# Modern Java Programming Course

## A Career-Focused Curriculum for Java 21+

**Course Philosophy:** Learn Java by solving real problems using modern approaches. Build professional coding habits from day one while understanding *why* concepts exist.

**Target Audience:** Individuals seeking a career in Java development with no prior Java experience.

**Java Version:** Java 21 (LTS) with modern features including Records, Sealed Classes, Pattern Matching, and Virtual Threads.

---

## Table of Contents

---

# Course Overview

## What Makes This Course Different

- ✅ **Modern Java First:** No legacy patterns to unlearn
- ✅ **Problem-Driven:** Understand *why* before *what*
- ✅ **Industry-Ready:** Write code that professionals would write
- ✅ **Multi-Paradigm:** Learn when to use OOP, FP, or Data-Oriented approaches
- ✅ **Practical Projects:** Every concept has real-world application

## Prerequisites

- Basic computer literacy
- Logical thinking skills
- Willingness to learn and experiment
- No prior programming experience required

## Learning Outcomes

By the end of this course, you will:
- Write professional-grade Java code using Java 21 features
- Understand when to apply different programming paradigms
- Build scalable, maintainable applications
- Handle errors gracefully and build resilient systems
- Write comprehensive tests for your code
- Design and implement RESTful APIs
- Work with modern concurrency patterns

# Learning Approach

## Problem → Solution → Concept

Each exercise follows this pattern:

1. **Problem Introduction**
   - Real-world scenario
   - Why this problem matters
   - Common approaches
2. **Naive Solution**
   - What beginners typically try
   - Why it's problematic
   - Performance/maintainability issues
3. **Modern Java Solution**
   - Leveraging Java 21 features
   - Clean, readable code
   - Professional patterns
4. **Trade-offs Discussion**
   - When to use this approach
   - When NOT to use it
   - Alternative solutions
5. **Real-World Context**
   - Where you'd encounter this
   - Industry examples
   - Common variations

# Module 1: Java Fundamentals & Problem Solving

**Duration:** 2 Weeks

## Concepts Covered

- ► Variables, types, and type safety
- ► Immutability and why it matters
- ► Control flow with modern syntax
- ► Methods and pure functions
- ► Basic I/O operations
- ► Records and simple data modeling
- ► Switch expressions with pattern matching

## Exercise 1: Temperature Converter

**Problem:** Convert temperatures between Celsius, Fahrenheit, and Kelvin.

**Why It Exists:** Data transformation is fundamental. Understanding type safety and immutability starts here.

**Code Example:**

```java
package com.javacourse.fundamentals;

/**
 * Temperature converter demonstrating records, enums, and switch expressions
 */
public record Temperature(double value, Unit unit) {

    public enum Unit {
        CELSIUS, FAHRENHEIT, KELVIN
    }

    /**
     * Convert temperature to target unit
     * Demonstrates: immutability, switch expressions, pattern matching
     */
    public Temperature convertTo(Unit targetUnit) {
        if (this.unit == targetUnit) {
            return this;
        }

        // First convert to Celsius as base
        double celsius = switch (this.unit) {
            case CELSIUS -> value;
            case FAHRENHEIT -> (value - 32) * 5 / 9;
            case KELVIN -> value - 273.15;
        };

        // Then convert from Celsius to target
        double result = switch (targetUnit) {
            case CELSIUS -> celsius;
            case FAHRENHEIT -> celsius * 9 / 5 + 32;
            case KELVIN -> celsius + 273.15;
        };

        return new Temperature(result, targetUnit);
    }

    /**
     * Format temperature for display
     */
    public String format() {
        return "%.2f°%s".formatted(value, switch (unit) {
            case CELSIUS -> "C";
            case FAHRENHEIT -> "F";
            case KELVIN -> "K";
        });
    }
}

// Usage
class TemperatureDemo {
    public static void main(String[] args) {
        var temp = new Temperature(100, Temperature.Unit.CELSIUS);
        System.out.println("Original: " + temp.format());

        var fahrenheit = temp.convertTo(Temperature.Unit.FAHRENHEIT);
        System.out.println("In Fahrenheit: " + fahrenheit.format());

        var kelvin = temp.convertTo(Temperature.Unit.KELVIN);
        System.out.println("In Kelvin: " + kelvin.format());
    }
}
```

**Skills Learned:**
- Records for immutable data
- Enums for type-safe constants
- Switch expressions (not statements!)
- Method chaining
- String formatting with `formatted()`

**Real-World Application:** APIs that transform measurement units, IoT sensor data processing, scientific calculations.

---

# Exercise 2: Grade Calculator

**Problem:** Calculate letter grades and handle various edge cases (incomplete work, invalid scores).

**Why It Exists:** Business logic needs to be expressed clearly. Type systems can prevent errors.

**Code Example:**

```java
package com.javacourse.fundamentals;

/**
 * Grade calculator using sealed interfaces for type-safe results
 */
public class GradeCalculator {

    /**
     * Sealed interface ensures we handle ALL possible outcomes
     * Compiler will warn if we miss a case
     */
    public sealed interface GradeResult
        permits Pass, Fail, Incomplete {}

    public record Pass(double score, String letter) implements GradeResult {}
    public record Fail(double score, String reason) implements GradeResult {}
    public record Incomplete(String reason) implements GradeResult {}

    /**
     * Calculate grade with all edge cases handled
     * Demonstrates: sealed types, pattern matching, validation
     */
    public static GradeResult calculateGrade(
        double score,
        boolean assignmentComplete) {

        // Validation
        if (!assignmentComplete) {
            return new Incomplete("Assignment not submitted");
        }

        if (score < 0 || score > 100) {
            return new Fail(score, "Invalid score range");
        }

        // Grade calculation with pattern matching
        return switch (score) {
            case double s when s >= 90 -> new Pass(s, "A");
            case double s when s >= 80 -> new Pass(s, "B");
            case double s when s >= 70 -> new Pass(s, "C");
            case double s when s >= 60 -> new Pass(s, "D");
            default -> new Fail(score, "Below passing threshold");
        };
    }

    /**
     * Format result for display
     * Pattern matching in switch ensures all cases handled
     */
    public static String formatResult(GradeResult result) {
        return switch (result) {
            case Pass(var score, var letter) ->
                "PASSED: Score %.1f - Grade %s".formatted(score, letter);
            case Fail(var score, var reason) ->
                "FAILED: Score %.1f - %s".formatted(score, reason);
            case Incomplete(var reason) ->
                "INCOMPLETE: " + reason;
        };
    }
}

// Usage
class GradeDemo {
    public static void main(String[] args) {
        var results = java.util.List.of(
            GradeCalculator.calculateGrade(95.5, true),
            GradeCalculator.calculateGrade(65.0, true),
```

```
                GradeCalculator.calculateGrade(50.0, true),
                GradeCalculator.calculateGrade(85.0, false)
            );

            results.forEach(result ->
                System.out.println(GradeCalculator.formatResult(result))
            );
        }
    }
```

**Skills Learned:**
- Sealed interfaces for algebraic data types
- Pattern matching with guards ( `when` clause)
- Exhaustiveness checking by compiler
- Domain modeling with types

