C# Advanced Programming - Practice Exercises

Learning Through Practice

“Programming is learned by writing programs” - these exercises are designed to give you hands-on experience with advanced C# concepts. Each exercise builds your understanding and coding skills progressively.

How to Use These Exercises:

1. **Start with Exercise 1** for each topic
2. **Build incrementally** - each exercise adds complexity
3. **Test your code** frequently as you develop
4. **Reference the lecture examples** when you need guidance
5. **Challenge yourself** with the bonus exercises

Topic 1: Delegates

Exercise 1.1: Basic Calculator with Delegates

Create a calculator that uses delegates to perform operations.

Learning Objectives: - Understand delegate declaration and usage - Practice method-to-delegate assignment - Experience passing delegates as parameters

Requirements: - Define a delegate `MathOperation` that takes two `double` parameters and returns a `double` - Create methods for Add, Subtract, Multiply, and Divide - Write a `Calculate` method that takes two numbers and a delegate - Test all operations with sample data

Implementation Hints:

```
// Start with this structure:
public delegate double MathOperation(double a, double b);

public static double Calculate(double x, double y, MathOperation operation)
{
    // Your implementation here
}
```

Example Output:

```
5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
5 / 3 = 1.67
```

Extension Ideas: - Add more operations (power, modulus, square root) - Handle division by zero gracefully - Create a calculator history using delegates

Exercise 1.2: Event System for Game

Create a simple event system for a game using multicast delegates.

Learning Objectives: - Master multicast delegate concepts - Understand event-driven programming patterns - Practice delegate composition and invocation

Requirements: - Define a delegate `GameEvent` that takes a string message - Create a `Player` class with events for: LevelUp, Die, CollectItem - Create `Logger`, `UI`, and `SoundSystem` classes that subscribe to these events - Simulate a player leveling up, dying, and collecting items - Show how multiple systems respond to the same event

Implementation Hints:

```
public class Player
{
    public GameEvent OnLevelUp;
    public GameEvent OnDie;
    public GameEvent OnCollectItem;

    public void LevelUp()
    {
        // Trigger event and notify all subscribers
        OnLevelUp?.Invoke($"Player reached level {currentLevel}!");
    }
}
```

Expected Behavior: When a player levels up, all three systems should respond: - Logger: writes to console/file - UI: updates display elements - SoundSystem: plays appropriate sounds

Extension Ideas: - Add event priorities (some handlers execute first) - Implement event cancellation - Create event history/replay system

Exercise 1.3: Data Processing Pipeline

Build a data processing pipeline using delegates.

Requirements: - Define a delegate `DataProcessor<T>` that takes and returns a generic type T - Create a `Pipeline<T>` class that can chain multiple processing steps - Implement processors for: ToUpper, Trim, RemoveNumbers (for strings) - Process a list of messy string data through your pipeline

Exercise 1.4: Custom Filter System

Create a flexible filtering system using predicates.

Requirements: - Create a `Product` class with properties: Name, Price, Category, InStock - Write a `FilterProducts` method that takes a list of products and a predicate - Create various filter predicates: ByPrice, ByCategory, InStockOnly - Combine multiple filters using delegate composition (AND, OR operations)

Exercise 1.5: Asynchronous Callback System

Implement a simple asynchronous operation system with callbacks.

Requirements: - Define delegates for: `OnSuccess<T>`, `OnError`, `OnProgress` - Create a `FileDownloader` simulator that uses these callbacks - Simulate download progress, success, and error scenarios - Show how multiple components can subscribe to the same callbacks

---

Topic 2: Lambda Expressions

Exercise 2.1: LINQ-Style Operations

Implement your own LINQ-style operations using lambda expressions.

