

Collections in C#.NET: Generic and Non Generic	Collections
	Non generic
	ArrayList
	Hash table
	Sorted list
	Stack
	Queue
	Generic
	List
	Dictionary
	Sorted list
	Stack
	Queue

Latest C# 7.0 features	Extension Methods
	Tuples and Deconstruction, Pattern Matching, Local Functions, Out Variables, Ref Locals and Returns, Discards (_) & Expression-bodied Members
C# 8.0 features	Nullable Reference Types, Async Streams, Ranges and Indices, Switch Expressions, Default Interface Methods, Pattern Matching Enhancements, Using Declarations & Readonly Members

```

Program.cs  Demo.Collection.Stack.Queue
25 //Step5: Check if the stack contains a specific value( Searching functionality)
26 //Step6: Display the current count of items in the stack(Using Count property of Stack class)
27 //Step7: Clear the stack of all items(Using Clear method of Stack class)
28 0 references
29 static void Main(string[] args)
30 {
31     1 Stack<int> stack = new Stack<int>(); //Stack that can hold integer
32     // Step 2: Push some values onto the stack
33     stack.Push(10); // In list we were using Add() method to add values
34     stack.Push(20);
35     stack.Push(30);
36     // Step 3: Pop a value from the stack and display it
37     2 int poppedValue = stack.Pop();
38     Console.WriteLine("Popped Value: " + poppedValue);
39     // Step 4: Peek at the top value of the stack without removing it
40     3 int topValue = stack.Peek();
41     Console.WriteLine("Top Value: " + topValue);
42     // Step 5: Check if the stack contains a specific value
43     4 bool contains20 = stack.Contains(20);
44     Console.WriteLine("Stack contains 20: " + contains20);
45     // Step 6: Display the current count of items in the stack
46     5 int count = stack.Count; //this is same as other collection classes
47     Console.WriteLine("Current Count: " + count);
48
49     // Step 7: Clear the stack of all items
50     6 stack.Clear();
51     Console.WriteLine("Stack cleared. Current Count after clearing: " +
52 }

```

Output

Show output from: Build

You are screen sharing Stop share

Popped Value: 30  
Top Value: 20  
Stack contains 20: True  
Current Count: 2  
Stack cleared. Current Count after clearing: 0

C:\Users\Parth\source\repos\Wipro MS\_Dynamics\_8thJan26\Day\_6\_Collectinosand\_other\_Features\Demo.Collection.Stack.Queue\bin\Debug\Demo.Collection.Stack.Queue.exe (process 37804) exited with code 0 (0x0).  
Press any key to close this window . . .

//Common use case of collection ie Stack and Queue in real life scenario are as below :

- //1. Stack : Undo mechanism in text editors , Browser history etc
- //2. Queue : Print spooling , CPU task scheduling , Call center systems etc
- //3. Priority Queue : Emergency room triage , Task scheduling in operating systems , Network packet routing etc
- //4. Deque : Browser history navigation , Undo/Redo functionality in applications , Task scheduling with varying priorities etc
- //5. Concurrent Collections : Real-time data processing , Multi-threaded web servers , Parallel computing tasks etc
- //6. list use cases : Data binding in UI frameworks , Implementing stacks and queues , Storing collections of items for processing etc

//Step1: Define a stack to hold integer values ex.  
 //Step2: Push some values onto the stack  
 //Step3: Pop a value from the stack and display it  
 //Step4: Peek at the top value of the stack without removing it( All functionality are provided by Stack class in System.Collections.Generic namespace)  
 //Step5: Check if the stack contains a specific value( Searching functionality)  
 //Step6: Display the current count of items in the stack(Using Count property of Stack class)  
 //Step7: Clear the stack of all items(Using Clear method of Stack class)

### Enterprise Training Management System (ETMS)

A platform used by an L&D organization to manage learners, trainers, courses, enrollments, and session analytics.

**Why is it important to select an appropriate collection class for any user story?**

“**performance**

**Scalability**

**Maintainability**

<p><b>Manage Course Catalog</b></p> <p><b>As a Training Admin,</b> I want to maintain a list of available courses So that I can easily add, update, and display them.</p> <p>List&lt;Course&gt;</p>	<p><b>Fast Course Lookup by Course Code</b></p> <p><b>As a System,</b> I want to fetch course details instantly using course code So that performance is optimized.</p> <p>Dictionary&lt;string, Course&gt;</p>	<p><b>Avoid Duplicate Learner Enrollments</b></p> <p><b>As a Program Manager,</b> I want to ensure a learner cannot enroll twice in the same course.</p> <p>HashSet&lt;int&gt;</p>
<p><b>Process Enrollment Requests in Order</b></p> <p><b>As an Enrollment Service,</b> I want to process enrollment requests in the order received.</p> <p>Queue&lt;EnrollmentRequest&gt;</p>	<p><b>Support Undo for Admin Actions</b></p> <p><b>As an Admin,</b> I want to undo the last action performed.</p> <p>Stack&lt;AdminAction&gt;</p>	<p><b>Display Sessions Sorted by Start Time</b></p> <p><b>As a Learner,</b> I want to see training sessions sorted chronologically.</p> <p>SortedList&lt;DateTime, Sessions&gt;</p>

### Handle Concurrent Enrollments (Enterprise Scale)

**As a System,**

I want to safely manage enrollments when multiple users enroll simultaneously.