**Real-World Application:** Student management systems, performance evaluation tools, certification systems.

---

# Exercise 3: Shopping Cart Calculator

**Problem:** Calculate total cost including tax and discounts.

**Why It Exists:** Money handling requires precision. Streams enable clean data processing.

**Code Example:**

```java
package com.javacourse.fundamentals;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.List;

/**
 * Shopping cart with proper money handling
 */
public class ShoppingCart {

    public record Item(
        String name,
        BigDecimal price,
        int quantity) {

        public Item {
            if (quantity < 1) {
                throw new IllegalArgumentException(
                    "Quantity must be positive"
                );
            }
            if (price.compareTo(BigDecimal.ZERO) < 0) {
                throw new IllegalArgumentException(
                    "Price cannot be negative"
                );
            }
        }

        public BigDecimal subtotal() {
            return price.multiply(BigDecimal.valueOf(quantity));
        }
    }

    private final List<Item> items;
    private final BigDecimal taxRate;

    public ShoppingCart(List<Item> items, BigDecimal taxRate) {
        this.items = List.copyOf(items); // Defensive copy
        this.taxRate = taxRate;
    }

    /**
     * Calculate subtotal using streams
     */
    public BigDecimal subtotal() {
        return items.stream()
            .map(Item::subtotal)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }

    /**
     * Calculate tax amount
     */
    public BigDecimal tax() {
        return subtotal()
            .multiply(taxRate)
            .setScale(2, RoundingMode.HALF_UP);
    }

    /**
     * Calculate total with tax
     */
    public BigDecimal total() {
        return subtotal().add(tax());
    }
```

```
    /**
     * Apply discount percentage
     */
    public BigDecimal totalWithDiscount(BigDecimal discountPercent) {
        var discount = total()
            .multiply(discountPercent.divide(
                BigDecimal.valueOf(100),
                2,
                RoundingMode.HALF_UP
            ));
        return total().subtract(discount);
    }

    /**
     * Get formatted receipt
     */
    public String getReceipt() {
        var receipt = new StringBuilder();
        receipt.append("=" .repeat(40)).append("\n");
        receipt.append("RECEIPT\n");
        receipt.append("=".repeat(40)).append("\n\n");

        items.forEach(item -> {
            receipt.append("%s x%d @ $%s = $%s\n".formatted(
                item.name(),
                item.quantity(),
                item.price(),
                item.subtotal()
            ));
        });

        receipt.append("\n");
        receipt.append("Subtotal: $%s\n".formatted(subtotal()));
        receipt.append("Tax: $%s\n".formatted(tax()));
        receipt.append("=" .repeat(40)).append("\n");
        receipt.append("TOTAL: $%s\n".formatted(total()));
        receipt.append("=".repeat(40)).append("\n");

        return receipt.toString();
    }
}

// Usage
class CartDemo {
    public static void main(String[] args) {
        var items = List.of(
            new ShoppingCart.Item(
                "Java Programming Book",
                new BigDecimal("49.99"),
                2
            ),
            new ShoppingCart.Item(
                "Wireless Mouse",
                new BigDecimal("25.50"),
                1
            ),
            new ShoppingCart.Item(
                "USB Cable",
                new BigDecimal("12.99"),
                3
            )
        );

        var cart = new ShoppingCart(items, new BigDecimal("0.08")); // 8% tax
        System.out.println(cart.getReceipt());

        System.out.println("With 10% discount: $" +
            cart.totalWithDiscount(new BigDecimal("10")));
```

```
    }
}
```

**Skills Learned:**
- BigDecimal for money (never use `double`!)
- Streams with `map` and `reduce`
- Method references (Item::subtotal)
- Immutable collections with defensive copying
- Compact constructors for validation

**Real-World Application:** E-commerce platforms, point-of-sale systems, invoicing software.

---

# Module 2: Collections & Data Structures

**Duration:** 2 Weeks

## Concepts Covered

- ▶ Collections Framework overview
- ▶ Choosing the right data structure
- ▶ Time and space complexity basics
- ▶ Immutable vs mutable collections
- ▶ Map, List, Set usage patterns
- ▶ Optional for null-safety
- ▶ Stream operations

## Exercise 4: Student Registry System

**Problem:** Manage student records with efficient lookup and filtering.

**Why It Exists:** CRUD operations are fundamental to most applications. Data retrieval patterns matter.

**Code Example:**

```java
package com.javacourse.collections;

import java.util.*;
import java.util.stream.Collectors;

public record Student(String id, String name, int age, String major) {}

/**
 * Student registry demonstrating Map usage and stream filtering
 */
public class StudentRegistry {
    private final Map<String, Student> studentsById;

    public StudentRegistry() {
        this.studentsById = new HashMap<>();
    }

    /**
     * Add or update student
     */
    public void save(Student student) {
        studentsById.put(student.id(), student);
    }

    /**
     * Find student by ID
     * Returns Optional to handle absence safely
     */
    public Optional<Student> findById(String id) {
        return Optional.ofNullable(studentsById.get(id));
    }

    /**
     * Find students by age range
     * Demonstrates: streaming map values, filtering
     */
    public List<Student> findByAgeRange(int minAge, int maxAge) {
        return studentsById.values().stream()
            .filter(s -> s.age() >= minAge && s.age() <= maxAge)
            .sorted(Comparator.comparing(Student::name))
            .toList();
    }

    /**
     * Find students by name pattern
     */
    public List<Student> findByNamePattern(String pattern) {
        return studentsById.values().stream()
            .filter(s -> s.name()
                .toLowerCase()
                .contains(pattern.toLowerCase()))
            .toList();
    }

    /**
     * Get students grouped by major
     */
    public Map<String, List<Student>> groupByMajor() {
        return studentsById.values().stream()
            .collect(Collectors.groupingBy(Student::major));
    }

    /**
     * Get average age by major
     */
    public Map<String, Double> averageAgeByMajor() {
        return studentsById.values().stream()
```

```
            .collect(Collectors.groupingBy(
                Student::major,
                Collectors.averagingInt(Student::age)
            ));
    }

    /**
     * Delete student
     */
    public boolean delete(String id) {
        return studentsById.remove(id) != null;
    }

    /**
     * Get all students
     */
    public List<Student> findAll() {
        return new ArrayList<>(studentsById.values());
    }
}

// Usage
class RegistryDemo {
    public static void main(String[] args) {
        var registry = new StudentRegistry();

        // Add students
        registry.save(new Student("S001", "Alice Johnson", 20, "CS"));
        registry.save(new Student("S002", "Bob Smith", 22, "Math"));
        registry.save(new Student("S003", "Charlie Brown", 19, "CS"));
        registry.save(new Student("S004", "Diana Prince", 21, "Physics"));

        // Find by ID
        registry.findById("S001").ifPresent(student ->
            System.out.println("Found: " + student.name())
        );

        // Find by age range
        System.out.println("\nStudents aged 20-22:");
        registry.findByAgeRange(20, 22).forEach(System.out::println);

        // Group by major
        System.out.println("\nStudents by major:");
        registry.groupByMajor().forEach((major, students) -> {
            System.out.println(major + ": " +
                students.stream()
                    .map(Student::name)
                    .collect(Collectors.joining(", "))
            );
        });

        // Average age by major
        System.out.println("\nAverage age by major:");
        registry.averageAgeByMajor().forEach((major, avg) ->
            System.out.printf("%s: %.1f years\n", major, avg)
        );
    }
}
```