Learning Objectives: - Understand lambda expression syntax and usage - Practice creating generic methods with lambda parameters - Learn to build fluent interfaces with lambdas

Requirements: - Create extension methods: `MyWhere`, `MySelect`, `MyOrderBy` for `IEnumerable<T>` - Use lambda expressions as parameters for these methods - Test with a list of integers and a list of custom objects - Compare performance with built-in LINQ methods

Implementation Hints:

```
public static class MyLinq
{
    public static IEnumerable<T> MyWhere<T>(this IEnumerable<T> source, Func<
T, bool> predicate)
    {
        foreach (var item in source)
        {
            if (predicate(item))
                yield return item;
        }
    }
}
```

Test Examples:

```
// Test with numbers
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evens = numbers.MyWhere(x => x % 2 == 0);
var doubled = numbers.MySelect(x => x * 2);

// Test with objects (use Person class from test data)
var adults = people.MyWhere(p => p.Age >= 18);
var names = people.MySelect(p => p.Name.ToUpper());
```

Extension Ideas: - Implement `MyGroupBy`, `MyJoin`, `MyAggregate` - Add performance benchmarking - Create deferred execution like real LINQ

Exercise 2.2: Event Handler Registration

Create a flexible event handler system using lambdas.

Requirements: - Create a `Button` class with a `Click` event - Register multiple event handlers using lambda expressions - Include handlers that: log clicks, count clicks, validate conditions - Show how to add and remove lambda-based handlers dynamically

Exercise 2.3: Functional Programming Patterns

Implement common functional programming patterns with lambdas.

Requirements: - Create `Map`, `Filter`, and `Reduce` functions for collections - Implement function composition: `Compose(f, g)` returns `x => f(g(x))` - Create a `Curry` function that converts multi-parameter functions to single-parameter chains - Demonstrate with mathematical operations and string processing

Exercise 2.4: Configuration Builder

Build a fluent configuration system using lambda expressions.

Requirements: - Create a `DatabaseConfig` class with properties for connection string, timeout, etc. - Implement a fluent builder pattern: `config.WithHost(h => h.Name = "localhost")` - Use lambdas to configure nested objects and collections - Support conditional configuration based on environment

Exercise 2.5: Expression Trees

Work with expression trees to build dynamic queries.

Requirements: - Create a simple query builder that constructs `Expression<Func<T, bool>>` - Support basic operations: equals, greater than, contains - Combine expressions with AND/OR logic - Compile and execute the expressions against data collections

---

Topic 3: Enumeration and Iterators

Exercise 3.1: Custom Range Generator

Create flexible range generators using yield return.

Learning Objectives: - Master the `yield return` keyword and lazy evaluation - Understand iterator methods and deferred execution - Practice creating memory-efficient data generators

Requirements: - Implement `Range(start, end)`, `Range(start, end, step)` - Create `EvenNumbers(max)`, `OddNumbers(max)`, `PrimeNumbers(max)` - Implement `InfiniteSequence()` that generates numbers forever - Test memory efficiency with large ranges

Implementation Hints:

```
public static IEnumerable<int> Range(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        yield return i;
    }
}

public static IEnumerable<int> PrimeNumbers(int max)
{
    for (int i = 2; i <= max; i++)
    {
        if (IsPrime(i))
            yield return i;
    }
}
```

Key Concepts: - `yield return` creates an iterator method - Values are generated on-demand (lazy evaluation) - Memory usage remains constant regardless of range size - Can create infinite sequences that don’t consume infinite memory

Testing Ideas:

```
// Test memory efficiency
var largeRange = Range(1, 1000000);
// Memory usage stays low until enumeration begins

var firstTen = largeRange.Take(10).ToList();
// Only first 10 values are actually generated
```

Exercise 3.2: File System Walker

Build a file system iterator that walks directories.

Requirements: - Create `WalkFiles(directory)` that yields file paths - Implement filtering: by extension, by size, by date - Support recursive and non-recursive modes - Add `WalkDirectories()` that yields directory information - Handle errors gracefully (permissions, missing files)

Exercise 3.3: Data Stream Processor

Create an iterator that processes data streams lazily.

Requirements: - Implement `ReadCsvRows(filename)` that yields one row at a time - Create `ParseNumbers(textlines)` that converts strings to numbers - Implement `BatchProcessor(items, batchSize)` that yields batches - Chain these together to process large CSV files efficiently

Exercise 3.4: Game Level Generator

Build a procedural level generator using iterators.

Requirements: - Create `GenerateRooms(width, height)` that yields room coordinates - Implement `GenerateEnemies(difficulty)` that yields enemy placements - Create `GenerateTreasures(rarity)` for treasure placement - Combine generators to create complete level layouts - Support seeded random generation for reproducible levels

Exercise 3.5: Async Enumerable

Implement asynchronous enumeration for I/O operations.

