# Week 5 - C# NOTES

Created by Dakshil Gorasiya

# LINQ

**Language-Integrated Query**, is a powerful feature in C# that allows us to write queries for data collections like arrays, lists, or databases in a more readable and expressive way.

Linq is executed two ways

1. Immediate: Immediate execution means that the data source is read and the operation is performed once. Ex: Sum, Average, Count, First, Max
2. Deferred: Deferred execution means that the operation isn't performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement.

To make deferred result immediate use ToList() or ToArray()

## There is two way to write linq query

### Method way

Ex:

```
var booksAfter1950Method = books.Where(b ⇒ b.PublicationYear > 1950)
                               .Join(authors,
                                  b ⇒ b.AuthorId,
                                  a ⇒ a.Id,
                                  (b, a) ⇒ new
                                  {
                                     BookTitle = b.Title,
```

```
                    AuthorName = a.Name,
                    Year = b.PublicationYear
                })
            .OrderBy(result ⇒ result.Year);
```

### Query way

Ex:

```
var booksAfter1950Query = from b in books
                          join a in authors on b.AuthorId equals a.Id
                          where b.PublicationYear > 1950
                          orderby b.PublicationYear
                          select new
                          {
                             BookTitle = b.Title,
                             AuthorName = a.Name,
                          };
```

## Operations

### Filter

```
List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
List<int> odds = nums.Where(n ⇒ n % 2 == 1).ToList();
```

### Select

```
List<string> strings = new List<string> { "apple", "banana", "cherry", "dat
e", "fig", "grape" };
List<int> lens = strings.Select(s ⇒ s.Length).ToList();
```

### Join

```
var booksAfter1950Method = books.Where(b ⇒ b.PublicationYear > 1950)
                        .Join(authors,
                           b ⇒ b.AuthorId,
                           a ⇒ a.Id,
                           (b, a) ⇒ new
                           {
                              BookTitle = b.Title,
                              AuthorName = a.Name,
                              Year = b.PublicationYear
                           })
                        .OrderBy(result ⇒ result.Year);
```

## Group by

```
var departmentStatsMethod = employees.GroupBy(d ⇒ d.Field<int>("Depa
rtmentID"))
                           .Join(departments,
                           empGroup ⇒ empGroup.Key,
                           d ⇒ d.Field<int>("DepartmentID"),
                           (empGroup, d) ⇒ new {
                               DepartmentName = d.Field<string>("Department
Name"),
                               EmployeeCount = empGroup.Count(),
                               AverageSalary = empGroup.Average(e ⇒ e.Field
<decimal>("Salary")),
                               MaximumSalary = empGroup.Sum(e ⇒ e.Field<d
ecimal>("Salary"))
                           });
```

## Order by, pagination

```
var pagedEmployeeMethod = employees.OrderByDescending(e ⇒ e.Field<
decimal>("Salary"))
                        .Skip(1)
```

```
                    .Take(2)
                    .Select(e ⇒ new
                    {
                        Name = e.Field<string>("Name"),
                        Salary = e.Field<decimal>("Salary")
                    });
```

### QuantifierOperator

```
bool isQaDepartment = departments.Any(d ⇒ d.Field<string>("Department
Name") == "QA");
bool isAllEmployeeReceiveMoreThan50000 = employees.All(d ⇒ d.Field<d
ecimal>("Salary") > 50000);
```

### Conversion

```
DataTable engineeringEmployeeTable = employees.Where(e ⇒ e.Field<int>
("DepartmentId") == 1).CopyToDataTable();


Dictionary<string, List<string>> genereDict = genres.ToDictionary(
                                    g ⇒ g.Name,
                                    g ⇒ books
                                    .Where(b ⇒ b.GenreIds.Contains(g.Id))
                                            .Select(b ⇒ b.Title)
                                            .ToList());
```

# EF Core

Library: Microsoft.EntiryFrameworkCore

To connect with MySql use nuget package:
Pomelo.EntityFrameworkCore.MySql

To use CLI to make migration use package:
Microsoft.EntityFrameworkCore.Tools

## DB first approach

First need to Scafold-dbcontext

```
Scaffold-DbContext "Server=localhost;user=root;password=********;data
base=employee" Pomelo.EntityFrameworkCore.MySql -OutputDir Models
```

It will create Models folder and model class in it

It will also create DbContext class. An instance of `DbContext` represents a session with the database which can be used to query and save instances of entities to a database.

