LINQ — Working with List<T> and DataTable

---

1. Introduction to LINQ

LINQ (Language Integrated Query) is a feature in C# that allows querying data directly within the language syntax.
It provides a consistent way to filter, sort, group, and transform data across various sources.

Key Features

- Unified syntax for querying objects, collections, DataTables, XML, and databases.
- Compile-time checking and IntelliSense support.
- Works with in-memory data (`List<T>`) and ADO.NET data (`DataTable`).
- Reduces boilerplate code and improves readability.

Namespaces Required

```
using System.Linq;
using System.Collections.Generic;
using System.Data;
```

---

2. LINQ Syntax Types

Query Syntax (SQL-like)

```
var result = from x in collection
             where x.Age > 25
             select x;
```

Method Syntax (Lambda-based)

```
var result = collection.Where(x => x.Age > 25).ToList();
```

Both syntaxes produce the same result internally.

---

3. LINQ with `List<T>`

LINQ is commonly used with **generic lists** and **in-memory collections**.

Example Setup

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string City { get; set; } = "";
}

List<Person> people = new()
{
    new() { Id = 1, Name = "Sahil", City = "Mumbai" },
    new() { Id = 2, Name = "Hakim", City = "Delhi" },
    new() { Id = 3, Name = "Nahin", City = "Mumbai" }
};
```

---

4. Common Methods

Method	Description
`Where()`	Filters data
`Select()`	Projects specific fields
`OrderBy()` / `OrderByDescending()`	Sorts data
`GroupBy()`	Groups data
`Distinct()`	Removes duplicates
`Any()`	Checks if any match
`All()`	Checks if all match

`Take(n)`	Takes first n results
`Skip(n)`	Skips first n results
`Join()`	Joins two sequences

---

5. LINQ with `DataTable`

LINQ can also query `DataTable` objects using **LINQ to DataSet**.

Note:

- Use `AsEnumerable()` to enable LINQ on DataTable.
- Use `Field<T>("ColumnName")` for type-safe access.

6. LComparison List vs Datatable

Feature	`List<T>`	`DataTable`
Type Safety	Strongly typed (`T`)	Weakly typed (`DataRow`)
Performance	Faster	Slower
LINQ Support	Built-in	Via `AsEnumerable()`
Conversion	Easy to use with models	Requires `Field<T>()`
Use Case	In-memory collections	Legacy / ADO.NET scenarios

💬 In Short

> ORM lets developers work with **C# objects instead of SQL queries**, making database access simpler, cleaner, and more maintainable.

📘 Overview of ORM Tools

🧩 1. What is ORM?

ORM (Object-Relational Mapping) is a technique that allows developers to interact with a **database using C# objects**, instead of writing SQL queries manually.

It acts as a **bridge** between:

- **C# classes (objects)**
- **Database tables**

🧠 In Simple Words

Without ORM:

```
SELECT * FROM Employees WHERE Department = 'IT';
```

With ORM:

```
var itEmployees = db.Employees.Where(e => e.Department ==
"IT").ToList();
```

The ORM tool automatically converts the C# LINQ query into SQL, executes it, and returns the results as C# objects.

🧱 2. Why ORM is Useful

ORM helps you avoid manual tasks like:

- Writing repetitive SQL queries
- Managing connections and commands
- Mapping data rows to C# objects

Instead, you work directly with **C# objects**.

✅ Benefits of ORM

Benefit	Description
Less SQL Code	Reduces the need to write SQL queries manually.
Type Safety	Data type mismatches are caught during compilation.
Maintainability	Cleaner and easier-to-read code.
Portability	Can easily switch between databases.
Productivity	Faster development as boilerplate code is minimized.

⚙️ 3. How ORM Works (Behind the Scenes)

You define a **C# class** that represents a table:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Department { get; set; }
}
```

1.

You define a **DbContext** to connect your app to the database:

```
public class AppDbContext : DbContext
{
    public DbSet<Employee> Employees { get; set; }
}
```

2.

When you run:

```
List<Employee> employees = context.Employees.ToList();
```

3.
    - ORM generates and executes the SQL: `SELECT * FROM Employees`
    - Returns results as a list of `Employee` objects

You don’t write SQL or handle database connections manually — ORM does it for you.

⚡ 4. Popular ORM Tools in .NET

ORM Tool	Description
Entity Framework Core (EF Core)	Official Microsoft ORM. Fully integrated with .NET and supports LINQ, migrations, and change tracking.
Dapper	A lightweight “micro ORM” developed by Stack Overflow. Extremely fast and allows raw SQL with automatic mapping.
NHibernate	A mature and feature-rich ORM inspired by Java’s Hibernate. Offers advanced caching and mapping features.
LLBLGen Pro	A commercial ORM tool that provides GUI designers and high performance.

🔐 Cryptography & Secure Coding Practices

1. Cryptography Goals

- Confidentiality → Keep data secret
- Integrity → Ensure data isn't modified
- Authentication → Verify sender identity
- Non-repudiation → Sender can't deny message

2. Types of Cryptography

Type	Description	Example Algorithm
Symmetric	Same key for encryption & decryption	AES
Asymmetric	Public/Private key pair	RSA
Hashing	One-way transformation	SHA256

3. Symmetric Encryption (AES Example)

- Use AES for fast data encryption.
- Must securely store the key.

