1. Testing

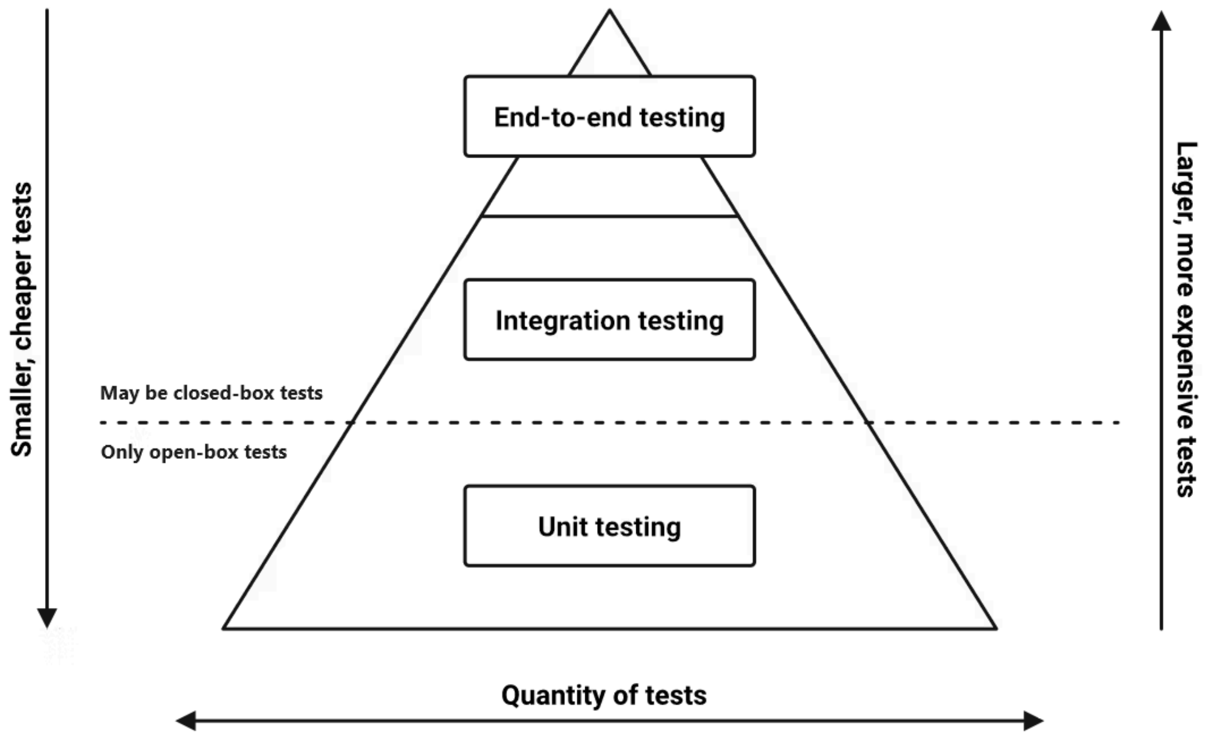
Testing is crucial part of software design. There are number of different kind of testing techniques perform by either programmers or testers.

We can divide tests based on knowledge of internal workings:

- black-box testing
- white-box testing
- gray-box testing

We can divide tests into:

- manual tests
- automated tests
    - unit tests
    - integration tests
    - e2e tests



We can divide tests by the type of test:

- load testing
- performance tests
- usability tests
- compatibility tests
- regression testing
- and much more...

A unit can be an individual function, method, procedure, module, or object. The primary goal of unit testing is to ensure that each unit of the software performs as designed.

Unit tests are typically written and maintained by developers. These tests are executed frequently throughout the development process to catch bugs early, facilitate code refactoring, and ensure that recent changes haven't adversely affected existing functionality.

A unit test framework, specific to the programming language being used (e.g., JUnit for Java, NUnit for .NET, or PyTest for Python), provides a structured way to define test cases, execute tests, and report results.

Characteristics of unit tests

- **Isolation**: Unit tests are designed to test units of code in isolation from the rest of the system. This is often achieved by using mock objects, stubs, or fakes to simulate the behavior of complex dependencies such as databases, file systems, or external services.
- **Simplicity**: Each test should focus on a single aspect of the unit's behavior, making it clear what is being tested and what the expected outcome is.
- **Automatability**: Unit tests are automated, meaning they can be run quickly and frequently without manual intervention. This automation is crucial for continuous integration and continuous delivery (CI/CD) pipelines.
- **Repeatability**: A unit test can be run any number of times in any environment with the expectation of the same outcome each time, assuming no changes have been made to the unit under test.