## Code first approach

Create models

Create a context class inherit it from DbContext

Register dbsets

```
public DbSet<Student> Students { get; set; }
```

override OnConfiguring method

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBu
ilder)
{
    optionsBuilder.UseMySql("Server=localhost;user=root;password=*****;
database=student", ServerVersion.AutoDetect("Server=localhost;user=roo
```

```
t;password=****;database=employee"));
}
```

Override OnModelCreating method to define relations

```
protected override void OnModelCreating(ModelBuilder modelBuilder){}
```

## To establish one to many relationship

In one side model

```
public int DepartmentId { get; set; }
[ForeignKey("DepartmentId")]
public Department Department { get; set; }
```

In many side model

```
public ICollection<Student> Students { get; set; }
```

In model creating method

```
modelBuilder.Entity<Student>()
    .HasOne(s ⇒ s.Department)
    .WithMany(d ⇒ d.Students)
    .HasForeignKey(s ⇒ s.DepartmentId)
    .OnDelete(DeleteBehavior.Restrict);
```

## To establish many to many relation

models

student model

```
public ICollection<Course> Courses { get; set; }
```

course model

```
public ICollection<Student> Students { get; set; }
```

OnModelCreting method

```
modelBuilder.Entity<Student>()
    .HasMany(s ⇒ s.Courses)
    .WithMany(c ⇒ c.Students)
    .UsingEntity(j ⇒ j.ToTable("StudentCourses"));
```

## Migrations

Migrations: A **migration** is a way to **incrementally update the database schema** to keep it in sync with your **C# model classes**.

To create a migration

```
Add-Migration <migration_name>
```

It will create a new file in Migration folder which have information about new updates It have two methods Up(To apply migration) and Down(To revert migration)

To apply migration to db

```
Update-Database [<MigrationName>]
```

We can use migration name to apply specific migration to db else last migration will be applied

To remove migration

```
Remove-Migration
```

It will remove the last migration

# CRUD

## Read

We can query data using LINQ

```
var students = context.Students
    .Where(s ⇒ s.DepartmentId == computerDepartment.Id)
    .Include(s ⇒ s.Courses)
    .Select(s ⇒ new
    {
        StudentName = s.Name,
        Course = s.Courses.Select(c ⇒ c.Title).ToList(),
        Department = s.Department.Name
    });
```

## Update

```
var data4 = context.T01s.FirstOrDefault(x ⇒ x.T01f01 == 7);
if (data4 != null)
{
    data4.T01f02 = "iOS";
    context.SaveChanges();
}
```

## Delete

```
var data3 = context.T01s.FirstOrDefault(x ⇒ x.T01f01 == 7);
context.T01s.Remove(data3);
context.SaveChanges();
```

## Insert

```
T01 t01 = new T01()
{
   T01f01 = 7,
   T01f02 = "Android",
   T01f03 = "anand"
};
context.T01s.Add(t01);
context.SaveChanges();
```

## SP

If it returns result first we need to create a DTO for sp’s result and add it in context class

```
string departmentName = "Engineering";
List<GetEmployeeByDepartmentResult> data5 = context.GetEmployeeByD
epartmentResults.FromSqlInterpolated($"CALL get_employee_by_departme
nt({departmentName})").ToList();

foreach (GetEmployeeByDepartmentResult item in data5)
{
   Console.WriteLine($"{item.T02F01} {item.T02F02} {item.T02F03}");
}
```

If it does not return a table we can use