Requirements: - Create `IAsyncEnumerable<T>` implementation for reading web APIs - Implement `FetchPagesAsync(url)` that yields web page data - Create `ProcessItemsAsync()` that handles each item asynchronously - Demonstrate cancellation and error handling - Compare performance with synchronous alternatives

---

Topic 4: Extension Methods

Exercise 4.1: String Utilities

Create a comprehensive string extension library.

Learning Objectives: - Understand extension method syntax and constraints - Practice extending built-in types with useful functionality - Learn to create fluent, chainable APIs

Requirements: - Implement: `IsPalindrome()`, `WordCount()`, `TitleCase()` - Add: `RemoveExtraSpaces()`, `ExtractNumbers()`, `IsValidEmail()` - Create: `Truncate(length, suffix)`, `ToSlug()` for URLs - Include: `CountOccurrences(substring)`, `ReplaceMultiple(dictionary)`

Implementation Hints:

```
public static class StringExtensions
{
    public static bool IsPalindrome(this string text)
    {
        if (string.IsNullOrEmpty(text)) return false;

        var clean = text.ToLower().Replace(" ", "");
        return clean == new string(clean.Reverse().ToArray());
    }

    public static string ToSlug(this string text)
    {
        return text.ToLower()
                   .Replace(" ", "-")
                   .RemoveSpecialCharacters()
                   .RemoveExtraHyphens();
    }
}
```

Key Concepts: - Extension methods must be in static classes - First parameter uses `this` keyword - Can chain extension methods together - Null checking is important for robustness

Test Examples:

```
// Use the test data from appendix
var testStrings = TestData.GetStringData();
foreach (var str in testStrings)
{
    Console.WriteLine($"{str} -> {str.ToSlug()}");
    Console.WriteLine($"Word count: {str.WordCount()}");
    Console.WriteLine($"Is palindrome: {str.IsPalindrome()}");
}
```

Exercise 4.2: Collection Enhancements

Extend collection types with useful operations.

Requirements: - For `IEnumerable<T>`: `Shuffle()`, `TakeRandom(count)`, `IsEmpty()` - For `List<T>`: `RemoveDuplicates()`, `Split(predicate)`, `MoveToFront(item)` - For `Dictionary<K,V>`: `GetOrAdd(key, factory)`, `TryGetValues(keys)` - Include: `ForEach(action)`, `WhereNot(predicate)`, `DistinctBy(selector)`

Exercise 4.3: Numeric Extensions

Create mathematical extensions for numeric types.

Requirements: - For int: `IsEven()`, `IsOdd()`, `IsPrime()`, `ToOrdinal()` (1st, 2nd, 3rd) - For double: `Round(decimals)`, `IsNearlyEqual(other, tolerance)` - For all numbers:

`Clamp(min, max)`, `ToPercentage()`, `Factorial()` - Include: `Between(min, max)`, `ToBytes()`, `FromRadians()`, `ToDegrees()`

Exercise 4.4: DateTime Extensions

Build a date/time utility library.

Requirements: - Implement: `StartOfWeek()`, `EndOfMonth()`, `IsWeekend()`, `IsHoliday()` - Add: `Age()`, `BusinessDaysUntil(date)`, `ToTimeAgo()` ("2 hours ago") - Create: `ToUnixTimestamp()`, `FromUnixTimestamp()`, `ToIso8601()` - Include: `IsInRange(start, end)`, `GetQuarter()`, `ToFiscalYear()`

Exercise 4.5: Validation Extensions

Create a fluent validation system using extensions.

Requirements: - For strings: `IsNotEmpty()`, `HasMinLength(n)`, `MatchesPattern(regex)` - For numbers: `IsPositive()`, `IsInRange(min, max)`, `IsNotZero()` - For objects: `IsNotNull()`, `Satisfies(predicate)` - Chain validations: `value.IsNotNull().IsPositive().IsInRange(1, 100)` - Collect all validation errors instead of failing fast

---

Topic 5: Anonymous Types

Exercise 5.1: Data Transformation Pipeline

Create data transformation using anonymous types for intermediate steps.

