Using LINQ to Query Collections

How to Sort, Search, and Filter your Arrays and Lists

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

What is LINQ?

LINQ (Language Integrated Query) is a feature introduced in C# 3.0 and the .NET Framework 3.5 that provides a consistent, readable, and concise syntax for querying and manipulating data. It allows developers to perform data operations directly within the programming language, integrating queries into C# code seamlessly. LINQ unifies the way data is accessed regardless of its source, whether it be objects, databases, or other data streams.

The primary purpose of LINQ is to bridge the gap between the data storage and the programming language. Traditional methods of data manipulation often required writing complex and verbose code, especially when dealing with different data sources. LINQ simplifies these tasks by providing a uniform query syntax that developers can use across different types of data collections. This integration into the C# language allows for more intuitive and maintainable code, promoting better coding practices and reducing the likelihood of errors.

Advantages of Using LINQ

Uniform Query Syntax

- LINQ offers a standard way to query different types of data sources. Whether you are working with collections, databases, or other data formats, LINQ provides a consistent syntax. This means you don't need to learn different querying languages for different data sources, making it easier to switch between tasks without the need for context switching.
- LINQ queries are often more readable and concise than their traditional counterparts. By using a declarative syntax, LINQ allows developers to describe what they want to achieve rather than how to achieve it. This leads to cleaner and more understandable code.

Strongly Typed Queries

- LINQ queries are integrated into the C# language and take advantage of the language's type system. This means that queries are checked for type correctness at compile time, reducing runtime errors and improving code reliability.
- Because LINQ is part of the C# language, developers can benefit from IntelliSense in Visual Studio, which provides real-time code suggestions, autocompletion, and documentation as they write queries. This feature helps prevent errors and increases productivity.

Compile-Time Checking

- Since LINQ queries are type-checked at compile time, many potential errors can be detected early in the development process. This allows developers to catch and fix mistakes before running the application, leading to a more robust codebase.

- Because LINQ queries are integrated with the language, they benefit from C#'s refactoring tools. Changes in the structure of the code or data model can be propagated safely through the queries, maintaining consistency and correctness.

Improved Maintainability

- By abstracting the data access logic into a unified syntax, LINQ reduces the complexity of the code. This simplification makes the code easier to read, understand, and maintain over time.
- LINQ queries can often be reused across different parts of an application or even different projects. This promotes code reuse and reduces the amount of duplicated logic, which can help prevent bugs and inconsistencies.

Functional Programming Features

- LINQ allows developers to use a declarative approach to data manipulation, focusing on what data to retrieve rather than how to retrieve it. This can lead to more intuitive and expressive code.
- LINQ supports higher-order functions such as `Select`, `Where`, `OrderBy`, and `GroupBy`, which can be composed and chained together to form complex queries in a readable and maintainable way.

Seamless Data Integration

- LINQ can operate on various data sources including in-memory collections, SQL databases, XML documents, and even remote data services. This versatility allows developers to write consistent query logic regardless of where the data resides.
- LINQ can be combined with other .NET technologies such as Entity Framework for database access, enabling seamless integration of database queries into the application code.

Enhanced Productivity

- By providing a unified and expressive syntax for querying data, LINQ can significantly reduce the amount of code developers need to write. This can speed up the development process and allow developers to focus more on business logic rather than data access code.
- LINQ queries are easier to debug and test compared to traditional data access methods. The integration with C#'s debugging tools allows for step-by-step examination of queries and immediate feedback, making it easier to identify and fix issues.

Query Syntax vs. Method Syntax

When learning LINQ (Language Integrated Query) in C#, it's important to understand that there are two primary ways to write queries: query syntax and method syntax.

While both achieve the same results, method syntax is more prevalent in modern C# code. This is due to its flexibility, expressiveness, and better integration with lambda expressions, making it a preferred choice among developers.

Query Syntax

Query syntax, sometimes referred to as "query comprehension syntax," closely resembles SQL (Structured Query Language). This declarative approach is intuitive for those with a SQL background, as it focuses on what data to retrieve rather than how to retrieve it. It emphasizes clarity and readability, making the code easier to understand at a glance.

Example

Imagine you have a collection of data, such as a list of numbers or objects. Using query syntax, you begin with a `from` clause to specify the data source, followed by clauses like `where`, `select`, `group by`, and others to filter, project, and organize the data.

Here's a narrative to illustrate query syntax:

> "Consider a list of students, and you want to find those who scored above 80 on their exams. Using query syntax, you start with the `from` clause to iterate over each student in the list. Next, you use the `where` clause to filter students with scores above 80. Finally, the `select` clause specifies that you want to retrieve the names of these students. This query reads almost like a sentence, clearly stating your intention to select students with high scores."

Despite its readability, query syntax is less commonly used in new code. This is partly because it has some limitations when compared to method syntax, especially for more complex queries.

Method Syntax

Method syntax is the preferred style in modern C# development. It uses a functional programming approach, chaining methods to form the query. Each method represents a step in the query process and typically takes lambda expressions as parameters. This style is highly expressive and integrates seamlessly with other parts of the C# language, making it powerful for both simple and complex queries.