```
context.Database.ExecuteSqlRaw();
```

### Transaction

```
using (var transaction = context.Database.BeginTransaction())
 {
    context.Database.ExecuteSqlRaw("Delete from t03");

    transaction.Rollback();
 }
```

To commit

```
transactin.Commit();
```

To create savepoint

```
transaction.CreateSavePoint(name);
```

To rollback to savepoint

```
transaction.RollbackToSavepoint(name);
```

# Dapper

Dapper is a micro ORM not full fledge ORM like EFCore

It does not manage migresion, does not track change

It maps the query result to C# POCO but does not auto generate query from linq we need to use raw sql

It is very fast compared to EFCore but sligthly slower than ADO.NET

Dapper is an excellent choice for applications that require simple and efficient data access. It is easy to use, lightweight, and high-performing, making it a perfect option for developers who need fast data access without the overhead of more complex ORM solutions.

To use dapper first we need to create POCO class with exact same name as query result

To create connection

```
await using var connection = new MySqlConnection(connectionString);
```

To start transaction

```
await using var transaction = await connection.BeginTransactionAsync();
await transaction.CommitAsync();
await transaction.RollbackAsync();
```

To fetch multiple rows

```
string getT01 = "SELECT * FROM T01";
var allDepartments = await connection.QueryAsync<T01>(getT01, transacti
on: transaction);
```

To fetch single row

```
string getOne = "SELECT * FROM T01 WHERE T01F01 = @DepartmentId";
int idToFind = 1;
var singleDepartment = await connection.QueryFirstOrDefaultAsync<T01>
(getOne, new {DepartmentId = idToFind}, transaction: transaction);
```

To perform join query

<T01, T02, T02>: First two is input model and last is output model

```
string q1 = "select * from t01 inner join t02 on t01f01 = t02f08";
```

```
IEnumerable<T02> emps1 = await connection.QueryAsync<T01, T02, T02>
(q1, (t01, t02) ⇒
            {
               t02.T01 = t01;
               return t02;
            }, transaction: transaction, splitOn: "T02F01");
```

To execute two different query

```
string q2 = @"
               SELECT * FROM T01;
               SELECT * FROM T02;
            ";
using(var gridReader = connection.QueryMultiple(q2, transaction: transacti
on))
{
   var t01s = gridReader.Read<T01>();
   var t02s = gridReader.Read<T02>();
}
```

To insert row

```
var newDepartment = new T01
            {
               T01F01 = 12,
               T01F02 = "dotnet",
               T01F03 = "Mumbai"
            };
string sqlInsert = "INSERT INTO T01(T01F01, T01F02, T01F03) VALUES (@T
01F01, @T01F02, @T01F03)";
int rowAffected = await connection.ExecuteAsync(sqlInsert, newDepartmen
t, transaction: transaction);
```

To update

```
var updatedDepartment = new T01
            {
               T01F01 = 12,
               T01F02 = "C#",
               T01F03 = "Anand"
            };
string sqlUpdate = "UPDATE T01 SET T01F02 = @T01F02, T01F03 = @T01F
03 WHERE T01F01 = @T01F01";
int rowUpdated = await connection.ExecuteAsync(sqlUpdate, updatedDepa
rtment, transaction: transaction);
```

To delete

```
int idToDelete = 12;
string sqlDelete = "DELETE FROM T01 WHERE T01F01 = @T01F01";
int rowDeleted = await connection.ExecuteAsync(sqlDelete, new { T01F01 =
idToDelete }, transaction: transaction);
```

To execute SP

```
string spName = "GET_EMPLOYEE_BY_DEPARTMENT";
var employees = await connection.QueryAsync<T02>(
               spName,
               new { p_dept_name = "Engineering" },
               commandType: System.Data.CommandType.StoredProcedure,
               transaction: transaction
               );
```

To execute SP with out parameter

```
var parameters = new DynamicParameters();
```

```
parameters.Add("@p_count", dbType: System.Data.DbType.Int32, direction: System.Data.ParameterDirection.Output);

string spName2 = "count_emp";

await connection.ExecuteAsync(spName2, parameters, commandType: System.Data.CommandType.StoredProcedure, transaction: transaction);

int count = parameters.Get<int>("@p_count");
```

# ORMLite

To create connection