**Skills Learned:**

- HashMap for O(1) lookup
- Optional for null-safety
- Stream filtering and sorting
- Collectors.groupingBy

- Method references
- Collection transformations

**Real-World Application:** User management systems, CRM platforms, school/university management.

---

# Exercise 5: Task Priority Queue

**Problem:** Manage tasks with priorities and deadlines.

**Why It Exists:** Understanding natural ordering and priority-based processing.

**Code Example:**

```java
package com.javacourse.collections;

import java.time.LocalDateTime;
import java.util.*;

/**
 * Task with natural ordering by priority then deadline
 */
public record Task(
    String name,
    int priority,
    LocalDateTime deadline,
    String description
) implements Comparable<Task> {

    /**
     * Natural ordering: higher priority first, then earlier deadline
     */
    @Override
    public int compareTo(Task other) {
        return Comparator
            .comparingInt(Task::priority)
            .reversed() // Higher priority first
            .thenComparing(Task::deadline)
            .compare(this, other);
    }
}

/**
 * Task manager using PriorityQueue
 */
class TaskManager {
    private final PriorityQueue<Task> tasks;

    public TaskManager() {
        this.tasks = new PriorityQueue<>();
    }

    /**
     * Add task (automatically sorted)
     */
    public void addTask(Task task) {
        tasks.offer(task);
    }

    /**
     * Get next highest priority task
     */
    public Optional<Task> getNextTask() {
        return Optional.ofNullable(tasks.poll());
    }

    /**
     * Peek at next task without removing
     */
    public Optional<Task> peekNextTask() {
        return Optional.ofNullable(tasks.peek());
    }

    /**
     * Get all tasks sorted by priority
     */
    public List<Task> getAllTasksSorted() {
        return new ArrayList<>(tasks);
    }

    /**
```

```
     * Get tasks by priority level
     */
    public List<Task> getTasksByPriority(int priority) {
        return tasks.stream()
            .filter(t -> t.priority() == priority)
            .toList();
    }

    public int remainingTasks() {
        return tasks.size();
    }
}

// Usage
class TaskDemo {
    public static void main(String[] args) {
        var manager = new TaskManager();

        // Add tasks with different priorities
        manager.addTask(new Task(
            "Fix critical bug",
            1,
            LocalDateTime.now().plusHours(2),
            "Production issue affecting users"
        ));

        manager.addTask(new Task(
            "Code review",
            3,
            LocalDateTime.now().plusDays(1),
            "Review PR #123"
        ));

        manager.addTask(new Task(
            "Update documentation",
            2,
            LocalDateTime.now().plusHours(4),
            "Update API docs"
        ));

        manager.addTask(new Task(
            "Deploy to staging",
            1,
            LocalDateTime.now().plusHours(1),
            "Deploy version 2.1.0"
        ));

        // Process tasks in priority order
        System.out.println("Processing tasks in priority order:\n");
        while (manager.remainingTasks() > 0) {
            manager.getNextTask().ifPresent(task -> {
                System.out.printf(
                    "[Priority %d] %s - Due: %s\n",
                    task.priority(),
                    task.name(),
                    task.deadline()
                );
            });
        }
    }
}
```

**Skills Learned:**

- Comparable interface
- Comparator composition

- PriorityQueue usage
- Time-based data with LocalDateTime
- Queue operations (offer, poll, peek)

**Real-World Application:** Task management systems, job schedulers, event processing, operating system task scheduling.

---

# Exercise 6: LRU Cache Implementation

**Problem:** Implement a cache that evicts least recently used items.

**Why It Exists:** Understanding caching strategies and LinkedHashMap's access ordering.

**Code Example:**

```java
package com.javacourse.collections;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Optional;

/**
 * Least Recently Used (LRU) Cache
 * Demonstrates: LinkedHashMap with access order, generics
 */
public class LRUCache<K, V> {
    private final int capacity;
    private final Map<K, V> cache;

    public LRUCache(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException(
                "Capacity must be positive"
            );
        }
        this.capacity = capacity;

        // LinkedHashMap with access order (true) and custom removal policy
        this.cache = new LinkedHashMap<>(capacity, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
                return size() > capacity;
            }
        };
    }

    /**
     * Get value from cache
     * Returns Optional to handle cache misses
     */
    public Optional<V> get(K key) {
        return Optional.ofNullable(cache.get(key));
    }

    /**
     * Put value in cache
     * Automatically evicts LRU entry if at capacity
     */
    public void put(K key, V value) {
        cache.put(key, value);
    }

    /**
     * Check if key exists in cache
     */
    public boolean containsKey(K key) {
        return cache.containsKey(key);
    }

    /**
     * Get current cache size
     */
    public int size() {
        return cache.size();
    }

    /**
     * Clear all entries
     */
    public void clear() {
        cache.clear();
    }
```

```
    /**
     * Get cache statistics for monitoring
     */
    public CacheStats getStats(int hits, int misses) {
        double hitRate = (double) hits / (hits + misses);
        return new CacheStats(size(), capacity, hitRate);
    }

    public record CacheStats(
        int currentSize,
        int capacity,
        double hitRate
    ) {}
}

// Usage and demonstration
class LRUCacheDemo {
    public static void main(String[] args) {
        var cache = new LRUCache<String, String>(3);

        // Add items
        cache.put("A", "Alice");
        cache.put("B", "Bob");
        cache.put("C", "Charlie");

        System.out.println("Cache size: " + cache.size());

        // Access A (makes it most recently used)
        cache.get("A").ifPresent(v ->
            System.out.println("Found: " + v)
        );

        // Add D (evicts B, the least recently used)
        cache.put("D", "Diana");

        System.out.println("Contains B: " + cache.containsKey("B")); // false
        System.out.println("Contains A: " + cache.containsKey("A")); // true

        // Demonstrate with cache statistics
        int hits = 0;
        int misses = 0;

        if (cache.get("A").isPresent()) hits++; else misses++;
        if (cache.get("B").isPresent()) hits++; else misses++;
        if (cache.get("C").isPresent()) hits++; else misses++;

        var stats = cache.getStats(hits, misses);
        System.out.printf(
            "\nCache Stats: %d/%d entries, %.1f%% hit rate\n",
            stats.currentSize(),
            stats.capacity(),
            stats.hitRate() * 100
        );
    }
}
```

**Skills Learned:**

- LinkedHashMap internals
- Anonymous class with method override
- Generic type parameters
- Capacity-based eviction
- Cache statistics tracking