11.2. Why we write tests?

1. ==**Catch Bugs Early and Regression Testing**==: Unit tests help identify bugs at an early stage of the development process. By testing individual units of code independently, developers can pinpoint errors before they become integrated into the larger system, where they are often harder and more expensive to diagnose and fix.
2. **Facilitate Changes**: With a comprehensive suite of unit tests, developers can make changes to the codebase confidently, knowing that tests will reveal if their changes have inadvertently broken existing functionality. This is especially valuable in larger projects or those with multiple contributors, as it helps maintain the stability of the software over time.
3. ==**Improve Code Quality**==: The process of writing unit tests encourages developers to write cleaner, more modular code. Units of code need to be well-isolated to be testable, which naturally leads to a more modular architecture. This not only makes the codebase easier to understand but also facilitates reuse of code.
4. **Documentation**: Unit tests serve as a form of live documentation. They demonstrate how the code is supposed to be used and what its expected behavior is under various conditions. New developers or contributors can look at the unit tests to quickly get up to speed with the codebase.
5. ==**Facilitate Design**==: Writing tests often helps in the software design process. The need to keep the code testable can guide developers towards better design choices, encouraging them to think about how components interact with each other and how to effectively encapsulate behavior.
6. **Refactoring Confidence**: Unit tests provide a safety net that allows developers to refactor code with confidence. Refactoring is the process of changing the internal structure of the code without altering its external behavior. Unit tests help ensure that refactoring efforts do not introduce new bugs.
7. **Continuous Integration (CI)**: Unit tests are integral to CI practices, where code changes are automatically built, tested, and merged into a shared repository multiple times a day. A robust unit test suite enables rapid feedback on code changes, reducing the integration issues and allowing for faster development cycles.

11.3. Naming tests

Unit test naming strategies are crucial for ensuring that test suites are easily understandable and maintainable.

Good test names can quickly convey the purpose of the test, the conditions under which it runs, and what outcome is expected. Here are several common strategies for naming unit tests, along with an explanation for why test names often end up being quite long:

1. **MethodName_StateUnderTest_ExpectedBehavior**
   This strategy involves starting with the name of the method being tested, followed by the condition or state under which it's being tested, and finally the expected behavior or outcome. This approach makes it very clear what aspect of the method is being tested and under what circumstances.
2. **MethodName_WhenStateUnderTest_ShouldExpectedBehavior**
   When incorporating the word "should" into a unit test naming convention, you articulate the expected behavior or outcome in a way that reads naturally and clearly states what the code under test should do under certain conditions.

For example:

- **calculateTotal_WhenDiscountApplied_ShouldReduceTotalAmount**

2. Important C# Features

1.1. Lambda expressions

Lambda expressions in C# are a **concise way to represent anonymous methods (methods with no names) or expressions**. They are especially useful for writing short pieces of code that can be passed as arguments or returned from other methods, commonly used with LINQ (Language Integrated Query) and when working with events or callback methods.

Historical Context

Lambda expressions were introduced in C# 3.0 (released with .NET Framework 3.5 in 2007) as part of the language's evolution toward supporting functional programming concepts. Before lambdas, developers used anonymous methods or explicit delegate instantiation, which were more verbose and less readable.

Syntax of Lambda Expressions

```
(argument-list) => expression
```

or

```
(argument-list) => { statement(s); }
```

The lambda operator => is read as "goes to" and separates the input parameters from the lambda body.

Expression vs. Statement Lambdas

Expression Lambda

An expression lambda has a single expression and implicitly returns its result:

```
var method= () => 42
```

Statement Lambda

A statement lambda contains a code block with one or more statements and requires an explicit return statement if returning a value:

```
// Statement lambda - multiple statements with explicit return
var method2 = (x, y) =>
{
    int result = x + y;
    return result;
}
```

Type Inference in Lambda Expressions

C# can often infer the parameter types of lambda expressions from their context:

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        // Type inference in action
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

        // The compiler infers that 'n' is an int
        var evenNumbers = numbers.FindAll(n => n % 2 == 0);

        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

```csharp
using System;

class Program
{
    static void Main(string[] args)
    {
        // Action with no parameters
        Action printHello = () => Console.WriteLine("Hello");
        printHello();

        // Action with one parameter
        Action<string> greet = name => Console.WriteLine($"Hello, {name}!");
        greet("World");

        // Action with two parameters
        Action<int, int> addAndPrint = (x, y) => Console.WriteLine($"{x} + {y} = {x + y}");
        addAndPrint(5, 7);
    }
}
```

1.3. Function

The Func delegate type is used to represent a method that returns a value and may take zero or more input parameters. The last type parameter specifies the return type.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Func with no input parameters, returning an integer
        Func<int> getRandomNumber = () => new Random().Next(1, 100);
        Console.WriteLine(getRandomNumber());
        // Func with one input parameter, returning a string
        Func<int, string> numberToString = number => $"Number is: {number}";
        Console.WriteLine(numberToString(42));
        // Func with two input parameters, returning their sum
        Func<int, int, int> add = (x, y) => x + y;
        Console.WriteLine(add(5, 3));
    }
}
```

1.4. Predicate Delegate

A Predicate is a special kind of Func that always returns a boolean value. It's commonly used for filtering operations.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // Predicate that checks if a number is even
        Predicate<int> isEven = num => num % 2 == 0;

        // Using the predicate with the FindAll method
        List<int> evenNumbers = numbers.FindAll(isEven);

        // Alternatively, we can pass the lambda directly
        List<int> oddNumbers = numbers.FindAll(num => num % 2 != 0);

        Console.WriteLine("Even numbers:");
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nOdd numbers:");
        foreach (var number in oddNumbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

1.5. Closures in Lambda Expressions

A closure allows a lambda expression to capture and use variables from its containing scope. This powerful feature enables more flexible and dynamic code:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Variable in the outer scope
        int factor = 10;

        // This lambda captures the 'factor' variable
        Func<int, int> multiplier = x => x * factor;

        Console.WriteLine(multiplier(5));  // Outputs: 50

        // Changing the captured variable affects the lambda
        factor = 20;
        Console.WriteLine(multiplier(5));  // Outputs: 100

        // Creating a counter using closure
        Func<int> createCounter()
        {
            int count = 0;
            return () => ++count;
        }

        var counter = createCounter();
        Console.WriteLine(counter());  // Outputs: 1
        Console.WriteLine(counter());  // Outputs: 2
        Console.WriteLine(counter());  // Outputs: 3
    }
}
```

3. Extension Methods in C#

Extension methods are one of the most powerful and practical features in C#. They allow you to add new functionality to existing types without modifying their source code. This lesson covers everything your students need to understand about extension methods, from basic concepts to advanced applications.