```
var dbFactory = new OrmLiteConnectionFactory(connectionString, MySqlDialect.Provider);

using (IDbConnection db = dbFactory.OpenDbConnection())
{
   // use db
}
```

To read entire table

```
List<T01> t01s = db.Select<T01>();
```

To fetch single row by id

```
var engineeringDepartment = db.SingleById<T01>(1);
```

To query

```
List<T01> mumbaiDepartments = db.Select<T01>(d ⇒ d.T01F03 == "Mumb
ai");
```

To insert

```
var newDepartment = new T01
            {
               T01F01 = 12,
               T01F02 = ".NET",
               T01F03 = "Anand"
            };

long departmentId = db.Insert(newDepartment);
```

To update

```
T01 forUpdate = db.Single<T01>(d ⇒ d.T01F01 == 2);
if(forUpdate != null)
{
   forUpdate.T01F02 = "Marketing";
   db.Update(forUpdate);
}
```

To delete by Id

```
db.DeleteById<T01>(12);
```

To delete multiple

```
db.Delete<T01>(d ⇒ d.T01F03 == "Mumbai");
```

To execute SP

```
List<GetEmployeeByDepartmentResult> result = db.SqlList<GetEmployeeByDepartmentResult>("CALL GET_EMPLOYEE_BY_DEPARTMENT(@p_dept_name)", new { p_dept_name = "Engineering" });
```

To execute SP with out parameter

```
using (var cmd = db.SqlProc("count_emp"))
{
    IDbDataParameter outputParamProc = cmd.AddParam("p_count", direction: ParameterDirection.Output, dbType: DbType.Int32);

    cmd.ExecuteNonQuery();

    int count = (int)outputParamProc.Value;

    Console.WriteLine(count);
}
```

To create transaction

```
using(IDbTransaction transaction = db.OpenTransaction())
{
    // Operation
    transaction.Rollback();
}
```

# Security

# Hashing

To create SHA256 hash

```
using (SHA256 sha256 = SHA256.Create())
{
    byte[] bytes = sha256.ComputeHash(System.Text.Encoding.UTF8.GetBytes(input));
    return Convert.ToHexString(bytes);
}
```

However we should use a random salt while hashing password and store it alongside with hashedpassword by concating it

## To hash password

Saltsize should be 16byte (128bit)

Hashsize should be 32byte (256bit)

Here iterations should be more that 10000 to make hashing slower so attaker can't `brute force` passwords. This is defined in pbkdf2 standard which is implemented by Rfc2898DeriveBytes

```
public static string HashPassword(string password)
{
    // Generate a salt
    byte[] salt = RandomNumberGenerator.GetBytes(SaltSize);

    // Derive the hash
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt, Iterations, HashAlgorithmName.SHA256);
    byte[] hash = pbkdf2.GetBytes(HashSize);

    // Combine salt and hash
    byte[] hashedBytes = new byte[SaltSize + HashSize];
    Array.Copy(salt, 0, hashedBytes, 0, SaltSize);
    Array.Copy(hash, 0, hashedBytes, SaltSize, HashSize);
```

```
    // Return as base64 string
    return Convert.ToBase64String(hashedBytes);
}
```

## To check password

Here we are using FixedTimeEquals because it compare each byte even if find that bytes are not matching so it prevents attacker to make time analysis attack.

Time analysis attack: Attacker try to find return time and try to analyze which is first byte which does not match and get some insights

```
public static bool VerifyPassword(string password, string storedHash)
{
    // Extract the bytes
    byte[] hashedBytes = Convert.FromBase64String(storedHash);

    // Get the salt
    byte[] salt = new byte[SaltSize];
    Array.Copy(hashedBytes, 0, salt, 0, SaltSize);

    // Get the hash
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt, Iterations, HashAl
gorithmName.SHA256);
    byte[] hash = pbkdf2.GetBytes(HashSize);

    // Compare the results
    // Use a time-constant comparison to prevent timing attacks
    return CryptographicOperations.FixedTimeEquals(hash, hashedBytes.As
Span(SaltSize));
}
```

# AES (Symmetric hashing)

Use AesGcm as it ensure integrity and confidentiality

Use random key and IV(Initialization vector)

Tag is used to ensure integrity it throw error if cipher text is altered

## Encryption