**Real-World Application:** Application caching, database query caching, web browser cache, CDN systems.

---

# Exercise 7: Word Frequency Counter

**Problem:** Count word occurrences in text efficiently.

**Why It Exists:** Text processing is common. Collectors provide powerful aggregation.

**Code Example:**

```java
package com.javacourse.collections;

import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;

/**
 * Text analyzer demonstrating advanced stream operations
 */
public class TextAnalyzer {

    /**
     * Count word frequencies
     * Demonstrates: grouping and counting
     */
    public Map<String, Long> countWords(String text) {
        return Arrays.stream(text.toLowerCase().split("\\W+"))
            .filter(word -> !word.isEmpty())
            .collect(Collectors.groupingBy(
                Function.identity(),
                Collectors.counting()
            ));
    }

    /**
     * Get top N most frequent words
     */
    public List<WordCount> getTopWords(String text, int n) {
        return countWords(text).entrySet().stream()
            .map(e -> new WordCount(e.getKey(), e.getValue()))
            .sorted(Comparator.comparing(WordCount::count).reversed())
            .limit(n)
            .toList();
    }

    /**
     * Find words longer than specified length
     */
    public Set<String> findLongWords(String text, int minLength) {
        return Arrays.stream(text.toLowerCase().split("\\W+"))
            .filter(word -> word.length() >= minLength)
            .collect(Collectors.toSet());
    }

    /**
     * Calculate text statistics
     */
    public TextStats calculateStats(String text) {
        var words = Arrays.stream(text.split("\\W+"))
            .filter(word -> !word.isEmpty())
            .toList();

        int totalWords = words.size();
        long uniqueWords = words.stream().distinct().count();
        double avgLength = words.stream()
            .mapToInt(String::length)
            .average()
            .orElse(0.0);

        return new TextStats(totalWords, uniqueWords, avgLength);
    }

    public record WordCount(String word, long count) {}

    public record TextStats(
        int totalWords,
        long uniqueWords,
```

```
        double averageWordLength
    ) {}
}

// Usage
class TextAnalyzerDemo {
    public static void main(String[] args) {
        var analyzer = new TextAnalyzer();

        String text = """
            Java is a powerful programming language.
            Java enables developers to write robust applications.
            Programming in Java is enjoyable and productive.
            Java continues to evolve with modern features.
            """;

        // Word frequencies
        System.out.println("Word Frequencies:");
        var frequencies = analyzer.countWords(text);
        frequencies.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
            .limit(10)
            .forEach(e -> System.out.printf("%s: %d\n", e.getKey(), e.getValue()));

        // Top words
        System.out.println("\nTop 5 words:");
        analyzer.getTopWords(text, 5).forEach(wc ->
            System.out.printf("%s (%d)\n", wc.word(), wc.count())
        );

        // Long words
        System.out.println("\nWords with 8+ characters:");
        analyzer.findLongWords(text, 8).forEach(System.out::println);

        // Statistics
        var stats = analyzer.calculateStats(text);
        System.out.printf("""

            Text Statistics:
            - Total words: %d
            - Unique words: %d
            - Average word length: %.1f characters
            """,
            stats.totalWords(),
            stats.uniqueWords(),
            stats.averageWordLength()
        );
    }
}
```

**Skills Learned:**

- Collectors.groupingBy with downstream collectors
- Function.identity() for grouping by element itself
- Text blocks (""")
- Statistical operations on streams
- Comparator chaining

**Real-World Application:** Search engines, text analytics, sentiment analysis, document processing, SEO tools.

---

# Module 3: Algorithms & Problem Solving

**Duration:** 2 Weeks

## Concepts Covered

- Algorithm complexity (Big O notation basics)
- Recursion vs iteration trade-offs
- Divide and conquer strategy
- Memoization for optimization
- When to use standard library vs custom implementation

## Exercise 8: Binary Search Implementation

**Problem:** Find element in sorted array efficiently.

**Why It Exists:** O(log n) vs O(n) makes a huge difference with large datasets.

**Code Example:**

```java
package com.javacourse.algorithms;

import java.util.*;

/**
 * Binary search demonstrating logarithmic time complexity
 */
public class BinarySearch {

    /**
     * Iterative binary search - O(log n) time, O(1) space
     * Returns Optional to handle element not found
     */
    public static <T extends Comparable<T>> Optional<Integer> search(
        List<T> sorted,
        T target) {

        int left = 0;
        int right = sorted.size() - 1;

        while (left <= right) {
            // Prevent overflow: use left + (right - left) / 2
            int mid = left + (right - left) / 2;
            int comparison = sorted.get(mid).compareTo(target);

            if (comparison == 0) {
                return Optional.of(mid);
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return Optional.empty();
    }

    /**
     * Recursive binary search - O(log n) time, O(log n) space
     * Demonstrates: recursion with explicit parameters
     */
    public static <T extends Comparable<T>> Optional<Integer> searchRecursive(
        List<T> sorted,
        T target) {
        return searchRecursiveHelper(sorted, target, 0, sorted.size() - 1);
    }

    private static <T extends Comparable<T>> Optional<Integer> searchRecursiveHelper(
        List<T> sorted,
        T target,
        int left,
        int right) {

        if (left > right) {
            return Optional.empty();
        }

        int mid = left + (right - left) / 2;
        int comparison = sorted.get(mid).compareTo(target);

        if (comparison == 0) {
            return Optional.of(mid);
        } else if (comparison < 0) {
            return searchRecursiveHelper(sorted, target, mid + 1, right);
        } else {
            return searchRecursiveHelper(sorted, target, left, mid - 1);
        }
```

```
    }

    /**
     * Find first occurrence (handles duplicates)
     */
    public static <T extends Comparable<T>> Optional<Integer> searchFirst(
        List<T> sorted,
        T target) {

        int left = 0;
        int right = sorted.size() - 1;
        int result = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison = sorted.get(mid).compareTo(target);

            if (comparison == 0) {
                result = mid;
                right = mid - 1; // Continue searching left
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return result == -1 ? Optional.empty() : Optional.of(result);
    }
}

// Demonstration and benchmarking
class BinarySearchDemo {
    public static void main(String[] args) {
        var numbers = List.of(1, 3, 5, 7, 9, 11, 13, 15, 17, 19);

        // Basic search
        BinarySearch.search(numbers, 7).ifPresent(index ->
            System.out.println("Found 7 at index: " + index)
        );

        // Element not found
        var notFound = BinarySearch.search(numbers, 8);
        System.out.println("Found 8: " + notFound.isPresent());

        // Compare with linear search performance
        var largeList = new ArrayList<Integer>();
        for (int i = 0; i < 1_000_000; i++) {
            largeList.add(i * 2);
        }

        // Binary search
        long start = System.nanoTime();
        BinarySearch.search(largeList, 999_998);
        long binaryTime = System.nanoTime() - start;

        // Linear search
        start = System.nanoTime();
        largeList.indexOf(999_998);
        long linearTime = System.nanoTime() - start;

        System.out.printf("""

            Performance comparison (1M elements):
            Binary search: %,d ns
            Linear search: %,d ns
            Binary is %.1fx faster
            """,
```

```
                binaryTime,
                linearTime,
                (double) linearTime / binaryTime
            );
        }
    }
```

**Skills Learned:**

- Binary search algorithm
- Iterative vs recursive approaches
- Preventing integer overflow
- Performance benchmarking
- Big O complexity in practice