3.1. What Are Extension Methods?

Extension methods let you "extend" existing types—even ones you didn't create—by adding new methods that appear as if they were part of the original class. When you call an extension method, it looks just like you're calling an instance method on that object.

Think of extension methods as a way to customize or enhance classes without inheritance or modifying their source code. This is particularly valuable when working with:

- Classes you can't modify (like string or DateTime from the .NET Framework)
- Sealed classes that can't be inherited from
- Interfaces where you want to provide default implementations

3.2. How Extension Methods Work

Behind the scenes, extension methods are actually static methods that the compiler treats in a special way. When you write:

```
string text = "Hello";
bool isEmpty = text.IsNullOrEmpty();
```

The compiler transforms this into:

```
string text = "Hello";
bool isEmpty = StringExtensions.IsNullOrEmpty(text);
```

This is why extension methods can't access private members of the extended type—they're just static methods with syntactic sugar to make them appear as instance methods.

3.3. Creating Extension Methods

The lesson covers the exact syntax rules:

- Define a static class to contain your extension methods
- Create static methods with the first parameter using the `this` keyword
- The type of this first parameter is what you're extending

3.4. Real-World Applications

I've included detailed examples showing how extension methods are used for:

1. **Interface Extensions** - Adding default implementations to interfaces
2. **Method Chaining** - Creating fluent interfaces for more readable code
3. **LINQ-Style Extensions** - Creating your own LINQ-like extension methods
4. **Primitive Type Extensions** - Adding useful methods to basic types like int, decimal, DateTime
5. **Building Fluent APIs** - Using extension methods to create expressive APIs

4. Anonymous Types in C#

4.1. Introduction to Anonymous Types

Anonymous types in C# are a feature that **allow you to create simple class types on the fly, without having to explicitly define a class beforehand**. Introduced in C# 3.0 alongside LINQ and lambda expressions, anonymous types make it easy to define simple data-carrying objects without the overhead of creating named class definitions.

Anonymous types are particularly useful when you need to create temporary objects that hold a small set of properties, especially in LINQ queries where you often want to reshape your data by selecting specific properties or computed values.

4.2. Why Anonymous Types Matter

Before anonymous types were introduced, developers faced several challenges:

1. **Excessive boilerplate code** for simple data-carrying classes
2. **Unnecessary named types** for objects that were only used temporarily
3. **Difficulty projecting data** in LINQ queries without creating multiple class definitions
4. **Cumbersome reshaping of data** when only certain properties were needed

Anonymous types provide a solution by:

- Reducing code verbosity
- Eliminating the need for throwaway class definitions
- Making LINQ projections more elegant
- Allowing you to focus on the data rather than the structure

4.3. Syntax and Creation of Anonymous Types

Anonymous types are created using the `new` keyword followed by curly braces containing property initializers:

```csharp
using System;

class Program
{
    static void Main()
    {
        // Creating an anonymous type with two properties
        var person = new { Name = "John Doe", Age = 30 };

        // Accessing properties of the anonymous type
        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");

        // Nesting anonymous types
        var employee = new
        {
            PersonalInfo = new { Name = "Jane Smith", Age = 28 },
            Position = "Software Developer",
            Salary = 85000
        };

        // Accessing nested properties
        Console.WriteLine($"Employee: {employee.PersonalInfo.Name}");
        Console.WriteLine($"Position: {employee.Position}");
    }
}
```

4.4. Key Characteristics of Anonymous Types

Anonymous types have several important characteristics to understand:

1. Read-Only Properties

Properties of anonymous types are read-only; you cannot modify them after creation:

```
var person = new { Name = "John", Age = 30 };

// This will not compile
// person.Age = 31;  // Error: Cannot modify the read-only property 'Age'
```

2. Strong Typing

Despite being "anonymous," these types are strongly typed at compile time:

```
var person = new { Name = "John", Age = 30 };

// Compiler knows the exact types of the properties
string name = person.Name;  // Works fine
int age = person.Age;       // Works fine

// This will not compile
// double age = person.Age;  // Error: Cannot implicitly convert type 'int' to 'double'
```

3. Property Names

Property names in anonymous types match the names used in the initializers, but you can specify different names:

```
string fullName = "John Doe";
int yearsOld = 30;

// Property names match variable names
var person1 = new { fullName, yearsOld };
Console.WriteLine($"{person1.fullName}, {person1.yearsOld}");

// Custom property names
var person2 = new { Name = fullName, Age = yearsOld };
Console.WriteLine($"{person2.Name}, {person2.Age}");
```

