.

Code- und Projektstruktur

- `v*/src/Main.java` : Einstieg, Laden der CSVs, Analysen, Ausgabe
- `v*/src/model/Match.java` : Mapping einer CSV-Zeile zu Match (angepasst ans neue matches.csv)
- `v*/src/model/PlayerStat.java` : Spielerstatistiken aus players.csv
- `v*/src/util/CsvUtil.java` : CSV-Reader (Header wird übersprungen)

Imperativ vs. Funktional

- V1 Imperativ:

    - Listen/Maps manuell aufbauen
    - Sortierung via Comparatoren
- V2 Funktional:
    - Pipelines mit Streams und Collectors
    - Kompakter und ausdrucksstärker bei Aggregationen (Gruppieren, Summieren, Sortieren)
- Beide Versionen erzeugen denselben Output.

Ausführen

- Imperativ: `v1-imperative/run.ps1` (Windows), `v1-imperative/run.sh` (Unix), `v1-imperative/run.bat`
- Funktional: `v2-functional/run.ps1` (Windows), `v2-functional/run.sh` (Unix), `v2-functional/run.bat`
- Voraussetzungen: Java 17 in `PATH`

Beispielauszug (gekürzt)

- Spiele: `25/07/2025 | Zurich vs Sion | 2:3`
- Tordifferenz: `Basel - Luzern | Diff: 0`
- Gesamttore pro Team: `Basel |  19 Tore`
- Top-Scorer:
    - `1. Aiyegun Tosin | 20 Tore (Zurich)`
- Teamvergleich (Kopf):
    - `Team |  W  D  L |  GF  GA  GD | Pkt`

Fazit

- Nutzen funktionaler Elemente:
    - Kompaktere, klarere Datenpipelines
    - Weniger mutable State → geringere Fehleranfälligkeit
    - Flexible Umformulierung (Filter/Sorts/Aggregate austauschbar)
- Refactoring-Ergebnis:
    - V2 vereinfacht gruppierende und aggregierende Logik deutlich
    - Lesbarkeit steigt v. a. bei mehrstufigen Aggregationen

Siehe auch `README.md` für Kurzstart.