**Real-World Application:** Database indexing, phone books, dictionary lookups, any sorted data search.

---

# Exercise 9: Fibonacci Calculator (Multiple Approaches)

**Problem:** Calculate Fibonacci numbers efficiently.

**Why It Exists:** Understanding recursion, memoization, and performance trade-offs.

**Code Example:**

```java
package com.javacourse.algorithms;

import java.util.*;
import java.util.stream.Stream;

/**
 * Fibonacci calculator demonstrating different approaches
 */
public class FibonacciCalculator {

    /**
     * Naive recursive - O(2^n) time - EDUCATIONAL ONLY
     * Demonstrates: exponential time complexity problem
     */
    public long fibRecursive(int n) {
        if (n <= 1) return n;
        return fibRecursive(n - 1) + fibRecursive(n - 2);
    }

    /**
     * Memoized recursive - O(n) time, O(n) space
     * Demonstrates: caching for performance
     */
    private final Map<Integer, Long> memo = new HashMap<>();

    public long fibMemoized(int n) {
        if (n <= 1) return n;

        return memo.computeIfAbsent(n, k ->
            fibMemoized(k - 1) + fibMemoized(k - 2)
        );
    }

    /**
     * Iterative - O(n) time, O(1) space - BEST PERFORMANCE
     * Demonstrates: space-efficient approach
     */
    public long fibIterative(int n) {
        if (n <= 1) return n;

        long prev = 0;
        long curr = 1;

        for (int i = 2; i <= n; i++) {
            long next = prev + curr;
            prev = curr;
            curr = next;
        }

        return curr;
    }

    /**
     * Stream-based - O(n) time, functional approach
     * Demonstrates: modern Java streams
     */
    public long fibStream(int n) {
        return Stream.iterate(
                new long[]{0, 1},
                f -> new long[]{f[1], f[0] + f[1]}
            )
            .limit(n + 1)
            .map(f -> f[0])
            .reduce((a, b) -> b)
            .orElse(0L);
    }
```

```java
    /**
     * Get Fibonacci sequence up to n
     */
    public List<Long> fibSequence(int n) {
        var sequence = new ArrayList<Long>();
        for (int i = 0; i <= n; i++) {
            sequence.add(fibIterative(i));
        }
        return sequence;
    }

    /**
     * Performance comparison
     */
    public record PerformanceResult(
        String method,
        long result,
        long timeNanos
    ) {}

    public List<PerformanceResult> benchmark(int n) {
        var results = new ArrayList<PerformanceResult>();

        // Iterative
        long start = System.nanoTime();
        long resultIterative = fibIterative(n);
        results.add(new PerformanceResult(
            "Iterative",
            resultIterative,
            System.nanoTime() - start
        ));

        // Memoized
        memo.clear();
        start = System.nanoTime();
        long resultMemoized = fibMemoized(n);
        results.add(new PerformanceResult(
            "Memoized",
            resultMemoized,
            System.nanoTime() - start
        ));

        // Stream
        start = System.nanoTime();
        long resultStream = fibStream(n);
        results.add(new PerformanceResult(
            "Stream",
            resultStream,
            System.nanoTime() - start
        ));

        // Recursive (only for small n)
        if (n <= 35) {
            start = System.nanoTime();
            long resultRecursive = fibRecursive(n);
            results.add(new PerformanceResult(
                "Recursive",
                resultRecursive,
                System.nanoTime() - start
            ));
        }

        return results;
    }
}

// Usage and benchmarking
class FibonacciDemo {
```

```
    public static void main(String[] args) {
        var calc = new FibonacciCalculator();

        // Calculate some Fibonacci numbers
        System.out.println("First 10 Fibonacci numbers:");
        calc.fibSequence(10).forEach(System.out::println);

        // Performance comparison
        int n = 40;
        System.out.printf("\nPerformance comparison for F(%d):\n\n", n);

        var results = calc.benchmark(n);
        results.forEach(r -> System.out.printf(
            "%-12s: %,15d (took %,d ns)\n",
            r.method(),
            r.result(),
            r.timeNanos()
        ));

        // Show exponential growth problem with recursive
        System.out.println("\nWhy recursive is slow (time in ms):");
        for (int i = 30; i <= 40; i++) {
            long start = System.currentTimeMillis();
            calc.fibRecursive(i);
            long time = System.currentTimeMillis() - start;
            System.out.printf("F(%d): %d ms\n", i, time);
        }
    }
}
```

**Skills Learned:**

- Recursion and its pitfalls
- Memoization pattern
- computeIfAbsent for caching
- Iterative optimization
- Stream.iterate for sequences
- Performance measurement

**Real-World Application:** Understanding caching strategies, optimization techniques, algorithm analysis.

---

# Exercise 10: Anagram Grouper

**Problem:** Group words that are anagrams of each other.

**Why It Exists:** Hash-based algorithms, custom grouping logic.

**Code Example:**

```java
package com.javacourse.algorithms;

import java.util.*;
import java.util.stream.Collectors;

/**
 * Anagram grouper demonstrating hashing and grouping
 */
public class AnagramGrouper {

    /**
     * Group anagrams together
     * Uses sorted characters as key - O(n * m log m) where n = words, m = avg length
     */
    public Map<String, List<String>> groupAnagrams(List<String> words) {
        return words.stream()
            .collect(Collectors.groupingBy(this::getSignature));
    }

    /**
     * Get signature (sorted characters) for a word
     */
    private String getSignature(String word) {
        char[] chars = word.toLowerCase().toCharArray();
        Arrays.sort(chars);
        return new String(chars);
    }

    /**
     * Alternative: character frequency as signature
     * Avoids sorting but uses more space
     */
    public Map<String, List<String>> groupAnagramsFrequency(List<String> words) {
        return words.stream()
            .collect(Collectors.groupingBy(this::getFrequencySignature));
    }

    private String getFrequencySignature(String word) {
        int[] freq = new int[26];
        for (char c : word.toLowerCase().toCharArray()) {
            if (c >= 'a' && c <= 'z') {
                freq[c - 'a']++;
            }
        }
        return Arrays.toString(freq);
    }

    /**
     * Find all anagrams of a specific word
     */
    public List<String> findAnagrams(List<String> words, String target) {
        String targetSig = getSignature(target);
        return words.stream()
            .filter(word -> getSignature(word).equals(targetSig))
            .filter(word -> !word.equalsIgnoreCase(target))
            .toList();
    }

    /**
     * Get largest anagram group
     */
    public Optional<List<String>> getLargestGroup(List<String> words) {
        return groupAnagrams(words).values().stream()
            .max(Comparator.comparing(List::size));
    }

    /**
```

```java
     * Count total anagram groups
     */
    public long countGroups(List<String> words) {
        return groupAnagrams(words).size();
    }
}

// Usage
class AnagramDemo {
    public static void main(String[] args) {
        var grouper = new AnagramGrouper();

        var words = List.of(
            "listen", "silent", "enlist",
            "hello", "world",
            "evil", "vile", "live",
            "a gentleman", "elegant man",
            "the eyes", "they see"
        );

        // Group all anagrams
        System.out.println("Anagram groups:");
        var groups = grouper.groupAnagrams(words);
        groups.values().stream()
            .filter(group -> group.size() > 1)
            .forEach(group -> System.out.println("- " + group));

        // Find anagrams of specific word
        System.out.println("\nAnagrams of 'listen':");
        grouper.findAnagrams(words, "listen")
            .forEach(w -> System.out.println("- " + w));

        // Largest group
        grouper.getLargestGroup(words).ifPresent(group ->
            System.out.println("\nLargest anagram group: " + group)
        );

        // Statistics
        System.out.printf(
            "\nTotal words: %d, Unique groups: %d\n",
            words.size(),
            grouper.countGroups(words)
        );

        // Compare performance of both methods
        var manyWords = generateRandomWords(10000);

        long start = System.nanoTime();
        grouper.groupAnagrams(manyWords);
        long sortTime = System.nanoTime() - start;

        start = System.nanoTime();
        grouper.groupAnagramsFrequency(manyWords);
        long freqTime = System.nanoTime() - start;

        System.out.printf("""

            Performance (10k words):
            Sorting method: %,d ns
            Frequency method: %,d ns
            """,
            sortTime,
            freqTime
        );
    }

    private static List<String> generateRandomWords(int count) {
        var random = new Random();
```

```
        var words = new ArrayList<String>();
        for (int i = 0; i < count; i++) {
            int length = 5 + random.nextInt(5);
            var sb = new StringBuilder();
            for (int j = 0; j < length; j++) {
                sb.append((char) ('a' + random.nextInt(26)));
            }
            words.add(sb.toString());
        }
        return words;
    }
}
```