Requirements: - Start with a `Customer` class (Name, Email, Orders, etc.) - Transform through multiple anonymous types for different processing steps - Create: customer summary, order statistics, contact information - Project into final result types for different output formats - Demonstrate type inference and IntelliSense benefits

Exercise 5.2: Configuration Builder

Build a settings system using anonymous types.

Requirements: - Create application settings using anonymous types - Support nested configurations (database, logging, UI settings) - Implement merging of configuration sources (default + user + environment) - Convert anonymous types to strongly-typed configuration classes - Handle missing properties and default values

Exercise 5.3: Report Generator

Generate various reports using anonymous types for flexibility.

Requirements: - Create sales data with products, customers, dates, amounts - Generate different report formats using anonymous type projections - Include: monthly summaries, customer reports, product performance - Support dynamic grouping and aggregation - Export to different formats (console, CSV, JSON simulation)

Exercise 5.4: Query Result Mapper

Map database-style query results using anonymous types.

Requirements: - Simulate database results with lists of dictionaries - Use anonymous types to provide strongly-typed access to query results - Support joins between multiple "tables" - Handle optional fields and null values - Convert to business objects for application use

Exercise 5.5: Test Data Builder

Create a test data generation system using anonymous types.

Requirements: - Use anonymous types to specify test data variations - Support inheritance and composition of test scenarios - Generate collections of test objects with controlled variations - Include: realistic data (names, addresses), edge cases, invalid data - Demonstrate how anonymous types simplify test setup

---

Topic 6: Tuples

Exercise 6.1: Multiple Return Values

Refactor methods to use tuples instead of out parameters.

Learning Objectives: - Understand tuple syntax and named tuple elements - Learn when to use tuples vs. custom classes - Practice tuple deconstruction and pattern matching

Requirements: - Create methods that return multiple related values using tuples - Implement: `ParseNamedDecimal(string)` → `(bool success, decimal value, string error)` - Create: `GetMinMaxAverage(numbers)` → `(min, max, average)` - Implement: `ValidateUser(user)` → `(isValid, errors, warnings)` - Compare readability with out parameters and custom result classes

Implementation Hints:

```
// Named tuple elements improve readability
public static (bool success, decimal value, string error) ParseNamedDecimal(s
tring input)
{
    if (string.IsNullOrEmpty(input))
        return (false, 0m, "Input cannot be null or empty");
    if (decimal.TryParse(input, out decimal result))
        return (true, result, null);

    return (false, 0m, $"Cannot parse '{input}' as decimal");
}

// Tuple deconstruction in usage
var (success, value, error) = ParseNamedDecimal("123.45");
if (success)
    Console.WriteLine($"Parsed value: {value}");
else
    Console.WriteLine($"Error: {error}");
```

Key Concepts: - Named tuples provide better readability than Item1, Item2 - Tuples are value types (struct) - Deconstruction allows clean unpacking of values - Good for temporary groupings, not complex business objects

Comparison Exercise:

```
// Old way with out parameters
public static bool TryParseDecimal(string input, out decimal value, out strin
g error)

// New way with tuples
public static (bool success, decimal value, string error) ParseDecimal(string
 input)

// Usage comparison shows tuple advantages
```

Exercise 6.2: Coordinate System

Build a 2D/3D coordinate system using tuples.

Requirements: - Use tuples for 2D points: `(double x, double y)` - Implement operations: distance, midpoint, rotation, translation - Extend to 3D: `(double x, double y, double z)` - Create methods for: vector math, collision detection, bounds checking - Demonstrate tuple deconstruction in calculations

Exercise 6.3: Database-Style Operations

Simulate database operations using tuples for records.

Requirements: - Use named tuples to represent database records - Implement: SELECT (projection), WHERE (filtering), JOIN operations - Create: `GroupBy()` that returns `(key, IEnumerable<record>)` tuples - Support sorting and paging with tuple-based results - Handle null values and optional fields in tuple records

Exercise 6.4: State Machine

Implement a simple state machine using tuples.

Requirements: - Use tuples to represent state transitions: `(currentState, input)` → `(newState, output)` - Create a finite state machine for: traffic light, vending machine, or game state - Support: state validation, transition logging, state history - Implement: state queries, event handlers, rollback functionality - Use pattern matching with tuples for state logic

Exercise 6.5: Configuration Parsing

Parse and validate configuration using tuples.