4. Type Equivalence

Two anonymous types with the same property names and types in the same assembly are considered the same type:

```
var person1 = new { Name = "John", Age = 30 };
var person2 = new { Name = "Jane", Age = 25 };

// These are compatible because they have the same structure
object[] people = new object[] { person1, person2 };

// We can cast back to the anonymous type using the first variable
var personFromArray = (person1.GetType())people[1];
Console.WriteLine(personFromArray.Name);  // Works!

// However, anonymous types with different property order are different types
var personA = new { Name = "John", Age = 30 };
var personB = new { Age = 25, Name = "Jane" };  // Different property order

// This would throw an InvalidCastException at runtime
// var invalid = (personA.GetType())personB;
```

5. ToString(), Equals(), and GetHashCode()

Anonymous types automatically override these methods for better debugging and equality comparison:

```
var person = new { Name = "John", Age = 30 };

// ToString() shows property values
Console.WriteLine(person.ToString());  // { Name = John, Age = 30 }

// Equals() compares property values
var samePerson = new { Name = "John", Age = 30 };
Console.WriteLine(person.Equals(samePerson));  // True

var differentPerson = new { Name = "Jane", Age = 30 };
Console.WriteLine(person.Equals(differentPerson));  // False

// GetHashCode() considers all property values
Console.WriteLine(person.GetHashCode() == samePerson.GetHashCode());  // True
```

4.5. Limitations of Anonymous Types

Despite their convenience, anonymous types have some limitations:

1. Method Return Types

You can't use anonymous types as explicit return types for methods:

```
// This won't compile
// public static { string Name, int Age } GetPerson() // Error
public static object GetPerson()
{
    return new { Name = "John", Age = 30 };
}

// To access properties, you would need:
// 1. Dynamic (loses compile-time checking)
dynamic person = GetPerson();
Console.WriteLine(person.Name);  // Works but no IntelliSense

// 2. Reflection (verbose)
var personObj = GetPerson();
var nameProperty = personObj.GetType().GetProperty("Name");
string name = (string)nameProperty.GetValue(personObj);
```

2. Public API Surface

Anonymous types aren't suitable for public APIs because:

- They're implementation details not visible outside the assembly
- They can't be referenced directly in method signatures
- Their structure could change without warning

3. No Interfaces or Methods

Anonymous types can only contain properties, not methods or interfaces:

```
// This won't compile
// var worker = new { Name = "John", DoWork = () => Console.WriteLine("Working") };
```

4.6. Summary

Anonymous types in C# provide a way to:

1. **Create simple data classes on the fly** without explicit class definitions
2. **Project and reshape data** efficiently in LINQ queries
3. **Reduce boilerplate code** for temporary data structures
4. **Focus on the data properties** that matter for a specific context

Key characteristics to remember:

- Anonymous types have read-only properties
- They automatically implement ToString(), Equals(), and GetHashCode()
- They are strongly typed at compile-time
- They work well with LINQ, lambda expressions, and var
- They have some limitations (can't be used as return types, no methods, etc.)

Anonymous types, alongside lambda expressions, extension methods, and LINQ, form a powerful set of features that make C# a more expressive and concise language for working with data.

5. LINQ - Language Integrated Query

Language Integrated Query (LINQ) is a set of features introduced in C# 3.0 that **adds powerful query capabilities directly into the C# language**. LINQ provides a consistent, expressive, and type-safe way to query and manipulate data from various data sources, including collections, XML documents, relational databases, and web services.

LINQ bridges the gap between the world of objects and the world of data, allowing developers to write queries using familiar C# syntax instead of learning different query languages for different data sources.

5.1. Introduction to LINQ

LINQ (Language Integrated Query) is a set of features introduced in C# 3.0 that **adds powerful query capabilities directly into the C# language**. It provides a consistent, expressive way to query and manipulate data from different sources, including collections, databases, XML, and web services.

Before LINQ, developers had to learn different query languages and APIs for different data sources:

- SQL for relational databases
- XPath/XQuery for XML
- Custom APIs for in-memory collections

LINQ unifies these experiences through a common query syntax and programming model, allowing developers to work with data using a consistent approach regardless of where the data comes from.

5.2. The Evolution and Importance of LINQ

LINQ was introduced in .NET Framework 3.5 and C# 3.0 (released in 2007) and represented a significant paradigm shift in how developers work with data.

Before LINQ

Consider a simple task like finding all even numbers in a list:

```
// Pre-LINQ approach
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
List<int> evenNumbers = new List<int>();

foreach (int number in numbers)
{
    if (number % 2 == 0)
    {
        evenNumbers.Add(number);
    }
}
```

With LINQ

Now look at the same operation using LINQ:

```
// LINQ approach
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers = numbers.Where(n => n % 2 == 0);
```

The LINQ approach is:

- More declarative (specifying what to do, not how to do it)
- Shorter and more readable
- Focused on the result rather than the implementation details

5.3. LINQ Architecture and Providers

LINQ is built on a provider model that allows it to work with different data sources:

1. **LINQ to Objects**: For querying in-memory collections that implement IEnumerable<\T>
2. **LINQ to SQL**: For querying SQL Server databases
3. **LINQ to Entities**: For querying databases using Entity Framework
4. **LINQ to XML**: For working with XML documents
5. **LINQ to DataSet**: For querying ADO.NET DataSets
6. **Custom LINQ Providers**: Third-party providers for services like Azure, MongoDB, etc.

Each provider translates LINQ operations into operations appropriate for the specific data source.