**Skills Learned:**

- Custom hashing/signature generation
- Collectors.groupingBy with custom key
- Character frequency counting
- Algorithm comparison
- Stream filtering with multiple conditions

**Real-World Application:** Word games, spell checkers, data deduplication, plagiarism detection.

---

# Exercise 11: Merge Sorted Lists

**Problem:** Merge two sorted lists maintaining order.

**Why It Exists:** Fundamental to merge sort, demonstrates two-pointer technique.

**Code Example:**

```java
package com.javacourse.algorithms;

import java.util.*;

/**
 * Merge sorted lists demonstrating two-pointer technique
 */
public class ListMerger {

    /**
     * Merge two sorted lists - O(n + m) time, O(n + m) space
     * Demonstrates: two-pointer technique
     */
    public static <T extends Comparable<T>> List<T> mergeSorted(
        List<T> list1,
        List<T> list2) {

        List<T> result = new ArrayList<>(list1.size() + list2.size());
        int i = 0, j = 0;

        // Merge while both lists have elements
        while (i < list1.size() && j < list2.size()) {
            if (list1.get(i).compareTo(list2.get(j)) <= 0) {
                result.add(list1.get(i++));
            } else {
                result.add(list2.get(j++));
            }
        }

        // Add remaining elements from list1
        while (i < list1.size()) {
            result.add(list1.get(i++));
        }

        // Add remaining elements from list2
        while (j < list2.size()) {
            result.add(list2.get(j++));
        }

        return result;
    }

    /**
     * Merge multiple sorted lists
     * Uses priority queue for efficiency - O(n log k) where k = number of lists
     */
    public static <T extends Comparable<T>> List<T> mergeMultiple(
        List<List<T>> sortedLists) {

        // Record to track which list and position
        record Element<T>(T value, int listIndex, int elementIndex)
            implements Comparable<Element<T>> {

            @Override
            public int compareTo(Element<T> other) {
                return ((Comparable<T>) value).compareTo(other.value);
            }
        }

        var pq = new PriorityQueue<Element<T>>();
        var result = new ArrayList<T>();

        // Add first element from each list
        for (int i = 0; i < sortedLists.size(); i++) {
            if (!sortedLists.get(i).isEmpty()) {
                pq.offer(new Element<>(
                    sortedLists.get(i).get(0),
```

```
                    i,
                    0
                ));
            }
        }

        // Process elements
        while (!pq.isEmpty()) {
            var elem = pq.poll();
            result.add(elem.value);

            // Add next element from same list
            int nextIndex = elem.elementIndex + 1;
            var sourceList = sortedLists.get(elem.listIndex);
            if (nextIndex < sourceList.size()) {
                pq.offer(new Element<>(
                    sourceList.get(nextIndex),
                    elem.listIndex,
                    nextIndex
                ));
            }
        }

        return result;
    }

    /**
     * Check if list is sorted
     */
    public static <T extends Comparable<T>> boolean isSorted(List<T> list) {
        for (int i = 1; i < list.size(); i++) {
            if (list.get(i).compareTo(list.get(i - 1)) < 0) {
                return false;
            }
        }
        return true;
    }

    /**
     * Find median of two sorted lists
     * Demonstrates: binary search on merged result
     */
    public static <T extends Comparable<T>> Optional<T> findMedian(
        List<T> list1,
        List<T> list2) {

        var merged = mergeSorted(list1, list2);
        if (merged.isEmpty()) {
            return Optional.empty();
        }

        int mid = merged.size() / 2;
        return Optional.of(merged.get(mid));
    }
}

// Usage
class ListMergerDemo {
    public static void main(String[] args) {
        // Merge two lists
        var list1 = List.of(1, 3, 5, 7, 9);
        var list2 = List.of(2, 4, 6, 8, 10);

        var merged = ListMerger.mergeSorted(list1, list2);
        System.out.println("Merged: " + merged);
        System.out.println("Is sorted: " + ListMerger.isSorted(merged));

        // Merge multiple lists
```

```
        var lists = List.of(
            List.of(1, 4, 7),
            List.of(2, 5, 8),
            List.of(3, 6, 9)
        );

        var mergedMultiple = ListMerger.mergeMultiple(lists);
        System.out.println("Merged multiple: " + mergedMultiple);

        // Find median
        ListMerger.findMedian(list1, list2).ifPresent(median ->
            System.out.println("Median: " + median)
        );

        // Performance comparison
        var large1 = new ArrayList<Integer>();
        var large2 = new ArrayList<Integer>();
        for (int i = 0; i < 100000; i++) {
            large1.add(i * 2);
            large2.add(i * 2 + 1);
        }

        long start = System.nanoTime();
        ListMerger.mergeSorted(large1, large2);
        long time = System.nanoTime() - start;

        System.out.printf(
            "\nMerged 200k elements in %,d ns (%.2f ms)\n",
            time,
            time / 1_000_000.0
        );
    }
}
```

**Skills Learned:**

- Two-pointer technique
- Priority queue for k-way merge
- Local records in methods
- Generic algorithms
- Performance analysis

**Real-World Application:** Merge sort implementation, database join operations, log file merging, time-series data merging.

---

# Module 4: Object-Oriented Programming & Design

**Duration:** 2 Weeks

## Concepts Covered

- When to use classes vs records vs sealed types
- Encapsulation done right (immutability preferred)
- Composition over inheritance
- Interface segregation
- Polymorphism with sealed types

