AGENDA

Combined Training: Java Streams Basics + Advanced

3 days · Individual date · On-site or online · German / English · Customizable agenda · Certificate of completion

Day 1: Streams Basics

Functional Programming in Java:

- Lambda expressions: syntax, types, and usage
- Functional interfaces (`Consumer`, `Supplier`, `Function`, `Predicate`, etc.)
- Method references: static, instance, constructor references
- Hands-on exercises: Lambda expressions vs. anonymous classes

Stream Basics:

- What is a stream? Conceptual differences from collections
- Anatomy of a stream pipeline: Source → Intermediate Operations → Terminal Operations
- Lazy evaluation and its implications
- Creating streams: from collections, arrays, files, ranges
- Primitive streams: `IntStream`, `LongStream`, `DoubleStream`
- Hands-on exercises: First stream pipelines with primitive streams

Intermediate Operations Deep Dive:

- Transformations with `map()`, `flatMap()`, `filter()`
- Selection and sorting: `distinct()`, `sorted()`, `limit()`, `skip()`
- Debugging with `peek()` – used correctly
- Chaining operations and performance aspects
- Hands-on exercises: More complex data processing

Terminal Operations:

- `collect()` – the most versatile terminal operator
- Built-in collectors: `toList()`, `toSet()`, `toMap()`
- `forEach()`, `forEachOrdered()`
- `reduce()` for aggregations
- Finding operations: `findFirst()`, `findAny()`
- Matching operations: `allMatch()`, `anyMatch()`, `noneMatch()`
- Hands-on exercises: Terminal operations with standard collectors

Days 2 and 3: Streams Advanced

Advanced Collectors and Reduce Operations:

- Aggregation with `reduce()`: concepts and practical use
- Debugging with `peek()`: when it's helpful and when not
- Grouping: `groupingBy()`, `partitioningBy()`
- Downstream collectors: `counting()`, `summingInt()`, `averagingDouble()`
- Nested grouping and complex aggregations
- Joining strings with `joining()`
- Hands-on: complex data analysis with collectors

Building Custom Collectors:

- Understanding the `Collector` interface: `supplier()`, `accumulator()`, `combiner()`, `finisher()`
- Characteristics: `CONCURRENT`, `IDENTITY_FINISH`, `UNORDERED`
- Implementing simple custom collectors
- More complex use cases: collecting custom objects
- Hands-on: writing your own collectors

Stream Performance and Spliterator Basics:

- When to use streams vs. traditional loops
- Performance pitfalls: boxing/unboxing, long pipelines
- Parallel streams: when they make sense, common fork/join pool
- Memory efficiency and large datasets
- Understanding spliterators and their role
- Spliterator characteristics: `ORDERED`, `SIZED`, `SUBSIZED`, `CONCURRENT`, etc.
- How characteristics affect stream performance

Legacy Code Refactoring Workshop:

- Refactoring session: transforming loops into streams
- Identifying good refactoring candidates
- Step-by-step transformation of legacy code
- Performance comparisons: before vs. after
- Readability vs. performance trade-offs
- When refactoring is worthwhile – and when it's not
- Hands-on: multiple legacy scenarios

Advanced Stream Patterns and Custom Spliterators:

- Conditional operations with `Optional`
- Error handling in streams
- Implementing custom spliterators for specific data structures
- Spliterator patterns: `tryAdvance()`, `trySplit()`, `estimateSize()`
- Improving parallelism with custom spliterators
- File streams: reading lines, directory walking
- Infinite streams: `generate()`, `iterate()`
- Hands-on: building a custom spliterator

Stream Gatherers (since Java 22, finalized in Java 24):

- What are stream gatherers and why were they introduced
- Understanding the `Gatherer` interface: `integrator()`, `combiner()`, `finisher()`
- Built-in gatherers: `fold()`, `scan()`, `windowFixed()`, `windowSliding()`
- Implementing your own gatherers
- State management in gatherers
- Hands-on: building gatherers for specific use cases

Integration and Architecture Patterns:

- Streams in repository patterns
- Functional programming principles with streams
- Designing stream-based APIs
- Migration strategies: legacy code to streams
- Anti-patterns and debugging: avoiding common mistakes
- Testing stream-based code

Workshop and Problem Solving:

- Complex real-world project: end-to-end stream pipeline
- Code review session: analyzing participant code
- Performance tuning based on real examples
- Q&A and individual use cases
- Summary of best practices