ConcurrentDictionary<int, Enrollment>

User Need	Collection Used	Reason
Ordered data	List<T>	Easy iteration
Fast lookup	Dictionary< TKey , TValue >	O(1) access
Uniqueness	HashSet<T>	No duplicates
Sequential processing	Queue<T>	FIFO
Undo functionality	Stack<T>	LIFO
Sorted view	SortedList< TKey , TValue >	Auto sorting
Concurrent access	ConcurrentDictionary	Thread safety

C# -> OOP to C#(Performance and productivity)

Feature	Introduced In
Generics	C# 2.0
LINQ(language integrated Query)	C# 3.0
Async / Await ( Multithreading)	C# 5.0
Null Safety	C# 8.0
Records (Immutability)	C# 9.0
Minimal APIs	C# 9.0+
Performance Optimizations	C# 7.2+

Modern Syntax & Productivity	C# 10–12
------------------------------	----------

C# Version	Release Year	Key Features Introduced
C# 1.0	2002	Classes, structs, interfaces, enums, delegates, properties, events, exception handling
C# 2.0	2005	<b>Generics</b> , nullable value types, anonymous methods, iterators ( <code>yield</code> ), partial classes
C# 3.0	2007	<b>LINQ</b> , lambda expressions, anonymous types, extension methods, auto-implemented properties
C# 4.0	2010	<code>dynamic</code> type, named & optional parameters, COM interop improvements
C# 5.0	2012	<b>Async / Await</b> , task-based asynchronous programming
C# 6.0	2015	String interpolation, null-conditional operator ( <code>? .</code> ), expression-bodied members, <code>nameof()</code>
C# 7.0	2017	Tuples, pattern matching, local functions, <code>out</code> variables
C# 7.1	2017	<code>Async Main</code> , default literal expressions
C# 7.2	2017	<code>ref readonly</code> , <code>in</code> parameters, performance enhancements
C# 7.3	2018	Improved generics constraints, safer <code>stackalloc</code> , performance improvements
C# 8.0	2019	<b>Nullable reference types</b> , async streams, default interface methods, switch expressions
C# 9.0	2020	<b>Records</b> , init-only setters, top-level programs, pattern matching enhancements
C# 10.0	2021	Global using directives, file-scoped namespaces, record structs, improved lambdas
C# 11.0	2022	Required members, raw string literals, generic math, list patterns

C# 12.0	2023	Primary constructors for classes & structs, collection expressions, inline arrays
---------	------	---

The screenshot shows the Microsoft Visual Studio IDE. The main window displays the code for `Program.cs` under the tab `Demo_Collection_Stack_Queue`. The code demonstrates the use of lambda expressions and implicit type variables. A tooltip is visible over the line `var evenNumber = numbers.Where(n => n % 2 == 0);`, explaining that `var` is an implicit type variable.

```

52 //with the help of lambda expression we can write it in one line like
53 Func<int, bool> IsEven = number => number % 2 == 0;
54
55 //IN a collection if we want to implement lambda expressino to find a number greater than 10
56 List<int> numbers = new List<int> { 3, 5, 81, 45, 32, 15, 70 };
57
58 // displaying elements of the list
59 foreach (int number in numbers)
60 {
61     Console.WriteLine(number);
62 }
63 int result = numbers.Find(n => n > 10); //lambda expression C#3.0
64 var evenNumber = numbers.Where(n => n % 2 == 0); //var is implicit type variable
65 Console.WriteLine("here are the list of even number in the house..");
66 foreach (var item in evenNumber)
67 {
68     Console.WriteLine(item);
69 }

```

A separate window titled "Microsoft Visual Studio Debug" shows the output of the program:

```

Popped Value: 30
Top Value: 20
Stack contains 20: True
Current Count: 2
Stack cleared. Current Count after clearing: 0
3
5
81
45
32
15
here are the list of even number in the house..
32

```

### Case study based on Features till C# 7 :

#### E-Commerce Order & Inventory Management System

The system processes customer orders, validates inventory, calculates pricing, and generates fulfillment instructions efficiently.

<b>Validate Order Status Dynamically</b>	<b>Return Multiple Values from Pricing Engine</b>	<b>Extend Existing Class Without Modifying It</b>
--	---	---

<b>As a System,</b> I want to process orders differently based on status So that business rules are enforced correctly.	<b>As a Pricing Service,</b> I want to return the price, tax, and discount together.	<b>As a Developer,</b> I want to add discount logic to Order class safely.
Feature Used: <b>Pattern Matching</b>	Tuples & Deconstruction	Extension Methods

<b>Reuse Logic Inside a Method</b>	<b>Improve Method Output Clarity</b>	<b>Avoid Unnecessary Variables</b>
<b>As a Developer,</b> I want to validate inventory without exposing helper methods.	<b>As a Service,</b> I want to return the success status and message clearly.	<b>As a Developer,</b> I want to ignore values I don't need.
Local Functions	Out Variables	Discards (_)