# Exercise 12: Bank Account System

**Problem:** Model different account types with business rules.

**Why It Exists:** State management, encapsulation, domain modeling with types.

**Code Example:**

```java
package com.javacourse.oop;

import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

/**
 * Bank account system demonstrating sealed types and immutability
 */
public class BankingSystem {

    /**
     * Sealed interface ensures all account types are known
     */
    public sealed interface Account
        permits CheckingAccount, SavingsAccount, InvestmentAccount {

        String accountNumber();
        BigDecimal balance();
        LocalDateTime createdAt();

        Account deposit(BigDecimal amount);
        Account withdraw(BigDecimal amount);

        default String getType() {
            return switch (this) {
                case CheckingAccount c -> "Checking";
                case SavingsAccount s -> "Savings";
                case InvestmentAccount i -> "Investment";
            };
        }
    }

    /**
     * Checking account with overdraft protection
     */
    public record CheckingAccount(
        String accountNumber,
        BigDecimal balance,
        BigDecimal overdraftLimit,
        LocalDateTime createdAt
    ) implements Account {

        public CheckingAccount {
            if (overdraftLimit.compareTo(BigDecimal.ZERO) < 0) {
                throw new IllegalArgumentException(
                    "Overdraft limit cannot be negative"
                );
            }
        }

        @Override
        public Account deposit(BigDecimal amount) {
            validatePositiveAmount(amount);
            return new CheckingAccount(
                accountNumber,
                balance.add(amount),
                overdraftLimit,
                createdAt
            );
        }

        @Override
        public Account withdraw(BigDecimal amount) {
            validatePositiveAmount(amount);
            var newBalance = balance.subtract(amount);
```

```java
            // Check overdraft limit
            if (newBalance.add(overdraftLimit).compareTo(BigDecimal.ZERO) < 0) {
                throw new InsufficientFundsException(
                    "Withdrawal exceeds overdraft limit"
                );
            }

            return new CheckingAccount(
                accountNumber,
                newBalance,
                overdraftLimit,
                createdAt
            );
        }
    }

    /**
     * Savings account with interest rate
     */
    public record SavingsAccount(
        String accountNumber,
        BigDecimal balance,
        BigDecimal interestRate,
        int withdrawalsThisMonth,
        LocalDateTime createdAt
    ) implements Account {

        private static final int MAX_FREE_WITHDRAWALS = 6;
        private static final BigDecimal WITHDRAWAL_FEE = new BigDecimal("5.00");

        @Override
        public Account deposit(BigDecimal amount) {
            validatePositiveAmount(amount);
            return new SavingsAccount(
                accountNumber,
                balance.add(amount),
                interestRate,
                withdrawalsThisMonth,
                createdAt
            );
        }

        @Override
        public Account withdraw(BigDecimal amount) {
            validatePositiveAmount(amount);

            // Apply fee if over limit
            var totalAmount = amount;
            if (withdrawalsThisMonth >= MAX_FREE_WITHDRAWALS) {
                totalAmount = totalAmount.add(WITHDRAWAL_FEE);
            }

            if (balance.compareTo(totalAmount) < 0) {
                throw new InsufficientFundsException("Insufficient funds");
            }

            return new SavingsAccount(
                accountNumber,
                balance.subtract(totalAmount),
                interestRate,
                withdrawalsThisMonth + 1,
                createdAt
            );
        }

        /**
         * Calculate and apply interest
```

```
         */
        public SavingsAccount applyInterest() {
            var interest = balance.multiply(interestRate);
            return new SavingsAccount(
                accountNumber,
                balance.add(interest),
                interestRate,
                withdrawalsThisMonth,
                createdAt
            );
        }

        /**
         * Reset monthly withdrawal counter
         */
        public SavingsAccount resetMonthlyWithdrawals() {
            return new SavingsAccount(
                accountNumber,
                balance,
                interestRate,
                0,
                createdAt
            );
        }
    }

    /**
     * Investment account with risk level
     */
    public record InvestmentAccount(
        String accountNumber,
        BigDecimal balance,
        RiskLevel riskLevel,
        LocalDateTime createdAt
    ) implements Account {

        public enum RiskLevel {
            LOW, MODERATE, HIGH, AGGRESSIVE
        }

        @Override
        public Account deposit(BigDecimal amount) {
            validatePositiveAmount(amount);
            return new InvestmentAccount(
                accountNumber,
                balance.add(amount),
                riskLevel,
                createdAt
            );
        }

        @Override
        public Account withdraw(BigDecimal amount) {
            validatePositiveAmount(amount);

            if (balance.compareTo(amount) < 0) {
                throw new InsufficientFundsException("Insufficient funds");
            }

            return new InvestmentAccount(
                accountNumber,
                balance.subtract(amount),
                riskLevel,
                createdAt
            );
        }

        /**
```

```
         * Simulate investment returns based on risk level
         */
        public InvestmentAccount applyReturns(BigDecimal returnRate) {
            var returns = balance.multiply(returnRate);
            return new InvestmentAccount(
                accountNumber,
                balance.add(returns),
                riskLevel,
                createdAt
            );
        }
    }

    /**
     * Transaction record for audit trail
     */
    public record Transaction(
        String accountNumber,
        TransactionType type,
        BigDecimal amount,
        LocalDateTime timestamp,
        BigDecimal balanceAfter
    ) {
        public enum TransactionType {
            DEPOSIT, WITHDRAWAL, INTEREST, TRANSFER
        }
    }

    /**
     * Account manager orchestrating operations
     */
    public static class AccountManager {
        private final List<Transaction> transactionHistory = new ArrayList<>();

        /**
         * Execute deposit and record transaction
         */
        public Account executeDeposit(Account account, BigDecimal amount) {
            var updated = account.deposit(amount);
            recordTransaction(
                account.accountNumber(),
                Transaction.TransactionType.DEPOSIT,
                amount,
                updated.balance()
            );
            return updated;
        }

        /**
         * Execute withdrawal and record transaction
         */
        public Account executeWithdrawal(Account account, BigDecimal amount) {
            var updated = account.withdraw(amount);
            recordTransaction(
                account.accountNumber(),
                Transaction.TransactionType.WITHDRAWAL,
                amount,
                updated.balance()
            );
            return updated;
        }

        /**
         * Transfer between accounts
         */
        public TransferResult transfer(
            Account from,
            Account to,
```

```
            BigDecimal amount) {

            try {
                var fromUpdated = from.withdraw(amount);
                var toUpdated = to.deposit(amount);

                recordTransaction(
                    from.accountNumber(),
                    Transaction.TransactionType.TRANSFER,
                    amount.negate(),
                    fromUpdated.balance()
                );
                recordTransaction(
                    to.accountNumber(),
                    Transaction.TransactionType.TRANSFER,
                    amount,
                    toUpdated.balance()
                );

                return new TransferResult(fromUpdated, toUpdated, true, null);
            } catch (InsufficientFundsException e) {
                return new TransferResult(from, to, false, e.getMessage());
            }
        }

        private void recordTransaction(
            String accountNumber,
            Transaction.TransactionType type,
            BigDecimal amount,
            BigDecimal balanceAfter) {

            transactionHistory.add(new Transaction(
                accountNumber,
                type,
                amount,
                LocalDateTime.now(),
                balanceAfter
            ));
        }

        /**
         * Get transaction history for account
         */
        public List<Transaction> getHistory(String accountNumber) {
            return transactionHistory.stream()
                .filter(t -> t.accountNumber().equals(accountNumber))
                .toList();
        }
    }

    public record TransferResult(
        Account fromAccount,
        Account toAccount,
        boolean success,
        String errorMessage
    ) {}

    /**
     * Custom exception for insufficient funds
     */
    public static class InsufficientFundsException extends RuntimeException {
        public InsufficientFundsException(String message) {
            super(message);
        }
    }

    private static void validatePositiveAmount(BigDecimal amount) {
        if (amount.compareTo(BigDecimal.ZERO) <= 0) {
```

```
                throw new IllegalArgumentException("Amount must be positive");
            }
        }
    }

    // Usage demonstration
    class BankingDemo {
        public static void main(String[] args) {
            var manager = new BankingSystem.AccountManager();

            // Create accounts
            var checking = new BankingSystem.CheckingAccount(
                "CHK-001",
                new BigDecimal("1000.00"),
                new BigDecimal("500.00"), // Overdraft limit
                LocalDateTime.now()
            );

            var savings = new BankingSystem.SavingsAccount(
                "SAV-001",
                new BigDecimal("5000.00"),
                new BigDecimal("0.02"), // 2% interest
                0,
                LocalDateTime.now()
            );

            // Perform operations
            System.out.println("Initial balances:");
            System.out.println("Checking: $" + checking.balance());
            System.out.println("Savings: $" + savings.balance());

            // Deposit
            checking = (BankingSystem.CheckingAccount) manager.executeDeposit(
                checking,
                new BigDecimal("500.00")
            );
            System.out.println("\nAfter deposit: $" + checking.balance());

            // Withdraw
            checking = (BankingSystem.CheckingAccount) manager.executeWithdrawal(
                checking,
                new BigDecimal("200.00")
            );
            System.out.println("After withdrawal: $" + checking.balance());

            // Transfer
            var transferResult = manager.transfer(
                checking,
                savings,
                new BigDecimal("300.00")
            );

            if (transferResult.success()) {
                checking = (BankingSystem.CheckingAccount) transferResult.fromAccount();
                savings = (BankingSystem.SavingsAccount) transferResult.toAccount();
                System.out.println("\nAfter transfer:");
                System.out.println("Checking: $" + checking.balance());
                System.out.println("Savings: $" + savings.balance());
            }

            // Apply interest to savings
            savings = savings.applyInterest();
            System.out.println("\nAfter interest: $" + savings.balance());

            // Transaction history
            System.out.println("\nTransaction history for " + checking.accountNumber() + ":");
            manager.getHistory(checking.accountNumber()).forEach(t ->
                System.out.printf(
```

```
                    "%s: %s $%s (Balance: $%s)\n",
                    t.timestamp().toLocalTime(),
                    t.type(),
                    t.amount(),
                    t.balanceAfter()
                )
            );
        }
    }
```