5.4. Two Syntaxes: Query Syntax vs. Method Syntax

LINQ offers two different syntactic styles for writing queries:

1. Query Syntax (similar to SQL)

```
// Query syntax
var evenNumbers = from n in numbers
                  where n % 2 == 0
                  select n;
```

2. Method Syntax (method chaining)

This syntax utilize mostly extension methods and lambdas.

```
// Method syntax
var evenNumbers = numbers.Where(n => n % 2 == 0);
```

Both syntaxes are functionally equivalent. The compiler translates query syntax into method syntax during compilation. Most operations can be expressed in either syntax, but some complex operations are only available in method syntax.

5.5. Which Syntax to Choose?

When to use Query Syntax:

- For queries with multiple operations (where, orderby, groupby, join)
- When the query is complex and readability is important
- When the query resembles a traditional SQL query structure
- When using let clauses or into continuations

Example:

```
var results = from c in customers
              join o in orders on c.ID equals o.CustomerID
              where o.OrderDate > DateTime.Now.AddMonths(-6)
              orderby c.LastName, c.FirstName
              select new { c.FirstName, c.LastName, o.OrderDate, o.TotalAmount };
```

When to use Method Syntax:

- For simple queries with just one or two operations
- When using operations that don't have query syntax equivalents
- When chaining multiple operations is more readable
- When using lambda expressions offers better clarity

Example:

```
var results = customers.Where(c => c.Orders.Any(o => o.TotalAmount > 1000))
                      .OrderByDescending(c => c.Orders.Count)
                      .Take(10);
```

5.6. Core LINQ Operations: Filtering

Where

The Where operation filters a sequence based on a predicate:

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

class Program
{
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 22, GPA = 3.8 },
            new Student { Id = 2, Name = "Bob", Age = 19, GPA = 3.2 },
            new Student { Id = 3, Name = "Charlie", Age = 21, GPA = 3.5 },
            new Student { Id = 4, Name = "Diana", Age = 20, GPA = 3.9 },
            new Student { Id = 5, Name = "Ethan", Age = 23, GPA = 3.0 }
        };

        // Query syntax
        var highPerformers1 = from s in students
                             where s.GPA > 3.5 && s.Age < 22
                             select s;

        // Method syntax
        var highPerformers2 = students.Where(s => s.GPA > 3.5 && s.Age < 22);

        // Using the result
        Console.WriteLine("High performing students under 22:");
        foreach (var student in highPerformers2)
        {
            Console.WriteLine($"Name: {student.Name}, Age: {student.Age}, GPA: {student.GPA}");
        }
    }
}

class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }
}
```

5.7. Core LINQ Operations: Projection

Select

The Select operation transforms elements from one form to another:

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

class Program
{
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice Johnson", Age = 22, GPA = 3.8 },
            new Student { Id = 2, Name = "Bob Smith", Age = 19, GPA = 3.2 },
            new Student { Id = 3, Name = "Charlie Brown", Age = 21, GPA = 3.5 }
        };

        // Query syntax - simple projection
        var names1 = from s in students
                     select s.Name;

        // Method syntax - simple projection
        var names2 = students.Select(s => s.Name);

        // Projection to anonymous type
        var studentSummaries = students.Select(s => new {
            FullName = s.Name,
            YearOfBirth = DateTime.Now.Year - s.Age,
            GradeStatus = s.GPA >= 3.5 ? "Excellent" : "Good"
        });

        Console.WriteLine("Student Summaries:");
        foreach (var summary in studentSummaries)
        {
            Console.WriteLine($"{summary.FullName}, Born: {summary.YearOfBirth}, Status:
{summary.GradeStatus}");
        }

        // Using SelectMany to flatten collections
        List<Course> courses = new List<Course>
        {
            new Course { Id = 101, Name = "Calculus", Students = new List<Student> { students[0],
students[2] } },
            new Course { Id = 102, Name = "Physics", Students = new List<Student> { students[0],
students[1] } }
        };

        // Get all student-course combinations
        var enrollments = courses.SelectMany(
            course => course.Students,
            (course, student) => new {
                CourseName = course.Name,
                StudentName = student.Name
            }
        );

        Console.WriteLine("\nCourse Enrollments:");
        foreach (var enrollment in enrollments)
        {
            Console.WriteLine($"{enrollment.StudentName} is enrolled in {enrollment.CourseName}");
        }
    }
}
```

```
class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }
}

class Course
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Student> Students { get; set; } = new List<Student>();
}
```

5.8. Core LINQ Operations: Ordering

OrderBy, OrderByDescending, ThenBy, ThenByDescending

These operations sort sequences based on one or more keys:

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

class Program
{
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 22, GPA = 3.8, Major = "Computer Science" },
            new Student { Id = 2, Name = "Bob", Age = 19, GPA = 3.2, Major = "Physics" },
            new Student { Id = 3, Name = "Charlie", Age = 21, GPA = 3.5, Major = "Computer Science" },
            new Student { Id = 4, Name = "Diana", Age = 20, GPA = 3.9, Major = "Mathematics" },
            new Student { Id = 5, Name = "Ethan", Age = 23, GPA = 3.0, Major = "Physics" }
        };

        // Query syntax - ordering
        var orderedStudents1 = from s in students
                              orderby s.Major, s.GPA descending
                              select s;

        // Method syntax - ordering
        var orderedStudents2 = students
            .OrderBy(s => s.Major)
            .ThenByDescending(s => s.GPA);