Requirements: - Parse configuration files into tuples of `(section, key, value)` - Validate configuration returning `(isValid, parsedConfig, errors)` - Support: type conversion, default values, environment overrides - Group related settings using nested tuples - Generate configuration documentation from tuple definitions

---

Topic 7: Pattern Matching

Exercise 7.1: Expression Evaluator

Build a mathematical expression evaluator using pattern matching.

Requirements: - Define expression types: `Number`, `BinaryOp`, `UnaryOp`, `Variable` - Use pattern matching to evaluate expressions recursively - Support: +, -, *, /, %, and parentheses - Include: variable substitution, expression simplification - Handle: division by zero, invalid operations, undefined variables

Exercise 7.2: HTTP Request Router

Create a web request router using pattern matching.

Requirements: - Define request types with method, path, headers, body - Use pattern matching to route requests to handlers - Support: path parameters, query strings, HTTP methods - Include: middleware processing, authentication checks, content negotiation - Handle: 404 errors, method not allowed, parameter validation

Exercise 7.3: Game Entity System

Design a game entity system with pattern matching.

Requirements: - Create entity types: `Player`, `Enemy`, `Item`, `Obstacle` - Use pattern matching for: collision detection, damage calculation, AI behavior - Support: entity interactions, state changes, event handling - Include: different enemy types, special abilities, environmental effects - Handle: entity lifecycle, cleanup, performance optimization

Exercise 7.4: Data Validation System

Build a comprehensive validation system using pattern matching.

Requirements: - Define validation rules using discriminated unions - Use pattern matching to apply rules to data objects - Support: field validation, cross-field validation, conditional rules - Include: custom error messages, severity levels, rule composition - Handle: internationalization, dynamic rules, performance optimization

Exercise 7.5: Command Pattern Implementation

Implement the command pattern using pattern matching.

Requirements: - Define command types: `Create`, `Update`, `Delete`, `Query` - Use pattern matching to execute commands against data stores - Support: undo/redo, command batching, transaction handling - Include: command validation, authorization, audit logging - Handle: command failure, rollback, distributed operations

---

Topic 8: Generics

Exercise 8.1: Generic Data Structures

Implement fundamental data structures with generics.

Requirements: - Create: `Stack<T>`, `Queue<T>`, `BinaryTree<T>` with full functionality - Implement: `CircularBuffer<T>`, `PriorityQueue<T>`, `LRUCache<T, V>` - Support: enumeration, LINQ compatibility, thread safety options - Include: capacity management, memory optimization, performance monitoring - Handle: null values, comparison operations, serialization

Exercise 8.2: Repository Pattern

Build a generic repository system for data access.

Requirements: - Create: `IRepository<T>` with CRUD operations - Implement: `InMemoryRepository<T>`, `FileRepository<T>` - Support: querying with expressions, paging, sorting - Include: unit of work pattern, change tracking, caching - Handle: concurrency, transactions, error recovery

Exercise 8.3: Event System

Design a type-safe event system using generics.

Requirements: - Create: `EventBus` with strongly-typed event publishing/subscribing - Implement: `IEvent<T>`, `IEventHandler<T>`, priority-based handling - Support: async event handling, event filtering, subscription management - Include: event replay, dead letter handling, performance monitoring - Handle: circular dependencies, memory leaks, exception isolation

Exercise 8.4: Validation Framework

Build a flexible validation framework with generics.

Requirements: - Create: `Validator<T>` with fluent rule definition - Implement: `ValidationRule<T, P>` for property-specific rules - Support: nested object validation, collection validation, conditional rules - Include: custom validators, localization, rule composition - Handle: performance optimization, circular references, async validation

Exercise 8.5: Functional Programming Library

Create functional programming utilities with generics.

Requirements: - Implement: `Option<T>`, `Result<T, E>`, `Either<L, R>` monads - Create: `Func<T, R>` extensions for composition, currying, memoization - Support: lazy evaluation, pipeline operations, error handling - Include: async variants, LINQ integration, performance optimization - Handle: null safety, exception transformation, resource management

---

Topic 9: Collections

Exercise 9.1: Custom Collection Types

Implement specialized collection types for specific use cases.