```
byte[] key = RandomNumberGenerator.GetBytes(32); // 256 bit key
byte[] nonce = RandomNumberGenerator.GetBytes(AesGcm.NonceByteSizes.MaxSize);
byte[] cipherText = new byte[plainTextBytes.Length];
byte[] tag = new byte[AesGcm.TagByteSizes.MaxSize]; // Authentication tag

aesGcm.Encrypt(nonce, plainTextBytes, cipherText, tag);

byte[] forStorage = new byte[nonce.Length + tag.Length + cipherText.Length];
Array.Copy(nonce, 0, forStorage, 0, nonce.Length);
Array.Copy(tag, 0, forStorage, nonce.Length, tag.Length);
Array.Copy(cipherText, 0, forStorage, nonce.Length + tag.Length, cipherText.Length);
```

## Decryption

```
byte[] extractedNonce = new byte[AesGcm.NonceByteSizes.MaxSize];
byte[] extractedTag = new byte[AesGcm.TagByteSizes.MaxSize];
byte[] extractedCipherText = new byte[forStorage.Length - extractedNonce.Length - extractedTag.Length];

Array.Copy(forStorage, 0, extractedNonce, 0, extractedNonce.Length);
Array.Copy(forStorage, extractedNonce.Length, extractedTag, 0, extractedTag.Length);
```

```
Array.Copy(forStorage, extractedNonce.Length + extractedTag.Length, ext
ractedCipherText, 0, extractedCipherText.Length);

byte[] decryptedBytes = new byte[extractedCipherText.Length];

aesGcm.Decrypt(extractedNonce, extractedCipherText, extractedTag, decr
yptedBytes);

string decrypted = Encoding.UTF8.GetString(decryptedBytes);
```

# RSA

Rsa is used for asymmetric key cryptography

Rsa is slow it should only use to encrypt small data, mostly keys

For large data use AES

## Encrypt

```
static byte[] RsaEncrypt(string plainText, RSAParameters publicKey)
{
   using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider
())
   {
      rsa.ImportParameters(publicKey);
      byte[] bytes = Encoding.UTF8.GetBytes(plainText);
      return rsa.Encrypt(bytes, false);
   }
}
```

## Decrypt

```
static byte[] RsaDecrypt(byte[] cipherText, RSAParameters privateKey)
{
   using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider
())
   {
      rsa.ImportParameters(privateKey);
      return rsa.Decrypt(cipherText, false);
   }
}
```

### To generate random keys

```
using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(20
48))
{
   RSAParameters publicKey = rsa.ExportParameters(false);
   RSAParameters privateKey = rsa.ExportParameters(true);
}
```

## Digital Signature

DS is used to ensure integrity

```
using(RSA rsa = RSA.Create(2048))
{
   byte[] data = Encoding.UTF8.GetBytes(original);
   byte[] sign = rsa.SignData(data, HashAlgorithmName.SHA256, RSASigna
turePadding.Pkcs1);

   Console.WriteLine($"Sign: {Convert.ToBase64String(sign)}");

   bool isVerified = rsa.VerifyData(data, sign, HashAlgorithmName.SHA256,
RSASignaturePadding.Pkcs1);
```