        Console.WriteLine("Students ordered by Major, then GPA (descending):");
        foreach (var student in orderedStudents2)
        {
            Console.WriteLine($"{student.Name}: {student.Major}, GPA: {student.GPA}");
        }

        // Dynamic ordering
        string orderField = "Age"; // This could come from user input
        bool ascending = false;    // This could come from user input

        // Dynamic ordering using reflection
        IEnumerable<Student> dynamicOrdered;
        if (ascending)
        {
            dynamicOrdered = students.OrderBy(s =>
                s.GetType().GetProperty(orderField).GetValue(s));
        }
        else
        {
            dynamicOrdered = students.OrderByDescending(s =>
                s.GetType().GetProperty(orderField).GetValue(s));
        }

        Console.WriteLine($"\nStudents ordered by {orderField} ({(ascending ? "ascending" :
"descending")}):");
        foreach (var student in dynamicOrdered)
        {
            Console.WriteLine($"{student.Name}: {student.Age} years old, GPA: {student.GPA}");
        }
    }
}

class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public double GPA { get; set; }
    public string Major { get; set; }
}
```

5.9. Core LINQ Operations: Grouping

GroupBy

The GroupBy operation organizes a sequence into groups based on specified keys:

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

class Program
{
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 22, Major = "Computer Science" },
            new Student { Id = 2, Name = "Bob", Age = 19, Major = "Physics" },
            new Student { Id = 3, Name = "Charlie", Age = 21, Major = "Computer Science" },
            new Student { Id = 4, Name = "Diana", Age = 20, Major = "Mathematics" },
            new Student { Id = 5, Name = "Ethan", Age = 23, Major = "Physics" },
            new Student { Id = 6, Name = "Frank", Age = 22, Major = "Mathematics" }
        };

        // Query syntax
        var groupedByMajor1 = from s in students
                              group s by s.Major into majorGroup
                              select new {
                                  Major = majorGroup.Key,
                                  Students = majorGroup.ToList(),
                                  Count = majorGroup.Count(),
                                  AverageAge = majorGroup.Average(s => s.Age)
                              };

        // Method syntax
        var groupedByMajor2 = students
            .GroupBy(s => s.Major)
            .Select(g => new {
                Major = g.Key,
                Students = g.ToList(),
                Count = g.Count(),
                AverageAge = g.Average(s => s.Age)
            });

        Console.WriteLine("Students grouped by major:");
        foreach (var group in groupedByMajor2)
        {
            Console.WriteLine($"Major: {group.Major}");
            Console.WriteLine($"Count: {group.Count}, Average Age: {group.AverageAge:F1}");
            Console.WriteLine("Students:");
            foreach (var student in group.Students)
            {
                Console.WriteLine($"  - {student.Name}, {student.Age} years old");
            }
            Console.WriteLine();
        }

        // Multiple grouping keys
        var groupedByMajorAndAge = students
            .GroupBy(s => new { s.Major, AgeGroup = s.Age / 10 * 10 })
            .Select(g => new {
                Major = g.Key.Major,
                AgeRange = $"{g.Key.AgeGroup}s",
                Count = g.Count(),
                Students = g.ToList()
            });
        Console.WriteLine("Students grouped by major and age range:");
        foreach (var group in groupedByMajorAndAge)
        {
            Console.WriteLine($"{group.Major} students in their {group.AgeRange}: {group.Count}");
            foreach (var student in group.Students)
            {
                Console.WriteLine($"  - {student.Name}, {student.Age} years old");
            }
            Console.WriteLine();
        }
    }
}

class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public string Major { get; set; }
}
```

5.10. Core LINQ Operations: Joining

Join and GroupJoin

The Join and GroupJoin operations combine elements from two sequences based on matching keys:

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

class Program
{
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Id = 1, Name = "Alice", Age = 22 },
            new Student { Id = 2, Name = "Bob", Age = 19 },
            new Student { Id = 3, Name = "Charlie", Age = 21 }
        };

        List<Course> courses = new List<Course>
        {
            new Course { Id = 101, Name = "Calculus", Credits = 4 },
            new Course { Id = 102, Name = "Physics", Credits = 3 },
            new Course { Id = 103, Name = "Programming", Credits = 4 }
        };

        List<Enrollment> enrollments = new List<Enrollment>
        {
            new Enrollment { StudentId = 1, CourseId = 101, Grade = "A" },
            new Enrollment { StudentId = 1, CourseId = 102, Grade = "B+" },
            new Enrollment { StudentId = 2, CourseId = 101, Grade = "B" },
            new Enrollment { StudentId = 2, CourseId = 103, Grade = "A-" },
            new Enrollment { StudentId = 3, CourseId = 102, Grade = "A" },
            new Enrollment { StudentId = 3, CourseId = 103, Grade = "A" }
        };

        // Inner Join - Query syntax
        var studentGrades1 = from s in students
                            join e in enrollments on s.Id equals e.StudentId
                            join c in courses on e.CourseId equals c.Id
                            select new {
                                StudentName = s.Name,
                                CourseName = c.Name,
                                e.Grade
                            };

        // Inner Join - Method syntax
        var studentGrades2 = students
            .Join(enrollments,
                  s => s.Id,
                  e => e.StudentId,
                  (s, e) => new { Student = s, Enrollment = e })
            .Join(courses,
                  se => se.Enrollment.CourseId,
                  c => c.Id,
                  (se, c) => new {
                      StudentName = se.Student.Name,
                      CourseName = c.Name,
                      se.Enrollment.Grade
                  });