4. Asymmetric Encryption (RSA Example)

- Use RSA for secure key exchange.
- Slower but secure.

5. Hashing

- Converts data → fixed length digest.
- Cannot be reversed.
- Use SHA256 or higher.

6. Secure Coding Practices

- Never hardcode credentials.
- Validate all input (prevent SQL Injection).
- Sanitize all output (prevent XSS).
- Always hash & salt passwords.
- Dispose cryptographic objects properly.
- Avoid outdated algorithms (MD5, SHA1).
- Always use HTTPS/TLS.

🧠 Usage and Limitations of Dynamic type

Usage of `dynamic`

1. **Runtime Type Resolution**
    - The `dynamic` type tells the compiler to skip compile-time type checking.
    - The actual type is resolved at **runtime**.

Example:

```
 dynamic value = "Hello";
Console.WriteLine(value.Length);  // Works because at runtime, it's a string
```

    -
2. **Interacting with COM Objects or Reflection**

- Useful when dealing with APIs that return objects whose structure isn't known at compile time (e.g., Office Interop, JSON, or XML objects).

Example:

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

-

3. **Working with ExpandoObject or DynamicObject**
    - You can create objects whose members can be added or removed at runtime.

Example:

```
dynamic person = new System.Dynamic.ExpandoObject();
person.Name = "Shahil";
person.Age = 21;
Console.WriteLine($"{person.Name} is {person.Age} years old");
```

-

4. **Interacting with JSON or Scripting Languages**
    - You can deserialize JSON into a `dynamic` object and access properties directly without defining a class.

Example:

```
string json = "{\"Name\": \"Alice\", \"Age\": 25}";
dynamic data =
System.Text.Json.JsonSerializer.Deserialize<dynamic>(json);
Console.WriteLine(data["Name"]);  // Output: Alice
```

-

5. **Simplifying Reflection Calls**
    - Instead of using reflection methods (`GetMethod`, `Invoke`, etc.), you can call members dynamically.

Example:

```
dynamic obj = new SomeClass();
obj.PrintMessage("Hello!");  // Calls method dynamically at runtime
```

-

Limitations of `dynamic`

1. **No Compile-Time Checking**
    - The compiler doesn't verify method names, property existence, or data types.
    - Errors appear **only at runtime**, leading to potential crashes.

Example:

```
dynamic value = "Hello";
Console.WriteLine(value.Lenght);  // Typo! Compiles fine, but runtime error
```

-

2. **Slower Performance**
    - Each operation on a `dynamic` object uses **runtime binding**, making it slower than using static types.
    - This can impact performance if used in loops or heavy computations.
3. **Loses IntelliSense Support**
    - Visual Studio (and other IDEs) can't predict members of a `dynamic` type.
    - You don't get autocomplete or compile-time hints, making development prone to mistakes.
4. **Limited Use in Generic Code**
    - Dynamic types can break generic type safety.

Example:

```
List<dynamic> list = new List<dynamic>();
```

```
list.Add("text");
list.Add(123);
// Type inconsistencies can occur at runtime
```

5. **Not Suitable for Strongly Typed APIs**
    - If you know the structure or type of data beforehand, `dynamic` reduces clarity and type safety.
    - Prefer `var` or explicit types in those cases.

Key Difference from `var`

Feature	`var`	`dynamic`
Type Resolution	Compile-time	Runtime
Type Safety	Strongly typed	Weakly typed
IntelliSense Support	Yes	No
Performance	Fast	Slower
Typical Use Case	Known types	Unknown types or runtime objects

📘 CRUD operations in C# with ServiceStack.OrmLite

What is OrmLite?

- A lightweight, typed ORM by ServiceStack.
- Map C# classes (POCOs) to database tables easily.
- Works with multiple databases via dialect providers.

Setup Steps:

1. Install NuGet package for target DB (e.g., ServiceStack.OrmLite.Sqlite).
2. Configure dialect provider: `OrmLiteConfig.DialectProvider = SqliteDialect.Provider;`
3. Create `IDbConnectionFactory`, open connection.

Model Definition Example:

```
public class Person
{
    [AutoIncrement]
    public int Id { get; set; }
    [Required]
    public string Name { get; set; } = string.Empty;
    public int Age { get; set; }
    public string City { get; set; } = string.Empty;
}
```

CRUD Operations:

- **Create (Insert):** `db.Insert(person, selectIdentity: true);`
- **Read (Select):** `db.Select<Person>();`, `db.SingleById<Person>(id);`, `db.Select<Person>(p => p.Age == 30);`
- **Update:** Modify object then `db.Update(person);`

- **Delete:** `db.Delete<Person>(p => p.Id == id);` or `db.DeleteById<Person>(id);`

Additional Features:

- Raw SQL queries: `db.SqlList<T>(sql, params)`
- Transactions: `using var trans = db.OpenTransaction()`
- Prefixed attributes for advanced mapping (`[Alias]`, `[Index]`, `[ForeignKey]`)
- Code-first schema creation: `db.CreateTableIfNotExists<T>()`

Advantages:

- Simplicity, minimal configuration
- Strongly typed POCOs
- Efficient, good performance
- Works cross-database

Limitations / Things to watch:

- You'll manage migrations yourself (no built-in EF-style migrations)
- Some complex ORM features (lazy loading, change tracking) are minimal
- Changing model may require table/schema adjustments manually
- Use the correct Dialect Provider for your database