# Arrays & Lists in C#

Week 7

# Today’s Agenda

- Introduction to Data Collections
- Understanding Arrays
- Working with Lists
- Common Operations & Algorithms
- AI-assisted Collections Manipulation
- Practical Coding Session
- Your Turn: Student Exercise

# Why Collections Matter

- Programs often need to work with **multiple related values**
- Accessing data items **individually becomes inefficient**
- Collections let us **organize and manipulate** data effectively
- **Real-world examples**: student grades, inventory items, sensor readings

# Arrays vs. Lists: Key Differences

| | Arrays | Lists |
|---|---|---|
| **Size** | Fixed at creation | Dynamic (can grow/shrink) |
| **Memory** | Contiguous memory block | References to elements |
| **Declaration** | `type[] name = new type[size]` | `List<type> name = new List<type>()` |
| **Flexibility** | Less flexible | More flexible |
| **Performance** | Slightly faster | Slight overhead |

# Arrays in C#

```
// Declaration and initialization
int[] numbers = new int[5]; // Creates array of 5 integers (all 0)

// Initialization with values
int[] scores = new int[] { 95, 88, 72, 84, 91 };
// Or shorter syntax
string[] days = { "Mon", "Tue", "Wed", "Thu", "Fri" };

// Accessing elements (zero-based indexing)
int firstScore = scores[0]; // 95
scores[2] = 75; // Change 72 to 75
```

# Arrays: Important Concepts

- Fixed size (cannot grow or shrink)
- Zero-based indexing
- All elements must be of same type
- Access by position is very fast (O(1))
- Multi-dimensional arrays are possible

```
// 2D array (3 rows, 4 columns)
int[,] grid = new int[3, 4];

// Jagged array (array of arrays)
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[] { 1, 3, 5 };
jaggedArray[1] = new int[] { 2, 4 };
```

# Common Array Operations

```
// Finding length
int count = numbers.Length;

// Looping through array
for (int i = 0; i < scores.Length; i++) {
    Console.WriteLine($"Score {i+1}: {scores[i]}");
}

// Using foreach loop
foreach (int score in scores) {
    Console.WriteLine($"Score: {score}");
}

// Array methods
Array.Sort(scores);     // Sort in-place
Array.Reverse(scores);  // Reverse in-place
bool found = Array.Exists(scores, s ⇒ s > 90); // Search with predicate
```

# Lists in C#

```
// Need to import
using System.Collections.Generic;

// Declaration and initialization
List<string> names = new List<string>();

// Adding elements
names.Add("Alice");
names.Add("Bob");
names.Add("Charlie");

// Initialization with values
List<int> numbers = new List<int> { 10, 20, 30, 40 };
```

# List Operations

```
// Adding and removing
names.Add("David");       // Add to end
names.Insert(1, "Eve");   // Insert at index 1
names.Remove("Bob");      // Remove specific item
names.RemoveAt(0);        // Remove at index

// Accessing elements
string first = names[0];  // Access by index (like arrays)
names[1] = "Frank";       // Modify element

// Properties and methods
int count = names.Count;  // Number of elements
bool hasAlice = names.Contains("Alice");
int position = names.IndexOf("Charlie");
```

# More List Features

```
// Range operations
names.AddRange(new[] { "Grace", "Henry" });  // Add multiple
names.RemoveRange(1, 2);  // Remove 2 items starting at index 1

// Advanced operations
names.Sort();             // Sort alphabetically
names.Reverse();          // Reverse order
names.Clear();            // Remove all elements

// Find operations
string firstWithA = names.Find(n => n.StartsWith("A"));
List<string> longNames = names.FindAll(n => n.Length > 4);
```

# Common Algorithms with Collections

1. **Searching**: Find element(s) matching criteria
2. **Sorting**: Arrange elements in specific order
3. **Filtering**: Create subset based on condition
4. **Aggregation**: Calculate sum, average, etc.
5. **Transformation**: Create new collection by modifying elements

These patterns appear in virtually all programming applications!

# AI-Assisted Collections Example

```
// AI can help with common collection patterns
// Example: AI-generated code to find students who passed

List<Student> students = GetStudents();
double passingGrade = 60.0;

// AI might generate:
List<Student> passingStudents = students
    .Where(student => student.Grade >= passingGrade)
    .OrderByDescending(student => student.Grade)
    .ToList();

foreach (var student in passingStudents) {
    Console.WriteLine($"{student.Name}: {student.Grade}");
}
```

# Real-World Applications

- **E-commerce**: Product inventory management
- **Games**: Tracking game objects, scores, player states
- **Finance**: Transaction histories, account records
- **IoT**: Sensor data collection and analysis
- **Social Media**: User posts, friends lists, comments

# Performance Considerations

- Arrays are slightly faster for fixed collections
- Lists are better when size may change
- Accessing by index is fast for both (O(1))
- Searching unsorted collections is slow (O(n))
- Consider using Dictionary<K,V> for key-based lookups

# Time for Guided Coding

Let’s build a simple grade tracker application:

- Store student grades in both arrays and lists
- Calculate statistics (average, highest, lowest)
- Sort and filter grades
- Compare array vs list approaches

# Your Turn: Student Exercise

Create a simple inventory management system that:

1. Stores product information (name, price, quantity)
2. Allows adding and removing products
3. Calculates total inventory value
4. Finds products below a certain quantity threshold
5. Sorts products by price

# Key Takeaways

- Arrays are fixed-size collections for simple scenarios
- Lists provide flexibility for dynamic collections
- Both use zero-based indexing and similar access patterns
- Choose the right collection type based on your needs
- Most real applications use multiple collection types

# Next Week

## Week 8: Strings & String Manipulation

- Working with text data
- String methods and properties
- Regular expressions
- Text processing algorithms

# Questions?