Pre-Joining Topics

Week 5: C# Fundamentals

LINQ

LINQ (Language Integrated Query) is a powerful feature introduced in **C# 3.0 (.NET Framework 3.5)** that allows you to **query data directly in C#** using a **SQL-like syntax**, but in a **type-safe, object-oriented** way.

LINQ integrates querying capabilities directly into the C# language syntax.

You can use LINQ to query data from:

- **Collections (arrays, lists, dictionaries, etc.)**
- **Databases (LINQ to SQL / Entity Framework)**
- **XML documents (LINQ to XML)**
- **DataSets**
- **Files, JSON, etc.**

Traditional Approach	LINQ Approach
You manually loop through collections, use if conditions, etc.	You just write a single expressive query.
Non-type-safe (runtime errors possible)	Type-safe and compile-time checking
Complex and lengthy code	Clean, readable, and short
Hard to maintain and modify	Easy to modify and extend
Works separately for each data source	Unified query syntax for all data sources

Architecture of LINQ

Internally, LINQ uses:

- **Extension Methods** (from System.Linq)
- **Delegates**, **Lambda Expressions**
- **Deferred Execution** (query executes only when you iterate over results)

- **IEnumerable<T>** (which includes list, array, queue, stack, hashset, dictionary etc) and **IQueryable<T>** interfaces

Types of LINQ

1. **LINQ to Objects** → Queries in-memory collections (List, Array, etc.)
2. **LINQ to SQL** → Queries SQL Server databases
3. **LINQ to XML** → Queries XML data
4. **LINQ to DataSet** → Queries DataSets
5. **LINQ to Entities** → Queries Entity Framework models
6. **Parallel LINQ (PLINQ)** → Parallelized LINQ for performance

LINQ Syntax Types

There are **two** syntax styles:

1. Query Syntax (SQL-like style)

```
var result = from s in students
        where s.Marks > 80
        orderby s.Name
        select s;
```

2. Method Syntax (Extension method + Lambda)

```
var result = students
        .Where(s => s.Marks > 80)
        .OrderBy(s => s.Name)
        .Select(s => s);
```

Deferred vs Immediate Execution

Deferred Execution:

Query executes **only when iterated** (e.g., in foreach).

```
var result = numbers.Where(n => n > 5);  // Not executed yet
```

```
foreach(var n in result)            // Executes here
   Console.WriteLine(n);
```

Immediate Execution:

When you use methods like .ToList(), .Count(), .ToArray(), query executes **immediately**.

```
var result = numbers.Where(n => n > 5).ToList(); // Executes immediately
```

Internal Implementation of LINQ

Step 1: Query is Written

```
var query = students.Where(s => s.Marks > 80).Select(s => s.Name);
```

Step 2: Extension Methods Are Invoked

All LINQ methods like Where, Select, OrderBy, etc. are **extension methods** in the **System.Linq.Enumerable** (for in-memory) or **System.Linq.Queryable** (for database) class.

So the compiler converts it to:

```
var query = Enumerable.Select(
        Enumerable.Where(students, s => s.Marks > 80),
        s => s.Name);
```

Step 3: Delegates and Lambda Expressions

- The expression s => s.Marks > 80 is a **lambda expression**.
- Internally, this becomes a **delegate** (like Func<Student, bool>).

So:

```
students.Where(s => s.Marks > 80);
```

is interpreted as:

```
students.Where(new Func<Student, bool>(s => s.Marks > 80));
```

The Where method stores this delegate and uses it to test each element later.

Step 4: Deferred Execution Begins

When you call .Where() or .Select(), nothing actually executes immediately.

The query is only **defined**, not executed yet.
LINQ builds an **expression pipeline** (like a chain of filters and transformations).

```
var query = students.Where(s => s.Marks > 80);
```

Step 5: Execution Happens During Enumeration

Execution actually happens when you **enumerate** the result:

```
foreach (var s in query)
{
   Console.WriteLine(s.Name);
}
```

Step 6: Iterator and Yield Return

LINQ uses the **iterator pattern** internally — meaning it yields results one by one.

Step 7: Deferred Execution vs Immediate Execution

- **Deferred Execution:**
  Methods like Where(), Select(), OrderBy() are lazy — they execute only when iterated.
- **Immediate Execution:**
  Methods like ToList(), Count(), Sum(), First(), etc. **force execution** immediately because they need to compute the result.

ORM

ORM = Object-Relational Mapping

- It's a technique that **maps database tables to classes** and **rows to objects**.
- This allows you to **interact with a database using objects** instead of writing raw SQL queries.

Benefits:

1. Reduces boilerplate SQL code.
2. Helps prevent SQL injection by using parameterized queries.
3. Makes database operations easier to maintain in code.
4. Works with multiple database systems with minimal code changes.

Popular ORMs in .NET:

- **Entity Framework Core** (EF Core) most widely used
- NHibernate

- Dapper (lightweight micro ORM)

Example analogy:

- Database table = Class
- Table row = Object / instance of class
- Column = Property

EF Core Setup & DbContext

What is DbContext?