        Console.WriteLine("Student grades (Inner Join):");
        foreach (var sg in studentGrades2)
        {
            Console.WriteLine($"{sg.StudentName} got {sg.Grade} in {sg.CourseName}");
        }

        // Group Join (similar to LEFT OUTER JOIN in SQL)
        var studentCourses = students
            .GroupJoin(enrollments,
                      s => s.Id,
                      e => e.StudentId,
                      (s, enrollmentsGroup) => new {
                          Student = s,
                          Enrollments = enrollmentsGroup
                      })
            .SelectMany(
                sg => sg.Enrollments.DefaultIfEmpty(),
                (sg, e) => new {
                    StudentName = sg.Student.Name,
                    CourseId = e?.CourseId,
                    Grade = e?.Grade
                })
            .GroupJoin(courses,
                      sg => sg.CourseId,
                      c => c.Id,
                      (sg, coursesGroup) => new {
                          sg.StudentName,
                          CourseName = coursesGroup.FirstOrDefault()?.Name ?? "Not Enrolled",
                          sg.Grade
                      });

        Console.WriteLine("\nStudent courses (Left Outer Join equivalent):");
        foreach (var sc in studentCourses)
        {
            Console.WriteLine($"{sc.StudentName} - {sc.CourseName}: {sc.Grade ?? "N/A"}");
        }
    }
}

class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

class Course
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Credits { get; set; }
}

class Enrollment
{
    public int StudentId { get; set; }
    public int CourseId { get; set; }
    public string Grade { get; set; }
}
```

5.11. Core LINQ Operations: Aggregation

Count, Sum, Min, Max, Average, Aggregate

These operations compute scalar values from a sequence:

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

class Program
{
    static void Main()
    {
        List<Product> products = new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 1200, Category = "Electronics" },
            new Product { Id = 2, Name = "Desk Chair", Price = 250, Category = "Furniture" },
            new Product { Id = 3, Name = "Coffee Maker", Price = 100, Category = "Kitchen" },
            new Product { Id = 4, Name = "Headphones", Price = 150, Category = "Electronics" },
            new Product { Id = 5, Name = "Monitor", Price = 300, Category = "Electronics" }
        };

        // Basic aggregation operations
        int count = products.Count();
        decimal totalValue = products.Sum(p => p.Price);
        decimal maxPrice = products.Max(p => p.Price);
        decimal minPrice = products.Min(p => p.Price);
        decimal averagePrice = products.Average(p => p.Price);

        Console.WriteLine("Product Inventory Summary:");
        Console.WriteLine($"Total Products: {count}");
        Console.WriteLine($"Total Inventory Value: ${totalValue}");
        Console.WriteLine($"Most Expensive Item: ${maxPrice}");
        Console.WriteLine($"Least Expensive Item: ${minPrice}");
        Console.WriteLine($"Average Price: ${averagePrice:F2}");

        // Aggregate - custom aggregation
        string productList = products.Aggregate("Our products: ",
            (list, next) => list + next.Name + ", ",
            result => result.TrimEnd(',', ' ') + ".");

        Console.WriteLine("\n" + productList);

        // Aggregation with grouping
        var categorySummary = products
            .GroupBy(p => p.Category)
            .Select(g => new {
                Category = g.Key,
                Count = g.Count(),
                TotalValue = g.Sum(p => p.Price),
                AveragePrice = g.Average(p => p.Price),
                MostExpensive = g.OrderByDescending(p => p.Price).First().Name
            });

        Console.WriteLine("\nCategory Summary:");
        foreach (var summary in categorySummary)
        {
            Console.WriteLine($"Category: {summary.Category}");
            Console.WriteLine($"  Items: {summary.Count}");
            Console.WriteLine($"  Total Value: ${summary.TotalValue}");
            Console.WriteLine($"  Average Price: ${summary.AveragePrice:F2}");
            Console.WriteLine($"  Most Expensive: {summary.MostExpensive}");
        }
    }
}
```

```
class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

5.12. Core LINQ Operations: Set Operations

Distinct, Union, Intersect, Except

These operations perform set operations on sequences:

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

class Program
{
    static void Main()
    {
        // Sample data
        int[] set1 = { 1, 2, 3, 4, 5, 1, 2 };
        int[] set2 = { 4, 5, 6, 7, 8 };

        // Distinct - removes duplicates
        var distinctSet1 = set1.Distinct();
        Console.WriteLine("Distinct elements in set1:");
        Console.WriteLine(string.Join(", ", distinctSet1));  // 1, 2, 3, 4, 5

        // Union - unique elements from both sets
        var unionSet = set1.Union(set2);
        Console.WriteLine("\nUnion of set1 and set2:");
        Console.WriteLine(string.Join(", ", unionSet));  // 1, 2, 3, 4, 5, 6, 7, 8

        // Intersect - elements common to both sets
        var intersectSet = set1.Intersect(set2);
        Console.WriteLine("\nIntersection of set1 and set2:");
        Console.WriteLine(string.Join(", ", intersectSet));  // 4, 5

        // Except - elements in first set but not in second
        var exceptSet = set1.Except(set2);
        Console.WriteLine("\nElements in set1 but not in set2:");
        Console.WriteLine(string.Join(", ", exceptSet));  // 1, 2, 3

        // Using custom equality comparer for complex objects
        List<Person> group1 = new List<Person>
        {
            new Person { Name = "Alice", Age = 30 },
            new Person { Name = "Bob", Age = 25 },
            new Person { Name = "Charlie", Age = 35 },
            new Person { Name = "Alice", Age = 28 }  // Different age
        };

        List<Person> group2 = new List<Person>
        {
            new Person { Name = "Bob", Age = 25 },   // Same as in group1
            new Person { Name = "Diana", Age = 32 },
            new Person { Name = "Alice", Age = 22 }  // Different age
        };

        // Custom comparer that only compares by Name
        var nameComparer = new PersonNameComparer();

        // Distinct by name
        var distinctPeople = group1.Distinct(nameComparer);
        Console.WriteLine("\nDistinct people by name in group1:");
        foreach (var person in distinctPeople)
        {
            Console.WriteLine($"{person.Name}, {person.Age}");  // Only one "Alice"
        }

        // Union by name
        var unionPeople = group1.Union(group2, nameComparer);
        Console.WriteLine("\nUnion of people by name:");
        foreach (var person in unionPeople)
        {
            Console.WriteLine($"{person.Name}, {person.Age}");  // Each name appears once
        }
    }
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class PersonNameComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name.Equals(y.Name, StringComparison.OrdinalIgnoreCase);
    }

    public int GetHashCode(Person obj)
    {
        return obj.Name.ToLower().GetHashCode();
    }
}
```

