AGENDA

Kombikurs Java Streams: Basics + Advanced

3 Tage · Individueller Termin · Vor Ort oder Online · Deutsch / Englisch · Anpassbare Agenda · Teilnahmezertifikat

Tag 1: Streams Basics

Funktionale Programmierung in Java:

- Lambda-Ausdrücke: Syntax, Typen und Verwendung
- Funktionale Interfaces (`Consumer`, `Supplier`, `Function`, `Predicate`, etc.)
- Methodenreferenzen: Static, Instance, Constructor References
- Praktische Übungen: Lambda-Ausdrücke vs. anonyme Klassen

Stream-Grundlagen:

- Was ist ein Stream? Konzeptuelle Unterschiede zu Collections
- Stream-Pipeline Anatomie: Source → Intermediate Operations → Terminal Operations
- Lazy Evaluation und ihre Auswirkungen
- Stream-Erstellung: von Collections, Arrays, Files, Ranges
- Primitive Streams: `IntStream`, `LongStream`, `DoubleStream`
- Praktische Übungen: Erste Stream-Pipelines mit primitiven Streams

Intermediate Operations Deep Dive:

- Transformationen mit `map()`, `flatMap()`, `filter()`
- Auswahl und Sortierung: `distinct()`, `sorted()`, `limit()`, `skip()`
- Debugging mit `peek()` – richtig eingesetzt
- Chaining von Operations und Performance-Aspekte
- Praktische Übungen: Komplexere Datenverarbeitung

Terminal Operations:

- `collect()` – der vielseitigste Terminal Operator
- Eingebaute Collectors: `toList()`, `toSet()`, `toMap()` – Stream to Map leicht gemacht
- `forEach()`, `forEachOrdered()`
- `reduce()` für Aggregationen
- find-Operationen: `findFirst()`, `findAny()` – Rückgabetyp `Optional` verstehen und richtig verwenden
- Matching-Operationen: `allMatch()`, `anyMatch()`, `noneMatch()`
- Praktische Übungen: Terminal Operations mit Standard-Collectors

Tage 2 und 3: Streams Advanced

Advanced Collectors und Reduce Operations:

- `reduce()` für Aggregationen: Konzepte und praktische Anwendung
- `peek()` für Debugging: Wann sinnvoll, wann nicht
- Grouping: `groupingBy()`, `partitioningBy()`
- Downstream Collectors: `counting()`, `summingInt()`, `averagingDouble()`
- Nested Grouping und komplexe Aggregationen
- `joining()` für String-Operationen
- Praktische Übungen: Komplexe Datenanalyse mit Collectors

Custom Collectors entwickeln:

- Collector Interface verstehen: `supplier()`, `accumulator()`, `combiner()`, `finisher()`
- Characteristics: `CONCURRENT`, `IDENTITY_FINISH`, `UNORDERED`
- Einfache Custom Collectors implementieren
- Komplexere Scenarios: Custom Object Collection
- Praktische Übungen: Eigene Collectors schreiben

Stream Performance und Spliterator Grundlagen:

- Wann Streams verwenden vs. traditionelle Schleifen
- Performance-Fallen: Boxing/Unboxing, zu lange Pipelines
- Parallel Streams: Wann sinnvoll, Common Fork/Join Pool
- Memory-Effizienz und große Datenmengen
- Spliterator verstehen: Was ist ein Spliterator und seine Rolle
- Spliterator Characteristics: `ORDERED`, `SIZED`, `SUBSIZED`, `CONCURRENT`, etc.
- Wie Characteristics die Stream-Performance beeinflussen

Legacy Code Refactoring Workshop:

- Gemeinsame Refactoring-Session: Schleifen zu Streams
- Identifikation geeigneter Refactoring-Kandidaten
- Schritt-für-Schritt Transformation von Legacy Code
- Performance-Vergleiche: Vorher/Nachher
- Code-Readability vs. Performance Trade-offs
- Wann Refactoring sinnvoll ist und wann nicht
- Praktische Übungen: Verschiedene Legacy-Szenarien

Fortgeschrittene Stream-Patterns und Custom Spliterators:

- Conditional Operations mit `Optional`
- Error Handling in Streams
- Custom Spliterator implementieren: Eigene Datenstrukturen für Streams
- Spliterator Patterns: `tryAdvance()`, `trySplit()`, `estimateSize()`
- Parallelisierung optimieren durch custom Spliterators
- Stream von Files: Lines, Directory Walking
- Infinite Streams: `generate()`, `iterate()`
- Praktische Übungen: Custom Spliterator für spezielle Datenstrukturen

Stream Gatherers (ab Java 22, finalisiert in Java 24):

- Was sind Stream Gatherers und warum wurden sie eingeführt
- Gatherer Interface verstehen: `integrator()`, `combiner()`, `finisher()`
- Built-in Gatherers: `fold()`, `scan()`, `windowFixed()`, `windowSliding()`
- Custom Gatherers implementieren
- State Management in Gatherers
- Praktische Übungen: Eigene Gatherers für spezielle Use Cases

Integration und Architecture Patterns:

- Streams in Repository Pattern
- Functional Programming Principles mit Streams
- Stream-basierte APIs designen
- Migration Strategies: Legacy Code zu Streams
- Anti-Patterns und Debugging: Typische Fehler vermeiden
- Testing von Stream-Code

Workshop und Problemlösung:

- Komplexes Praxisprojekt: Ende-zu-Ende Stream-Pipeline
- Code Review Session: Teilnehmer-Code analysieren
- Performance-Optimierung an realen Beispielen
- Q&A und individuelle Problemstellungen
- Best Practices Zusammenfassung