- DbContext is the **main class** in Entity Framework Core that manages the database connection and **maps your classes to tables**.
- It's like a **bridge between your code and database**.

Responsibilities of DbContext:

1. Querying the database
2. Saving data back to the database
3. Tracking changes to objects
4. Configuring database tables and relationships

Setting up EF Core in a C# Project

Step 1: Create a Console App

dotnet new console -n EFCoreDemo

cd EFCoreDemo

Step 2: Install EF Core packages (for SQL Server)

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

Flow of Operation

1. Create object

var student = new Student { Name = "Heer", Age = 22 };

2. **Add to DbSet**

context.Students.Add(student);

3. **SaveChanges** → EF Core generates **SQL INSERT** behind the scenes:

INSERT INTO Students (Name, Age) VALUES ('Heer', 22);

4. **Database row is created** → Object in memory now has Id assigned (if identity column).

Microsoft.EntityFrameworkCore

It provides all the ORM (Object Relational Mapping) functionality — allowing you to work with databases using .NET objects instead of writing SQL manually.

Microsoft.EntityFrameworkCore.Tools
It allows you to use Entity Framework CLI commands in Visual Studio Package Manager Console or dotnet CLI.

MySql.Data

Purpose: This is the official MySQL data provider for .NET.
It provides low-level ADO.NET support for MySQL — things like MySqlConnection, MySqlCommand, etc

MySql.EntityFrameworkCore

Purpose: This is the Entity Framework Core provider for MySQL.
It allows EF Core (the main ORM) to work specifically with MySQL databases.

Fluent API in Entity Framework Core (EF Core)

Introduction

- Fluent API is a code-based configuration of your EF Core models, written in the OnModelCreating method of your DbContext.
- Provides full control over entity properties, table mapping, relationships, and constraints.
- More powerful and flexible than Data Annotations, especially for complex mappings.

. Key Points

1. Fluent API is written inside OnModelCreating() — EF Core automatically uses it.
2. Configurable items with Fluent API:
    - Table and column names
    - Primary keys
    - Relationships (1:1, 1:N, M:N)
    - Constraints (Required, MaxLength, Default values)
    - Cascade delete or update rules
3. One-to-Many relationship example: Department → Employees
4. Cascade delete ensures related Employees are deleted if Department is deleted.
5. Navigation properties allow EF Core to automatically manage foreign keys and relationships.

1️⃣ Data Annotations

- Configuration is **done using attributes** directly on the model class or properties.
- Quick and simple for basic constraints.
- Limited for complex relationships or advanced scenarios.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;


public class Employee
{
    [Key]            // Primary key
    public int Id { get; set; }

    [Required]       // Not nullable
    [MaxLength(50)]  // Max length
    public string Name { get; set; }
```

```
    [Column("DeptName")] // Custom column name
    public string Department { get; set; }
}
```

---

2 Fluent API

- Configuration is **done in code** inside OnModelCreating() of DbContext.
- More flexible; can handle **complex relationships, composite keys, cascade rules, indexes**.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
        .HasKey(e => e.Id);         // Primary key

    modelBuilder.Entity<Employee>()
        .Property(e => e.Name)
        .IsRequired()               // Not nullable
        .HasMaxLength(50)
        .HasColumnName("EmployeeName"); // Custom column name
}
```

Feature	Data Annotations	Fluent API
Location	On model class	Inside DbContext (OnModelCreating)
Complexity	Simple / basic	Flexible / advanced
Relationships	Limited	Full support (1:1, 1:N, M:N)
Composite keys / indexes	Not supported	Supported
Cascade rules	Limited	Fully configurable

Topic 1: Cryptography Basics

Definition:
Cryptography is the science of **protecting information** by transforming it into a secure format, so that only authorized people can read it.

It has **two main goals**:

1. **Confidentiality** – keeping information secret
2. **Integrity & Authentication** – ensuring data is not altered and verifying sender identity

Types of Cryptography

Type	Description	Example Use
Symmetric Key	Same key used for **encryption and decryption**	AES, DES
Asymmetric Key	Uses a **public key** to encrypt and a **private key** to decrypt	RSA
Hashing	Converts data into a **fixed-length string**, cannot be reversed	SHA-256, MD5

Secure Coding Practices

Definition:
Secure coding practices are a set of guidelines and techniques to **write code that is resistant to vulnerabilities and attacks**.

The goal is to **prevent security issues** like SQL injection, XSS, data leaks, and unauthorized access.

1️⃣ **Input Validation**

- Always **validate user input** before processing or storing it.
- Prevents attacks like **SQL Injection, Command Injection, and Buffer Overflows**.

Example (C#):

```
// Validate an email input

string email = GetUserInput();

if (!Regex.IsMatch(email, @"^[^@\s]+@[^@\s]+\.[^@\s]+$"))

{
```

```
    Console.WriteLine("Invalid email format!");
}
else
{
    Console.WriteLine("Valid email: " + email);
}
```

✅ Checks that only properly formatted emails are accepted.