```
    Console.WriteLine($"Signature verification status : {isVerified}");
}
```

## Best practise

1. Never use own crypto use standard, well-known, battle-tested algorithms like System.Security.Cryptography
2. Use modern algorithm like AES-256, RSA, SHA-256m PBKDF2
3. Securely manage keys:

   Never hard code keys

   Don’t store it in configuration files

   Ex: use Secret Manager

   which store json file in separate file in users/appdata so it never track by version control

   To access it

   ```
   var configuration = builder.Build();
   Console.WriteLine($"Secret Key: {configuration["APIKEY"]}");
   ```

   To use it while scaffold-dbcontext it will automatically find from configuration file and secrets file

   ```
   scaffold-dbcontext "Name=DefaultConnection" Microsoft.EntityFra
   meworkCore.SqlServer -o Models
   ```
4. Use authenticated encryption: Like AesGcm which provide integrity with confidentiality
5. Salt password: Use unique, cryptographically random salt for each user. Store the salt alongside the hash in database. This prevents attackers from using pre-computed rainbow tables
6. For key use cryptographically secure random number like System.Security.Cryptography.RandomNumberGenerator

# Dynamic Type

The *dynamic language runtime* (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR). Dynamic type let’s us use variable without declaring its type

Its type is decided at run time

It is different from object as object needs boxing and unboxing while dynamic type don’t need it

But it doesn’t provide compile time warnings or errors if type mismatch it will throw RuntimeBinderException at run time

Ex: It will work without exception

```
dynamic a = 5;
a = "HELLO";
```

## Usage

- **Interoperability with Dynamic Languages**: It's excellent for working with libraries from languages like Python or Ruby.
- **Working with COM Objects**: Interacting with technologies like Microsoft Office automation becomes much simpler.
- **Parsing Dynamic Data**: Handling data from sources like JSON or XML where the structure isn't known beforehand. You can navigate the parsed object without creating a rigid class structure first.
- **Simplifying Reflection**: It can sometimes provide a cleaner syntax than using complex reflection code to invoke methods dynamically.

Ex:

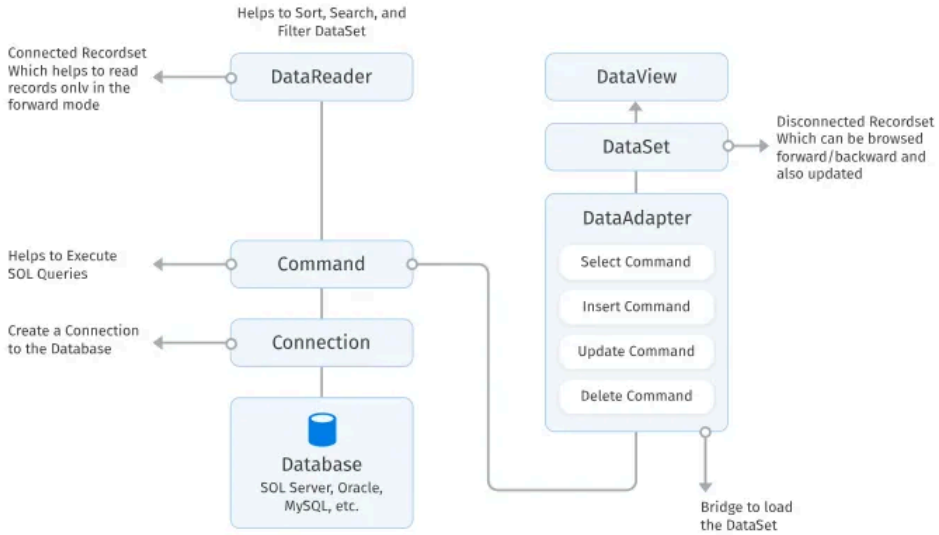
```
string jsonString = @"{
  'user': { 'id': 123, 'name': 'Alex', 'roles': ['Admin', 'Editor'] },
  'lastLogin': '2025-10-15T14:30:00Z'
}";

dynamic jsonObj = JsonConvert.DeserializeObject<dynamic>(jsonString);
Console.WriteLine($"User ID: {jsonObj.user.id}");
```

### Limitation

- No compile time error checking
- performance overhead : DLR has to perform work at run time to figure out what we are trying to call
- Loss of intellisense and tooling support
- reduce code readability and maintainability

# ADO.NET

Architecture of ADO.NET




DataReader is forward only, read only, stream that can be used to read row by row, it is connection oriented

DataAdapter uses cached data, it can be filled and updated to original DB

Use Command when

- **want to run a single query** or **command** (like `INSERT`, `UPDATE`, `DELETE`, or `SELECT`).
- **don’t need to store** or **cache** results in memory.
- want **fine-grained control** over execution (parameters, transactions, etc.).

Use DataReader when

- need **fast and efficient reading** of large result sets.
- **only need to display** or process data — not modify it.
- can **keep the connection open** while reading.

Use DataAdapter when

- want **disconnected data access** (load data, close connection, work offline).
- need to **fetch, modify, and update** data back to the database.
- want **automatic generation of commands** (`InsertCommand`, `UpdateCommand`, `DeleteCommand`).

# DataAdapter

## To fill datatable