**Skills Learned:**
- Sealed interfaces for closed hierarchies
- Immutable domain modeling
- Composition with manager class
- Custom exceptions
- Transaction pattern
- Pattern matching in default methods
- Business rule enforcement through types

**Real-World Application:** Banking systems, financial software, e-wallet applications, payment processing.

---

(Due to length constraints, I'll continue with a summary structure for the remaining modules. Let me know if you'd like any specific module expanded in full detail like the above.)

---

# Module 5: Functional Programming in Java

**Duration:** 2 Weeks

## Key Exercises

**Exercise 15: Data Transformation Pipeline**
- Advanced stream operations
- Collectors with downstream processing
- Sales data analysis

**Exercise 16: Validation Framework**
- Functional composition
- Validator chaining
- Result types

**Exercise 17: Retry Logic**
- Higher-order functions
- Resilience patterns
- Supplier usage

---

# Module 6: Concurrency & Asynchronous Programming

**Duration:** 2 Weeks

## Key Exercises

**Exercise 18: Parallel Data Processor**
- Parallel streams
- Performance benchmarking
- Thread safety with immutability

**Exercise 19: Async API Aggregator**
- CompletableFuture composition
- Async data fetching
- Error handling

**Exercise 20: Rate Limiter**
- Semaphore usage
- Resource management
- Thread coordination

---

# Module 7: Design Patterns (Modern Approach)

**Duration:** 2 Weeks

## Key Exercises

**Exercise 21: Strategy Pattern (Functional)**
- Functions as strategies
- Payment processing
- Plugin architecture

**Exercise 22: Builder Pattern with Records**
- Fluent APIs
- Optional parameters
- Validation

**Exercise 23: Observer Pattern (Reactive)**
- EventBus implementation
- Pub/sub messaging
- Decoupled components

---

# Module 8: Error Handling & Resilience

**Duration:** 2 Weeks

## Key Exercises

**Exercise 24: Result Type**
- Railway-oriented programming
- Explicit error handling
- Monadic operations

**Exercise 25: Circuit Breaker**
- Fault tolerance

- State machines
- Cascading failure prevention

---

# Module 9: Testing & Quality

**Duration:** 1 Week

## Key Exercises

**Exercise 26: Test-Driven Calculator**
- TDD workflow
- JUnit 5
- Assertions

**Exercise 27: Parameterized Testing**
- Data-driven tests
- Multiple scenarios
- Test organization

---

# Module 10: Real-World Application

**Duration:** 3 Weeks

## Capstone Project: RESTful Task Management API

**Full-Stack Implementation:**
- Domain layer with sealed types
- Repository abstraction
- Service layer with business logic
- REST API controllers
- Event-driven architecture
- Async operations

- Error handling with Result types
- Comprehensive testing

**Technologies:**
- Java 21
- Spring Boot (optional)
- H2/PostgreSQL
- JUnit 5

---

# Course Timeline

## 20-Week Program

| Weeks | Module | Focus |
|---|---|---|
| 1-2 | Module 1 | Java Fundamentals |
| 3-4 | Module 2 | Collections & Data Structures |
| 5-6 | Module 3 | Algorithms |
| 7-8 | Module 4 | OOP & Design |
| 9-10 | Module 5 | Functional Programming |
| 11-12 | Module 6 | Concurrency |
| 13-14 | Module 7 | Design Patterns |
| 15-16 | Module 8 | Error Handling |
| 17 | Module 9 | Testing |
| 18-20 | Module 10 | Capstone Project |

---

# Teaching Methodology

## For Each Exercise

**1. Problem Introduction (15 minutes)**
- Real-world scenario
- Why this problem matters
- Common use cases

**2. Naive Approach (20 minutes)**
- What beginners try first
- Live coding demonstration
- Identify problems

**3. Modern Java Solution (45 minutes)**
- Introduce modern features
- Step-by-step implementation
- Code walkthrough

**4. Trade-offs Discussion (20 minutes)**
- When to use this approach
- When NOT to use it
- Alternative solutions

**5. Hands-On Practice (60 minutes)**
- Students implement variation
- Pair programming
- Code review

**6. Real-World Context (15 minutes)**
- Industry examples
- Production considerations
- Best practices

## Assessment Methods

**Weekly:**
- Coding exercises
- Code reviews
- Pair programming sessions

**Module End:**
- Mini-project applying concepts

- Peer code review
- Technical presentation

**Final:**
- Capstone project
- Code quality assessment
- Technical interview simulation

---

# Additional Resources

## Per Module

- Complete code examples and starter templates
- Unit tests students must pass
- Code review checklist
- "Professional vs Educational" comparison
- Performance benchmarks where relevant

## Community & Support

- Online Q&A forum
- Weekly office hours
- Peer study groups
- Code review sessions

## Career Preparation

- Resume building with projects
- Mock interviews
- Open source contribution guidance
- Job search strategies

---

# Success Metrics

Students completing this course will:
- ✅ Write production-ready Java 21 code
- ✅ Understand modern Java paradigms (OOP, FP, DOP)
- ✅ Build scalable applications
- ✅ Handle errors gracefully
- ✅ Write comprehensive tests
- ✅ Make informed technical decisions
- ✅ Be ready for junior/mid-level Java positions

---

# Next Steps

1. **Set up development environment**
   - Install Java 21
   - IDE setup (IntelliJ IDEA recommended)
   - Git basics

2. **Connect with peers**
   - Join course community
   - Form study groups
   - Participate in code reviews

3. **Start Module 1**
   - Review fundamentals
   - Complete Exercise 1
   - Begin building portfolio

---

*This course is continuously updated to reflect the latest Java releases and industry best practices.*