Asp.NET MVC Core

Trainer : Priyanka R Rangole

Email: priyanka.rangole@sunbeaminfo.com

C# Collections

Collections in C# are **specialized classes** used to store and manage groups of values or objects.
There are **two types**:
•**Non-generic collections** → System.Collections
•**Generic collections** → System.Collections.Generic

Why Generic Collections?
•**Type-safe** → Only store specific data type
•**Faster performance**
•**Prevents runtime errors** → Catches mistakes at compile time
•**Avoids boxing/unboxing**

Generic Collections (System.Collections.Generic)
Below are commonly used generic collection classes:

Generic Collection	Description
List	Stores elements of a specific type. Automatically grows as items are added.
Dictionary<TKey, TValue>	Stores key–value pairs (unique keys).
SortedList<TKey, TValue>	Stores key–value pairs in ascending order of keys.
Queue	FIFO (First In First Out). Uses Enqueue() to add & Dequeue() to remove.
Stack	LIFO (Last In First Out). Uses Push() to add & Pop()/Peek() to retrieve.
HashSet	Stores unique elements only (no duplicates). Very fast lookup.

Non-Generic Collections (System.Collections)

These collections store **objects of any type**, but are **less safe and slower** due to boxing/unboxing

Non-Generic Collection	Usage / Description
ArrayList	Resizable array that stores items of any type (not type-safe).
SortedList	Key–value pairs stored in ascending key order (generic + non-generic versions available).
Stack	LIFO structure (generic + non-generic).
Queue	FIFO structure (generic + non-generic).
Hashtable	Stores key–value pairs using hashing for fast lookup. Keys must be unique.
BitArray	Stores bits (true = 1, false = 0) in a compact form. Useful for flags/boolean values.

Use Generic Collections for:
•Faster performance
•Type safety
•Clean and professional C# code

Non-Generic Collections:
•Older, less safe
•Still used in some legacy systems

ARRAYLIST — (C# Non-Generic Collection)

What is ArrayList?

•ArrayList is a **non-generic** collection in C#.
•Located in **System.Collections** namespace
•Similar to an array, **but size increases dynamically**.
•Can store **any type of data** (int, string, bool, objects, null).

When to use?

✔ When you **don’t know the type** of data
✔ When you **don’t know the size** ahead of time
✖ Not recommended for modern projects → use List<T> instead.

Creating an ArrayList

```
using System.Collections;

ArrayList arlist = new ArrayList();
```

Adding Elements into ArrayList

Using Add()

```
var arlist1 = new ArrayList();
arlist1.Add(1);
arlist1.Add("Bill");
arlist1.Add(true);
arlist1.Add(4.5);
arlist1.Add(null);
```

Accessing ArrayList Elements

ArrayList uses **indexer** like an array
(Index starts at 0)

```
ArrayList arlist = new ArrayList() { 1, "Bill", 300, 4.5f };

int first = (int)arlist[0];
string second = (string)arlist[1];
```

What is an Exception?

- An **Exception** is an error that occurs at **runtime**.
- In C#, every exception is an **object of a class** derived from the Exception base class.

Exception Class Hierarchy

```
Exception
│
├── SystemException     (Base class for all built-in/system exceptions)
│      ├── NullReferenceException
│      ├── DivideByZeroException
│      ├── IndexOutOfRangeException
│      ├── InvalidOperationException
│      ├── FileNotFoundException
│      └── many more...
│
└── ApplicationException  (Used earlier for custom exceptions → NOT recommended now)
```

Note:Microsoft **recommends** creating custom exceptions by inheriting **directly from Exception**, NOT from ApplicationException.

Why Exceptions Occur?

Examples:
•Accessing a null object → NullReferenceException
•Dividing by zero → DivideByZeroException
•Invalid index → IndexOutOfRangeException

Exception → base class

✓ **SystemException → all built-in exceptions**

✓ **ApplicationException → old recommendation, now NOT preferred**

✓ **Use custom exceptions only when no system exception fits**

Exception Handling in C#

Exception handling ensures that your program **does not crash** when an unexpected error occurs.
C# handles exceptions using:
•**try**
•**catch**
•**finally**

What is Exception Handling?
Exception handling allows you to:
•Prevent application crash
•Show meaningful messages
•Log errors
•Continue program execution safely

Syntax

```
try
{
   // Code that may throw an exception
}
catch (Exception ex)
{
   // Handle exception
}
finally
{
   // Always executed (cleanup code)
}
```

try Block
•Contains the **risky/suspected code**.
•If any exception occurs → control immediately jumps to the **catch** block.
•Without catch/finally, try alone causes a compile-time error.

catch Block
•Handles exceptions.
•Can log, display messages, or take corrective actions.
•Can accept a parameter to get exception details.

finally Block
•Always executes — whether exception occurs or not.
•Used for **cleanup**:

- Close files
- Release connections
- Dispose objects

Basic Example (without handling)

```
Console.WriteLine("Enter a number:");
var num = int.Parse(Console.ReadLine());
Console.WriteLine(num * num);
```

Example with try–catch–finally

```
try
{
   Console.WriteLine("Enter a number:");

   var num = int.Parse(Console.ReadLine());

   Console.WriteLine($"Square of {num} is {num * num}");
}
catch
{
   Console.WriteLine("Error occurred.");
}
finally
{
   Console.WriteLine("Re-try with a different number.");
}
```

Exception Filters (Multiple Catch Blocks)

```
try
{
   int num = int.Parse(Console.ReadLine());
   int result = 100 / num;
}
catch(DivideByZeroException)
{
   Console.WriteLine("Cannot divide by zero.");
}
catch(InvalidOperationException)
{
   Console.WriteLine("Invalid operation.");
}
catch(FormatException)
{
   Console.WriteLine("Enter valid number.");
}
catch(Exception)
{
   Console.WriteLine("Unknown error occurred.");
}
```

C# — throw Keyword

The throw keyword is used when **you want to raise an exception manually** in your code.
Normally, exceptions come automatically from CLR.
But sometimes **you want to check your own conditions** and raise an exception

When do we use throw?

- When input values are invalid
- When objects are null
- When a method must stop execution due to some condition
- When you want to force an error and let the caller handle it

Manually Throwing an Exception

```
static void Main(string[] args)
{
    Student std = null;

    try
    {
        PrintStudentName(std);
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

 static void PrintStudentName(Student std)
{
    if (std == null)
        throw new NullReferenceException("Student object is null.");

    Console.WriteLine(std.StudentName);
}
```

Custom Exception Types in C#

C# already provides many built-in exceptions like:
•DivideByZeroException
•FormatException
•OutOfMemoryException
But in real applications, sometimes **your business rules get violated**, and built-in exceptions are not enough.

Example:A school app may require student names to contain **only alphabets**.
If someone enters **James007**, you want to throw a custom exception like:
InvalidStudentNameException

```
class Student
{
   public int StudentID { get; set; }
   public string StudentName { get; set; }
}
[Serializable]
class InvalidStudentNameException : Exception
{
   public InvalidStudentNameException() {  }

   public InvalidStudentNameException(string name)
      : base(String.Format("Invalid Student Name: {0}", name)){}
}
```

LINQ (Language Integrated Query)

What is LINQ?
LINQ (**Language Integrated Query**) is a feature of C# that allows you to write **queries directly inside the C# language** using a consistent, readable syntax.
It is integrated into C#, just like methods, classes, or events.
LINQ lets you query different data sources using **one common syntax**.

Data sources you can query using LINQ:
•**Collections / Objects** (LINQ to Objects)
•**Databases** (LINQ to SQL / Entity Framework)
•**XML files** (LINQ to XML)
•**Datasets** (LINQ to DataSet)
•**Web services, JSON, etc.**

Why LINQ?

Before LINQ:
•Every data source had its **own query language** (SQL for DB, XPath for XML, loops for objects)
•Developers had to manually convert query results into objects
With LINQ:
•One common query syntax for all sources
•Strongly typed queries (C# checks errors at compile-time)
•Easy to read, maintain, and reuse
•Results come as **objects** (no need for conversions)

Basic LINQ Query Example

```
string[] names = { "Bill", "Steve", "James", "Mohan" };
```

LINQ Query (Query Syntax):

```
var myLinqQuery = from name in names
                  where name.Contains('a')
                  select name;
```

Query Execution:

```
foreach (var name in myLinqQuery)
   Console.Write(name + " ");
```

LINQ always needs a data source
(Example: arrays, lists, XML, database, dataset)
Writing a LINQ query does not run it
The query is executed only when you iterate over it
→ This is called **Deferred Execution**
LINQ queries return objects
You can use OOP concepts directly on query results.

LINQ Syntax Styles

LINQ supports two formats:

1. **Query Syntax** → similar to SQL
2. **Method Syntax** → uses extension methods (Where, Select, etc.)

```
var myLinqQuery = names.Where(n => n.Contains('a'));
```

LINQ Method Syntax uses **extension methods** from

System.Linq namespace such as:

- Where()
- Select()
- OrderBy()
- GroupBy()
- First() / FirstOrDefault()
- Any(), All()
- Count(), Sum(), Average()
- Take(), Skip()
- Distinct()

LinQ Methods

Method	Purpose
Where()	Filter
Select()	Transform / Pick fields
OrderBy() / OrderByDescending()	Sorting
GroupBy()	Grouping
First() / FirstOrDefault()	First element
Any() / All()	Boolean checks
Count(), Sum(), Min()…	Aggregation
Take(), Skip()	Paging
Distinct()	Remove duplicates
ToList(), ToArray()	Convert results