```
MySqlConnection connection = new MySqlConnection(connectionString);

MySqlDataAdapter adapter = new MySqlDataAdapter(query, connection);

MySqlCommandBuilder builder = new MySqlCommandBuilder(adapter);

connection.Open();

DataTable T01 = new DataTable();

adapter.Fill(T01);
```

## Delete

```
foreach(DataRow row in T01.Rows)
{
   if ((string)row["T01F02"] == "iOS")
   {
      row.Delete();
   }
}
adapter.Update(T01);
```

### Insert

```
T01.Rows.Add(10, "Android", "Anand");
adapter.Update(T01);
```

### Update

```
foreach (DataRow row in T01.Rows)
{
   if ((int)row["T01F01"] == 1)
   {
      row["T01F03"] = "Delhi";
   }
}
adapter.Update(T01);
```

### Read

```
foreach (DataRow row in T01.Rows)
{
   Console.WriteLine($"{row["T01F01"]} {row["T01F02"]} {row["T01F0
3"]}");
}
```

## DataReader

```
MySqlConnection connection = new MySqlConnection(connectionString);

MySqlCommand command = new MySqlCommand(query, connection);

connection.Open();
```

```
MySqlDataReader reader = command.ExecuteReader();

if(reader.HasRows)
{
   while(reader.Read())
   {
      Console.WriteLine($"{reader.GetInt32(0)}, {reader.GetString(1)} {reade
r.GetString(2)}, {reader.GetString(3)}");
   }
}
```

# Commands

## Raw query

```
string insertQuery = "INSERT INTO T01(T01F01, T01F02, T01F03) VALUES
(@id, @name, @location)";

connection.Open();

MySqlCommand command = new MySqlCommand(insertQuery, connectio
n);

command.Parameters.AddWithValue("@id", 11);
command.Parameters.AddWithValue("@name", "iOS");
command.Parameters.AddWithValue("@location", "delhi");

int rows = command.ExecuteNonQuery();

connection.Close();

Console.WriteLine($"Affected rows : {rows}");
```

## SP

```
connection.Open();

MySqlCommand command2 = new MySqlCommand("GET_EMPLOYEE_BY_
DEPARTMENT", connection);


command2.CommandType = System.Data.CommandType.StoredProcedur
e;

command2.Parameters.AddWithValue("p_dept_name", "Engineering");

MySqlDataReader reader = command2.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine($"{reader.GetInt32(0)} {reader.GetString(1)} {reader.G
etString(2)}");
}

connection.Close();
```

## SP with out parameter

```
connection.Open();

MySqlCommand command3 = new MySqlCommand("count_emp", connect
ion);

command3.CommandType = System.Data.CommandType.StoredProcedur
e;

command3.Parameters.Add("p_count", MySqlDbType.Int32);
command3.Parameters["p_count"].Direction = System.Data.ParameterDire
ction.Output;
```

```
command3.ExecuteNonQuery();

connection.Close();

int count = (int)command3.Parameters["p_count"].Value;
Console.WriteLine(count);
```

## Transaction

```
connection.Open();

MySqlTransaction transaction = connection.BeginTransaction();

try
{
   MySqlCommand insertCommand = new MySqlCommand("", connection,
transaction);
   insertCommand.CommandText = "DELETE FROM T01";
   insertCommand.ExecuteNonQuery();

   throw new Exception("Somethind went wrong");

   transaction.Commit();

}
catch (Exception ex)
{
   transaction.Rollback();
   Console.WriteLine("Transaction Rollbacked" + ex.Message);
}

connection.Close();
```