Requirements: - Create: `ObservableList<T>` with change notifications - Implement: `SortedSet<T>` that maintains order automatically - Build: `MultiMap<K, V>` that allows multiple values per key - Create: `TimedCache<K, V>` with automatic expiration - Include: thread safety, enumeration safety, memory efficiency

Exercise 9.2: Collection Algorithms

Implement advanced algorithms for collection processing.

Requirements: - Create: sorting algorithms (QuickSort, MergeSort, HeapSort) - Implement: searching algorithms (binary search, interpolation search) - Build: set operations (union, intersection, difference, symmetric difference) - Create: graph algorithms (DFS, BFS, shortest path) - Include: performance benchmarking, algorithm selection, visualization helpers

Exercise 9.3: Data Processing Pipeline

Build a high-performance data processing system.

Requirements: - Create: `DataProcessor<T>` with transformation pipeline support - Implement: parallel processing, batching, streaming - Support: filtering, mapping, reducing, grouping operations - Include: error handling, progress reporting, memory management - Handle: backpressure, flow control, resource cleanup

Exercise 9.4: Collection Utilities

Create a comprehensive utility library for collections.

Requirements: - Implement: deep cloning, comparison, serialization for any collection - Create: collection diff/merge operations, change tracking - Build: collection converters between different types - Create: collection validators, statistics generators - Include: performance optimization, memory profiling, debugging helpers

Exercise 9.5: Concurrent Collections

Implement thread-safe collection operations.

Requirements: - Create: lock-free data structures (stack, queue, hash table) - Implement: producer-consumer patterns, work distribution - Support: atomic operations, memory barriers, cache optimization - Include: deadlock detection, performance monitoring, stress testing - Handle: ABA problems, memory reclamation, scalability issues

---

Topic 10: LINQ

Exercise 10.1: Custom LINQ Providers

Implement your own LINQ query provider.

Requirements: - Create: `IQueryable<T>` implementation for custom data sources - Implement: expression tree parsing and translation - Support: basic operations (Where, Select, OrderBy, GroupBy) - Include: query optimization, caching, performance monitoring - Handle: complex expressions, nested queries, parameter binding

Exercise 10.2: Data Analytics Engine

Build a data analytics system using LINQ.

Requirements: - Create: sales/financial data analysis with complex aggregations - Implement: trend analysis, correlation calculations, forecasting - Support: time-series operations, moving averages, statistical functions - Include: data visualization helpers, export capabilities - Handle: large datasets, memory optimization, parallel processing

Exercise 10.3: Query Builder

Create a dynamic query builder with LINQ.

Requirements: - Build: runtime query construction from user input - Implement: type-safe property selection, operator handling - Support: complex conditions (AND, OR, nested), sorting, paging - Include: query validation, SQL generation, caching - Handle: injection prevention, performance optimization, error handling

Exercise 10.4: Data Transformation Engine

Implement a flexible data transformation system.

Requirements: - Create: mapping between different object types using LINQ - Implement: configuration-based transformations, custom converters - Support: nested objects, collections, conditional mapping - Include: validation, error handling, performance monitoring - Handle: circular references, null values, type mismatches

Exercise 10.5: Stream Processing System

Build a real-time stream processing engine.

Requirements: - Create: continuous query processing over data streams - Implement: windowing operations (tumbling, sliding, session) - Support: complex event processing, pattern detection - Include: state management, fault tolerance, scaling - Handle: late data, out-of-order events, backpressure

---

Bonus Challenges

Multi-Topic Integration Exercises

Challenge A: Build a Complete Mini-Framework

Combine multiple topics to create a web application framework: - Use delegates for middleware pipeline - Implement routing with pattern matching - Create generic repository with LINQ support - Add validation using extension methods - Include configuration with anonymous types

Challenge B: Game Engine Architecture

Design a simple game engine incorporating: - Entity system with generics and pattern matching - Event system using delegates and lambda expressions - Collection-based spatial partitioning - LINQ-based query system for game objects - Iterator-based level generation