2️⃣ Use Parameterized Queries

- Never build SQL queries by concatenating strings from user input.
- Always use **parameterized queries** or ORM methods.

Unsafe Example (Vulnerable to SQL Injection):

```
string username = GetUserInput();
string query = $"SELECT * FROM Users WHERE Username = '{username}'"; // Bad
```

Safe Example (Using parameterized query):

```
using (var cmd = new MySqlCommand("SELECT * FROM Users WHERE Username = @username", conn))
{
    cmd.Parameters.AddWithValue("@username", username);
    var reader = cmd.ExecuteReader();
}
```

✅ Protects against SQL Injection attacks.

3️⃣ Password Storage

- Never store plain-text passwords.
- Always **hash passwords** using **salted hashes**.

Example (C# SHA-256 with salt):

```
string password = "mypassword";
```

```
byte[] salt = new byte[16];
using (var rng = RandomNumberGenerator.Create())
{
   rng.GetBytes(salt);
}

using (var sha = SHA256.Create())
{
   byte[] passwordBytes = Encoding.UTF8.GetBytes(password);
   byte[] salted = salt.Concat(passwordBytes).ToArray();
   byte[] hash = sha.ComputeHash(salted);
   Console.WriteLine("Hashed Password: " + Convert.ToBase64String(hash));
}
```

✅ Even if database leaks, original password cannot be retrieved easily.

4️⃣ **Principle of Least Privilege**

- Applications and users should have **only the permissions they need**.
- Avoid using **admin/root accounts** for application DB connections.

5️⃣ **Exception Handling**

- Avoid showing **internal errors** to users.
- Log detailed exceptions securely for developers.
- Return **generic error messages** to prevent information leakage.

```
try
{
   DoDatabaseOperation();
}
catch (Exception ex)
```

```
{
   // Log exception internally
   Logger.Log(ex);

   // Show generic message to user
   Console.WriteLine("Something went wrong. Please try again later.");
}
```

Dynamic Type:

In C#, variables are usually **statically typed**, meaning their type must be known at **compile-time**. However, there are scenarios where the type of a variable is not known until **runtime**. The dynamic type was introduced in **C# 4.0** to handle such cases.

- **Key Idea:** dynamic bypasses compile-time type checking; its type is resolved at **runtime**.
- **Difference from var:** var is inferred at compile-time, while dynamic is determined at runtime.
- `dynamic` is a **special type** in C# whose operations are **checked at runtime** rather than compile-time.
- Variables declared as `dynamic` can **change their type** during program execution.

Syntax Example:

```
dynamic data = 10;       // integer at runtime
Console.WriteLine(data); // Output: 10

data = "Hello";          // now it is a string
Console.WriteLine(data); // Output: Hello
```

4. Usage of dynamic

dynamic is useful in scenarios where the type is **unknown at compile-time**:

1. **Interop with COM objects or Office Automation**:

```
dynamic excel = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));

excel.Visible = true;
```

2. **Working with JSON or XML**:

```
dynamic json = JsonConvert.DeserializeObject(jsonString);
```

```
Console.WriteLine(json.name); // Resolved at runtime
```

3. **Using Reflection**:

```
dynamic obj = GetSomeObject();
obj.DoSomething();  // Method resolved at runtime
```

4. **Heterogeneous Data Handling**:

```
dynamic data = 5;      // integer
data = "string now";   // string
data = new List<int>(); // list
```

Limitations of dynamic

1. **No Compile-time Type Checking**
    - Calling a non-existent method or property **will crash at runtime**:

```
dynamic data = 5;
data.NonExistingMethod(); // Compiles, but runtime error
```

2. **Performance Overhead**
    - Operations are slower compared to statically typed variables because type resolution happens at runtime.
3. **Limited IntelliSense Support**
    - IDEs like Visual Studio cannot provide full code suggestions.
4. **Reduced Safety**
    - More prone to runtime errors; type errors are not caught during compilation.
5. **Cannot Be Used in Certain Contexts**
    - dynamic cannot be used as **case labels in switch statements**, **constants**, or **attribute parameters**.

Feature	**Raw ADO.NET (MySqlCommand)**	**ServiceStack.OrmLite**
Type	Low-level data access (manual SQL)	Lightweight ORM (Object–Relational Mapper)

Feature	Raw ADO.NET (MySqlCommand)	ServiceStack.OrmLite
How it works	You write full SQL commands (SELECT, INSERT, etc.)	You work directly with C# objects — OrmLite creates SQL for you
Code Complexity	More code (connections, commands, readers, etc.)	Less code, cleaner syntax
Control over SQL	Full control	Limited (but you can still use custom SQL)

Operation	ADO.NET (Raw SQL)	OrmLite
Insert/Update/Delete (Single Row)	Very fast (no overhead)	Slightly slower (adds mapping + SQL generation)
Read (SELECT)	Fast	Slightly slower
Bulk Insert (1000+ rows)	Faster if manually optimized	Slower unless you use batch features
Small Queries (normal app usage)	Almost same	Almost same