Intermediate Java

Duration: 5days | NTJVA201

Description

nTier's intermediate Java training course teaches programming focuses on core Object Oriented Principles and concepts, Unit Testing and Test Driven Development, key Java functionalities. It is intended for students with some previous Java experience or training and who already know the fundamentals of the Java architecture and basic procedural programming. This course provides in-depth coverage of object-oriented concepts and how to apply them to Java software design and development. This includes the introduction of some core design patterns and the advantages that they bring to an extensible design.

Students must be able to write, compile, test and debug simple Java programs using structured programming techniques, strong data types and flow- control constructs such as conditionals and loops. At least six months of Java experience is recommended.

Objectives

- Chiefly, learn to program effectively in the Java language
- Understand Java as a purely object-oriented language, and implement software as systems of classes
- Implement and use inheritance and polymorphism, including interfaces and abstract classes
- Utilize Functional Interfaces i.e. Lambdas, Generics, Optional and LocalDateTime API
- Understand the structure of streams in Java including predicates, mapping, reducing and finding
- Use JUnit capabilities
- Use Test Driven development (TDD) to evolve the design, utilize common refactoring techniques to overcome code smells
- Learn how to leverage Files, and how to use streams to manage file I/O
- Learn how to use Java Serialization to internalize and externalize potentially complex graphs of objects

Prerequisites

Students must be able to write, compile, test and debug simple Java programs using structured programming techniques, strong data types and flow-control constructs such as conditionals and loops. At least six months of Java experience is recommended.

Outline

- Object-Oriented Software
    - Complex Systems
    - Abstraction
    - Classes and Objects

  - Responsibilities and Collaborators
  - Lightweight Design
  - Relationships
  - Visibility

- Inheritance and Polymorphism in Java
  - Extending Classes
  - Using Derived Classes
  - Type Identification
  - Compile-Time and Run-Time Type
  - Polymorphism
  - Overriding Methods
  - Superclass Reference
  - Lab – Build by extension rather than modification

- Interfaces and Abstract Classes
  - Separating Interface and Implementation
  - Designing Interfaces and Interface Realization
  - Implementing and Extending Interfaces
  - Default Methods, Static Methods and Functional Interfaces (Lambda)
  - Abstract Classes
  - Lab - Advanced Interfaces

- Generics
  - Raw Types
  - Generic Methods and Bounded Type Parameters
  - Upper and Lower Bounded WildCards
  - Type Erasure
  - Lab – Using Generics

- Unit Testing
  - JUnit
  - JUnit Life Cycle, @BeforeAll, @BeforeEach, @Test, @AfterEach, @AfterAll,
  - Using assertThat and Hamcrest Matchers
  - Parameterized Tests
  - Testing Exceptions
  - Lab – Unit Testing

- Test Driven Development (TDD)
  - Defining your API
  - Writing tests First, Design Test, Code, refactor cycles
  - Refactoring techniques and code Smells
  - Test Antipatterns
  - Lab TDD

- Understanding Patterns
  - Refactoring to Patterns
  - Factories and Builders
  - Lab - Creational pattern
  - Strategy

  - Adapter
  - Template Method
  - Lab – Structural and behavior patterns

- Lambdas
  - Functional Interfaces
  - Java.util.function package - Consumer, BiConsumer, Function, BiFunction and Predicate
  - Generics and Type coercion in Lambdas
  - Creating your own Functional interfaces
  - Method References
  - Lab - using Lambdas

- Streams
  - Streams vs Collections
  - Intermediate and Terminal Operations
  - Filtering with Predicates
  - Mapping and transforming, flatMaps
  - Reducing, numeric Streams
  - Lab – simple streams
  - Collectors
  - Grouping
  - Finding elements in streams
  - Parallel Streams
  - Fork and Join framework
  - Lab – Parallel streams, executor

- Optional
  - Representing optional values instead of using null references
  - Optional.of(), Optional.ofNullable()
  - Using get(), orElse(), isPresent()
  - Lab - using Optional

- Date and Time API
  - LocalDateTime
  - Formatting Dates
  - Parsing date time objects
  - Lab - LocalDateTime

- Working with Files
  - Delegation-Based Stream Model
  - InputStream and OutputStream
  - Media-Based Streams
  - Filtering Streams
  - Readers and Writers
  - File Class
  - Modeling Files and Directories
  - File Streams
  - Random-Access Files
  - Lab – Readers and writers

- Java Serialization
    - The Challenge of Object Serialization
    - Serialization API
    - Serializable Interface
    - ObjectInputStream and ObjectOutputStream
    - The Serialization Engine
    - Transient Fields
    - readObject and writeObject
    - Externalizable Interface
    - JAXB marshalling and unmarshalling
    - Lab - Serialization