Cognixia®

COURSE CONTENT

www.cognixia.com

COURSE NAME

JDK 17+

DURATION

16 Half Days (64 Hours)

PREREQUISITES

- Fundamental Understanding of Agile practices and Working experience on Java platform

COURSE OUTLINE

INTRODUCTION TO JAVA EVOLUTION

- Overview of Java versions and release cycle
- The business case for upgrading Java
- Long-Term Support (LTS) and its significance
- Key differences and performance benefits between versions
- Migration considerations and challenges

CORE LANGUAGE FEATURES & ENHANCEMENTS

FUNCTIONAL PROGRAMMING

- Lambda expressions and syntax
- Functional interfaces architecture
- Predicate, Consumer, Function, and Supplier interfaces
- Creating custom functional interfaces
- Using lambdas with collections and concurrency
- Method references and their applications
- Local Variable Syntax for Lambda Parameters

INTERFACE EVOLUTION AND CAPABILITIES

- Default methods implementation in interfaces
- Static methods in interfaces
- Private methods in interfaces
- Interface inheritance hierarchies
- Best practices for modular and maintainable code

TYPE SYSTEM IMPROVEMENTS

- Local-variable type inference with var
- Best practices and style guidelines

- Appropriate use cases and situations to avoid
- Impact on code readability and maintenance
- Performance implications of type inference

ENHANCED CONDITIONAL LOGIC

- Switch expressions with arrow syntax
- Multiple case labels in a single case
- Yield keyword for returning values
- Pattern matching in switch statements
- Complex switch statement optimization

PATTERN MATCHING AND TYPE CHECKING

- Pattern matching for instanceof
- Type patterns and variable binding
- Eliminating redundant type casting
- Flow scoping and variable lifecycle
- Simplifying conditional logic chains

COLLECTIONS, STREAMS, AND DATA HANDLING

FUNCTIONAL PROGRAMMING WITH STREAMS

- Stream API architecture and pipeline operations
- Lazy evaluation and execution optimization
- Functional-style data processing patterns
- Terminal and intermediate operations
- Map, filter, and reduce operations
- Performance benchmarking and optimization
- Parallel Streams Usage
- Performance considerations and when to use parallel streams
- Fork/Join framework fundamentals
- Common pitfalls and best practices
- Measuring performance improvements

ADVANCED STREAM OPERATIONS

- Advanced streaming methods
- takeWhile(), dropWhile(), iterate()
- Working with specialized streams
- Collectors and aggregation operations
- Teeing collectors for dual-result collection
- Best practices for stream performance

COLLECTION FACTORY METHODS

- Creating immutable collections
- List, Set, and Map factory methods
- Memory and performance optimizations
- Comparison with traditional collection creation

OPTIONAL API AND NULL SAFETY

- Preventing NullPointerException
- Optional chaining with map() and flatMap()
- Retrieving values with orElse(), orElseGet(), orElseThrow()
- Best practices for Optional usage
- Common anti-patterns to avoid

OBJECT-ORIENTED PROGRAMMING ENHANCEMENTS

RECORDS FOR DATA MODELING

- Simplifying data carrier classes
- Implicit methods: constructor, getters, equals, hashCode, toString
- Customizing canonical constructors
- Adding validation logic and custom methods
- Use cases: DTOs, API models, immutable data
- Composition patterns with records

SEALED CLASSES FOR CONTROLLED INHERITANCE

- Restricting class hierarchies
- Permitted subclasses declaration
- Integration with final and non-sealed modifiers
- Compiler verification and validation
- Relationship with pattern matching
- Domain modeling with sealed types

SOFTWARE DESIGN PRINCIPLES AND PATTERNS

SOLID PRINCIPLES IN EVERYDAY CODING

- Single Responsibility Principle (SRP) applied to Java classes
- Open/Closed Principle (OCP) with abstractions and interfaces
- Liskov Substitution Principle (LSP) in inheritance hierarchies
- Interface Segregation Principle (ISP) with focused interfaces
- Dependency Inversion Principle (DIP) implementation techniques
- Practical refactoring examples to apply SOLID principles

- Code smells that indicate SOLID violations

CREATIONAL DESIGN PATTERNS:

- Singleton: Lazy initialization and Thread-Safe implementation
- Factory: Creating objects without specifying concrete classes
- Abstract Factory: Families of related objects
- Builder: Step-by-step construction of complex objects
- Prototype: Cloning objects to avoid expensive instantiation

