Advance Programming Techniques (APT)

Lecture # 8

Ehtisham Rasheed

Department of Computer Science
University of Gurjat, Gujrat

Problems with Arrays

- Fixed Size
- Cumbersome Operations
  - Insertion
  - Deletion
- Leads To
  - Manual memory management
  - Complex code
  - Bugs

Collections

- Collections are classes that provide an easy way to work with a group of objects
- Collections have following advantages over arrays
  - Collections are dynamic – they grow or shrink as needed
  - Rich Functionality – Collections come with built-in methods (like Add, Sort, Remove etc) that make code easier
  - Type Safe (Generics) – Generic collections are type safe

Types of C# Collection Classes

- In C# collections are divided into 3 classes
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Concurrent

System.Collections Classes

- They help us to create non-generic collection
- We can create classes where we can add data elements of multiple data types
- Oldest collection types (before generics were introduced in .NET 2.0)
- Less type-safe and slower than generics
- Following are the common classes that come under this namespace
  - ArrayList Class
  - Hashtable Class
  - Stack & Queue
  - Sorted List

System.Collections.Generic Classes

- They help us to create a generic collection
- In this we store type compatible data elements
- Faster and safer than non-generic
- Preferred in modern C# development
- Following are the common classes that come under this namespace
  - List <T>
  - Stack <T>
  - Queue <T>
  - Dictionary <TKey, TValue>
  - SortedList <TKey, TValue>

System.Collections.Concurrent Classes

- Designed for multi-threaded and parallel programming
- They provide classes that help to achieve thread-safe code
- Thread-safe → multiple threads can read/write without locks
- Useful when building high-performance, scalable applications
- Following are the common classes that come under this namespace
  - ConcurrentStack<T>
  - ConcurrentQueue<T>
  - ConcurrentDictionary<TKey, TValue>

C# List

- List<T> is a class that contains multiple objects of the same data type that can be accessed using an index. For example,

```
// list containing integer values
List<int> number = new List<int>() { 1, 2, 3 };
```

- We can access individual element by index
- We can iterate list using for as well as foreach loop
- We can Add(), Insert() and Remove [Remove(), RemoveAt()] elements from List

C# Stack & Queue

```
Stack<int> history = new Stack<int>();
history.Push(1);
history.Push(2);
Console.WriteLine(history.Pop());  // LIFO

Queue<string> requests = new Queue<string>();
requests.Enqueue("Request 1");
requests.Enqueue("Request 2");
Console.WriteLine(requests.Dequeue());  // FIFO
```

C# Dictionary


```
Dictionary<int, string> students = new Dictionary<int, string>();
students.Add(101, "Ali");
students.Add(102, "Sara");

foreach (var kvp in students)
    Console.WriteLine($"Roll {kvp.Key}: {kvp.Value}");
```

C# SortedList

```
// Create a SortedList of Product IDs and Names
SortedList<int, string> products = new SortedList<int, string>();

products.Add(103, "Keyboard");
products.Add(101, "Monitor");
products.Add(104, "Mouse");
products.Add(102, "CPU");

Console.WriteLine("Product List (Sorted by Key):");
foreach (var item in products)
{
    Console.WriteLine($"ID: {item.Key}, Name: {item.Value}");
}
```

List<T> - Dynamic Array

- When to use
  - When you need a **resizable array** (can grow/shrink dynamically)
  - When **index-based access** (by position) is important
  - When **order of elements** matters
  - Best for storing and iterating through **a list of items** (students, products, employees, etc.)
- Don’t use when
  - You need **fast insertion/removal** at both ends – use **Queue** or **Stack**
  - You need **key-based access** — use **Dictionary**

Stack<T> - LIFO (Last In, First Out)

- When to use
  - When you need **reverse-order processing** (last item added is first removed)
  - Common in **Undo/Redo**, **Expression Evaluation**, **Backtracking**, etc
- Don’t use when
  - You need random access or iteration order — use **List**

Queue<T> - FIFO (First In, First Out)

- When to use
  - When you need to process items in the **same order** they arrive
  - Common in **task scheduling**, **message processing**, **print jobs**, **order queues**, etc
- Don’t use when
  - You need reverse-order or random access — use **Stack** or **List**

Dictionary<Tkey, TValue> - Key-Value Pairs

- When to use
  - When you need **fast lookups** using a **unique key**
  - Excellent for scenarios like:
    - StudentID → StudentName
    - ProductCode → Price
    - Username → Password
- Don’t use when
  - Order of insertion or sorting matters — use **SortedList**

SortedList<Tkey, TValue> - Sorted Key-Value Pairs

- When to use
  - When you want a **dictionary-like structure** that also keeps **elements sorted by key** automatically
  - Useful for things like:
    - Product catalog sorted by ID or name
    - Student list sorted by roll number
    - Scoreboard sorted by score
- Don’t use when
  - Data doesn’t need to be sorted → **Dictionary** is faster

Example Scenario

- All these collections can be used in one application. Like Library Management System
  - List<Books> → All Available Books
  - Stack<Books> → Recently viewed books
  - Queue<Member> → Members waiting for a book
  - Dictionary<int, Book> → BookID to BookInfo
  - SortedList<int, string> → Sorted catalog by BookID