Challenge C: Data Processing Platform

Create a data processing platform featuring: - ETL pipeline using iterators and LINQ - Type-safe configuration with tuples and anonymous types - Extensible processing with delegates and generics - Validation framework using extension methods - Pattern matching for data routing

---

Tips for Success

Problem-Solving Approach:

1. **Break down** each exercise into smaller steps
2. **Start simple** and add complexity gradually
3. **Test frequently** with small examples first
4. **Refactor** your code as you learn better patterns

Development Practices:

1. **Write unit tests** for your solutions
2. **Use meaningful names** for classes and methods
3. **Add comments** explaining complex logic
4. **Consider edge cases** and error handling

Learning Resources:

1. **Reference the lecture examples** when stuck
2. **Use IntelliSense** to explore available methods
3. **Read C# documentation** for built-in types
4. **Experiment** with variations of the exercises

Going Further:

1. **Optimize performance** of your solutions
2. **Add additional features** beyond requirements
3. **Combine concepts** from different exercises
4. **Share and discuss** your solutions with peers

Exercise Tracking

Use this checklist to track your progress:

Delegates

- □ Exercise 1.1: Basic Calculator
- □ Exercise 1.2: Event System
- □ Exercise 1.3: Data Pipeline
- □ Exercise 1.4: Filter System
- □ Exercise 1.5: Async Callbacks

Lambda Expressions

- □ Exercise 2.1: LINQ Operations
- □ Exercise 2.2: Event Handlers
- □ Exercise 2.3: Functional Patterns
- □ Exercise 2.4: Configuration Builder
- □ Exercise 2.5: Expression Trees

Enumeration and Iterators

- □ Exercise 3.1: Range Generator
- □ Exercise 3.2: File System Walker
- □ Exercise 3.3: Stream Processor
- □ Exercise 3.4: Level Generator
- □ Exercise 3.5: Async Enumerable

Extension Methods

- □ Exercise 4.1: String Utilities
- □ Exercise 4.2: Collection Enhancements
- □ Exercise 4.3: Numeric Extensions
- □ Exercise 4.4: DateTime Extensions
- □ Exercise 4.5: Validation Extensions

Anonymous Types

- □ Exercise 5.1: Data Transformation
- □ Exercise 5.2: Configuration Builder
- □ Exercise 5.3: Report Generator
- □ Exercise 5.4: Query Mapper
- □ Exercise 5.5: Test Data Builder

Tuples

- □ Exercise 6.1: Multiple Returns
- □ Exercise 6.2: Coordinate System
- □ Exercise 6.3: Database Operations
- □ Exercise 6.4: State Machine

- □ Exercise 6.5: Configuration Parsing

Pattern Matching

- □ Exercise 7.1: Expression Evaluator
- □ Exercise 7.2: HTTP Router
- □ Exercise 7.3: Game Entities
- □ Exercise 7.4: Data Validation
- □ Exercise 7.5: Command Pattern

Generics

- □ Exercise 8.1: Data Structures
- □ Exercise 8.2: Repository Pattern
- □ Exercise 8.3: Event System
- □ Exercise 8.4: Validation Framework
- □ Exercise 8.5: Functional Library

Collections

- □ Exercise 9.1: Custom Collections
- □ Exercise 9.2: Collection Algorithms
- □ Exercise 9.3: Processing Pipeline
- □ Exercise 9.4: Collection Utilities
- □ Exercise 9.5: Concurrent Collections

LINQ

- □ Exercise 10.1: Custom Providers
- □ Exercise 10.2: Analytics Engine
- □ Exercise 10.3: Query Builder
- □ Exercise 10.4: Transformation Engine
- □ Exercise 10.5: Stream Processing

Bonus Challenges

- □ Challenge A: Mini-Framework
- □ Challenge B: Game Engine
- □ Challenge C: Data Platform

Happy Coding! 🚀

Remember: The goal is not just to complete the exercises, but to deeply understand the concepts and patterns. Take your time, experiment, and don’t hesitate to go beyond the requirements.

Appendix: Test data for exercises2.3 can be found in the project

ExerciseData.cs – has all the classes needed plus a class to generate data for the exercises.

sample-data.csv – some test data in CSV format

config.json – some test data in JSON format

This test data provides realistic scenarios for all exercises and helps you focus on learning the concepts rather than creating sample data!