5.13. Core LINQ Operations: Element Operations

First, FirstOrDefault, Single, SingleOrDefault, Last, LastOrDefault, ElementAt, ElementAtOrDefault

These operations retrieve specific elements from a sequence:

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

class Program
{
    static void Main()
    {
        List<Product> products = new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 1200, Category = "Electronics" },
            new Product { Id = 2, Name = "Desk Chair", Price = 250, Category = "Furniture" },
            new Product { Id = 3, Name = "Coffee Maker", Price = 100, Category = "Kitchen" }
        };

        // First - gets the first element that satisfies a condition
        // Throws InvalidOperationException if no element is found
        try
        {
            Product laptop = products.First(p => p.Category == "Electronics");
            Console.WriteLine($"First electronics product: {laptop.Name}");

            // This will throw an exception
            Product tablet = products.First(p => p.Name.Contains("Tablet"));
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Exception: {ex.Message}");
        }

        // FirstOrDefault - gets the first element or a default value if none found
        Product furniture = products.FirstOrDefault(p => p.Category == "Furniture");
        Console.WriteLine($"First furniture product: {furniture?.Name ?? "None"}");

        Product unknown = products.FirstOrDefault(p => p.Category == "Clothing");
        Console.WriteLine($"First clothing product: {unknown?.Name ?? "None"}");

        // Single - expects exactly one element
        // Throws exception if zero or more than one element is found
        try
        {
            Product kitchenProduct = products.Single(p => p.Category == "Kitchen");
            Console.WriteLine($"The only kitchen product: {kitchenProduct.Name}");

            // This would throw - multiple electronics products
            // Product electronic = products.Single(p => p.Category == "Electronics");
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"Exception: {ex.Message}");
        }

        // Last - gets the last element that satisfies a condition
        Product mostExpensiveElectronic = products
            .Where(p => p.Category == "Electronics")
            .OrderBy(p => p.Price)
            .Last();
        Console.WriteLine($"Most expensive electronic: {mostExpensiveElectronic.Name}");
        // ElementAt - gets the element at a specific index
        Product secondProduct = products.ElementAt(1);
        Console.WriteLine($"Second product: {secondProduct.Name}");

        // ElementAtOrDefault - gets the element or default if index out of range
        Product fifthProduct = products.ElementAtOrDefault(4);
        Console.WriteLine($"Fifth product: {fifthProduct?.Name ?? "Index out of range"}");
    }
}

class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

5.14. Core LINQ Operations: Partitioning

Take, Skip, TakeWhile, SkipWhile, TakeLast, SkipLast

These operations extract subsets of elements from a sequence:

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

class Program
{
    static void Main()
    {
        List<int> numbers = new List<int> { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

        // Take - gets the first n elements
        var firstThree = numbers.Take(3);
        Console.WriteLine("First three numbers:");
        Console.WriteLine(string.Join(", ", firstThree));  // 10, 20, 30

        // Skip - skips the first n elements and returns the rest
        var afterFirst4 = numbers.Skip(4);
        Console.WriteLine("\nAfter skipping first 4 numbers:");
        Console.WriteLine(string.Join(", ", afterFirst4));  // 50, 60, 70, 80, 90, 100

        // TakeWhile - takes elements as long as a condition is true
        var takeLessThan60 = numbers.TakeWhile(n => n < 60);
        Console.WriteLine("\nTake while less than 60:");
        Console.WriteLine(string.Join(", ", takeLessThan60));  // 10, 20, 30, 40, 50

        // SkipWhile - skips elements as long as a condition is true
        var skipLessThan60 = numbers.SkipWhile(n => n < 60);
        Console.WriteLine("\nSkip while less than
```

6. Other Examples

On the following website, you can find more examples of using LINQ:
https://learn.microsoft.com/en-us/dotnet/csharp/linq/
http://linq101.nilzorblog.com/linq101-lambda.php