STRUCTURAL DESIGN PATTERNS:

- Adapter: Interfacing incompatible classes
- Facade: Simplified interface to complex subsystems
- Bridge: Separating abstraction from implementation
- Decorator: Attaching additional responsibilities dynamically
- Proxy: Surrogate for controlling access to objects
- Composite: Tree structures of simple and composite objects
- Flyweight: Sharing for efficient handling of large numbers of objects

BEHAVIORAL DESIGN PATTERNS:

- Template Method: Algorithm skeleton with subclass-defined steps
- Observer: One-to-many dependency between objects
- Chain of Responsibility: Passing requests along a handler chain
- Strategy: Encapsulating interchangeable algorithms
- Mediator: Reducing dependencies between objects
- Visitor: Operations to be performed on elements of an object structure

CONCURRENCY AND ASYNCHRONOUS PROGRAMMING

ASYNCHRONOUS PROGRAMMING WITH COMPLETABLEFUTURE

- Key features of CompletableFuture
- Asynchronous Execution: Tasks running in separate threads
- Non-blocking Operations: thenApply, thenAccept, and thenCompose
- Composition and Combination: Creating complex asynchronous workflows
- Error Handling: Managing exceptions during asynchronous operations
- Manual Completion: Controlling future completion states
- Real-world application patterns

PLATFORM AND RUNTIME ENHANCEMENTS

STRING AND TEXT PROCESSING IMPROVEMENTS

- Compact Strings architecture
- Memory and performance optimizations
- Encoding efficiency between Latin-1 and UTF-16
- Performance benchmarks and measurements
- Text Blocks for multi-line strings
- Incidental whitespace handling
- Escape sequences and string formatting

INTERNAL API ENCAPSULATION

- Understanding strong encapsulation principles
- Reflection access restrictions
- Replacement APIs for common internal usage
- Migration strategies for legacy applications
- Security benefits of enhanced encapsulation

MODULARIZATION AND CODE ORGANIZATION

THE JAVA MODULE SYSTEM

- Fundamentals of modular programming
- Module declarations and module-info.java
- Module directives: requires, exports, opens, uses, provides
- Strong encapsulation and information hiding
- Service provider interfaces in modular systems
- Migration from classpath to module path
- Multi-release JAR files
- Module resolution algorithms
- Performance benefits and reduced footprint

DEBUGGING AND DIAGNOSTICS

STACK TRACE ANALYSIS

- StackWalker API implementation
- Efficient stack frame access
- Filtering and limiting capabilities
- Performance benefits over traditional approaches
- Integration with logging and diagnostics
- Custom stack frame processing

TEXT AND NUMBER FORMATTING

ADVANCED STRING HANDLING

- Multi-line text processing
- Indentation and formatting control
- Escape sequence handling
- Best practices for readability and maintenance

NUMBER FORMATTING AND LOCALIZATION

- CompactNumberFormat implementation
- Locale-aware number representation
- Customization options and patterns
- Financial and scientific notation
- Internationalization best practices

TEST-DRIVEN DEVELOPMENT (TDD) APPROACH

- TDD life cycle: Red-Green-Refactor
- Writing effective unit tests with JUnit 5
- Parameterized tests and dynamic tests
- Test extensions and custom annotations
- Mocking dependencies with Mockito
- Test coverage measurement and reporting
- Integration testing strategies in modular applications
- Property-based testing techniques

MIGRATION STRATEGIES AND BEST PRACTICES

- Upgrading Legacy Applications
- Step-by-step migration approach
- Handling deprecated and removed APIs
- Managing reflective access changes
- Testing strategies for compatibility
- Performance validation post-migration
- Tooling for migration assistance

THE FUTURE OF JAVA

- Future language feature roadmap
- Adapting to the rapid release cycle
- Best practices for writing future-proof code
- Community resources and continued learning

HANDS-ON LABS

- Modernizing legacy code with new language features
- Functional programming with streams and lambdas
- Implementing domain models with records and sealed classes
- Building modular applications
- Performance benchmarking and optimization
- Implementing and refactoring with design patterns
- Applying SOLID principles to existing codebases
- Working with CompletableFuture for async operations
- Developing applications using Test-Driven Development