<b>Optimize Inventory Access</b>	<b>Simplify Read-Only Logic</b>
<b>As a System,</b> I want to modify stock values efficiently.  Ref Locals & Returns	<b>As a Developer,</b> I want cleaner syntax for simple methods.  Expression-Bodied Members

C# 7.0 Feature	Used For
Pattern Matching	Dynamic order status
Tuples	Multiple return values
Extension Methods	Business logic extension
Local Functions	Encapsulated helpers
Out Variables	Cleaner output handling
Ref Locals	Performance optimization
Discards	Ignore unused values
Expression-bodied	Clean syntax

# Case Study

## Modern Learning Management System (LMS) Using C# 8.0 Features

---

### 1. Business Context

A training organization is modernizing its **Learning Management System (LMS)** to improve:

- Code safety and null handling
- Asynchronous data processing
- Readability and maintainability
- Performance for large datasets
- Consistency across enterprise APIs

The system is upgraded to **C# 8.0** to leverage modern language features.

---

### 2. Objectives

Objective	Description
Code Safety	Eliminate null reference errors
Async Processing	Stream large datasets efficiently
Maintainability	Use concise and expressive syntax

Performance	Reduce memory overhead
Enterprise Readiness	Support versioned APIs

### 3. C# 8.0 Features Applied

Feature	Usage Context
Nullable Reference Types	Prevent null-related runtime failures
Async Streams	Stream learner records
Ranges and Indices	Slice datasets
Switch Expressions	Clean decision logic
Default Interface Methods	Backward-compatible APIs
Pattern Matching Enhancements	Advanced type checks
Using Declarations	Automatic resource cleanup
Readonly Members	Immutable data safety

---

## 4. User Stories and Feature Implementation

---

### User Story 1: Prevent Null Reference Failures

**As a System Architect,**

I want compile-time detection of null reference risks.

Feature	Nullable Reference Types

```
#nullable enable

class Learner

{
    public string Name { get; set; } = string.Empty;
    public string? Email { get; set; }
}
```

Outcome:

Null safety is enforced at compile time.

---

### User Story 2: Stream Large Learner Records Asynchronously

**As an Analytics Module,**

I want to process learners without loading all data into memory.

<b>Feature</b>	<b>Async Streams</b>
----------------	----------------------

```
async IAsyncEnumerable<Learner> GetLearnersAsync()
{
    for (int i = 1; i <= 5; i++)
    {
        await Task.Delay(500);

        yield return new Learner { Name = $"Learner {i}" };
    }
}
```

---

### User Story 3: Process Partial Datasets Efficiently

**As a Reporting Service,**  
I want to analyze recent learners only.

<b>Feature</b>	<b>Ranges and Indices</b>
----------------	---------------------------

```
var learners = new[] { "A", "B", "C", "D", "E" };

var recentLearners = learners[^2..];
```

---

### User Story 4: Simplify Business Rules

**As a Developer,**

I want cleaner logic for course status evaluation.

Feature	Switch Expressions
---------	--------------------

```
string GetCourseLevel(int rating) => rating switch
{
    >= 8 => "Advanced",
    >= 5 => "Intermediate",
    _ => "Beginner"
};
```

---

**User Story 5: Maintain Backward-Compatible Interfaces****As an API Designer,**

I want to add new methods without breaking implementations.

Feature	Default Interface Methods
---------	---------------------------

```
interface ICourseService
{
    void Enroll();

    void Notify() => Console.WriteLine("Default Notification");
}
```

## User Story 6: Improve Type Safety in Condition Checks

**As a System,**

I want to validate inputs more accurately.

Feature	Pattern Matching Enhancements

```
bool Validate(object input) =>  
    input is int count && count > 0;
```

## User Story 7: Ensure Deterministic Resource Cleanup

**As a Developer,**

I want automatic disposal of resources.

Feature	Using Declarations

```
using var reader = new StreamReader("data.txt");  
string content = reader.ReadToEnd();
```

## User Story 8: Protect Immutable Data

**As a Security Requirement,**

I want critical data to be read-only.

Feature	Readonly Members

```

struct Certificate

{
    public readonly int Id;
    public readonly string Title;

    public Certificate(int id, string title)
    {
        Id = id;
        Title = title;
    }
}

```

## 5. Feature-to-Business Mapping Summary

C# 8.0 Feature	Business Value
Nullable Reference Types	Fewer runtime failures
Async Streams	Scalable data processing

Ranges & Indices	Efficient data slicing
Switch Expressions	Clean business logic
Default Interface Methods	API stability
Pattern Matching	Safer validations
Using Declarations	Resource safety
Readonly Members	Data integrity

## 6. Learning Outcomes

Outcome	Description
Code Reliability	Compile-time null checks
Performance	Reduced memory usage
Maintainability	Cleaner, expressive syntax
Enterprise Readiness	Backward-compatible systems

---

## 7. Conclusion

C# 8.0 introduces significant improvements that move .NET applications toward **safer, more expressive, and enterprise-grade architectures.**

These features collectively reduce technical debt and prepare systems for scalable, cloud-native deployments.