Example

Returning to our example of filtering students with scores above 80, method syntax would look something like this:

> "You start by calling the `Where` method on the list of students, passing a lambda expression that checks if the student's score is above 80. After filtering the list, you call the `Select` method to project the names of these students. Each method call defines a specific operation, and the chain of methods clearly outlines the steps of the query."

Advantages

Method syntax is particularly valued for its:

- It easily handles complex operations and transformations.
- Lambda expressions allow for concise and powerful query logic.
- It fits naturally with other .NET features and practices.

Industry Trends

In the industry, method syntax has become the de facto standard for several reasons:

- Method syntax is better suited for complex queries that involve multiple operations like joins, grouping, and aggregations.
- The use of lambda expressions makes method syntax more expressive and powerful, enabling developers to write concise and clear code.
- Method syntax aligns well with other functional programming practices within C# and .NET, promoting a consistent coding style.

While query syntax can still be found in educational materials and legacy codebases, method syntax is predominant in modern applications. Its ability to handle intricate query logic and its seamless integration with the rest of the C# language make it the preferred choice for contemporary developers.

Understanding the LINQ Workflow

In LINQ, the workflow typically involves three main stages: defining your data source, creating the query, and executing the query. Understanding this workflow is essential to effectively utilizing LINQ in your C# applications.

Data Sources

The first step in any LINQ query is to identify and define your data source. LINQ can work with various types of data sources, including:

- In-memory collections: Arrays, lists, dictionaries, and other collections.
- Databases: Using LINQ to SQL or Entity Framework to query database tables.
- JSON data: Parsing JSON data into objects or collections.
- Other data sources: Files, streams, and custom collections.

Here's a narrative to illustrate this concept:

> "Imagine you have a list of students, each represented by a Student object containing properties like Name, Age, and Score. This list serves as your data source. LINQ enables you to query this list to retrieve, filter, and manipulate the student data."

Defining the data source in code is straightforward. For example, using a list of Student objects:

```
List<Student> students = new List<Student>
{
    new Student { Name = "Alice", Age = 20, Score = 85 },
    new Student { Name = "Bob", Age = 22, Score = 78 },
    new Student { Name = "Charlie", Age = 21, Score = 90 }
};
```

For JSON data, you would typically deserialize the JSON into a collection of objects. Here's an example using System.Text.Json:

```
string json = @"
[
    { ""Name"": ""Alice"", ""Age"": 20, ""Score"": 85 },
    { ""Name"": ""Bob"", ""Age"": 22, ""Score"": 78 },
    { ""Name"": ""Charlie"", ""Age"": 21, ""Score"": 90 }
]";

List<Student> students = JsonSerializer.Deserialize<List<Student>>(json
    );
```

Query Creation

Once you have your data source, the next step is to create a LINQ query. In method syntax, this involves chaining together methods that define the operations to be performed on the data. Common methods include `Where`, `Select`, `OrderBy`, `GroupBy`, and `Join`.

Here's how the query creation process works:

- Start with the data source: Call a LINQ extension method on your data source.
- Chain methods: Add methods to filter, project, sort, or group the data.
- Use lambda expressions: Define the criteria for each method using lambda expressions.

Continuing with our student list example, let's create a query to select students who scored above 80:

```
var highScorersQuery = students
    .Where(student => student.Score > 80)
    .Select(student => student.Name);
```

In this query:

- `Where` filters the students based on their Score.
- `Select` projects the filtered students, retrieving only their Name.

Each method in the chain represents a step in the query. The use of lambda expressions (`student => student.Score > 80` and `student => student.Name`) provides the logic for filtering and selecting data.

Query Execution

After creating the query, the final step is to execute it. In LINQ, execution can be deferred or immediate:

- **Deferred Execution:** The query is not executed until you iterate over the query variable, such as in a `foreach` loop or by calling methods like `ToList()` or `ToArray()`. This allows you to build up a query incrementally and execute it only when needed.
- **Immediate Execution:** The query is executed immediately when methods like `ToList()`, `ToArray()`, or `FirstOrDefault()` are called. These methods force the query to run and return a concrete result.

Continuing with our example, let's execute the query and print the results:

```
var highScorers = highScorersQuery.ToList();

foreach (var student in highScorers)
{
    Console.WriteLine(student);
}
```

In this code:

- `ToList()` triggers the execution of the query, converting the result into a list.
- A `foreach` loop iterates over the results and prints each student's name.

Working with Collections

LINQ seamlessly integrates with arrays and lists, allowing you to perform powerful data manipulations with minimal code. Here's a brief overview of how to use LINQ with these collections:

```
int[] numbersArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
List<int> numbersList = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
   };
```

Common LINQ Operations

Filtering (Where)

The `Where` method filters a collection based on a predicate (a condition defined by a lambda expression). This operation is useful for extracting a subset of elements that meet specific criteria.
Example:

```
var evenNumbersArray = numbersArray.Where(n => n % 2 == 0);
var evenNumbersList = numbersList.Where(n => n % 2 == 0);
```

In these examples, `Where` filters the elements of `numbersArray` and `numbersList` to include only even numbers.

Sorting (OrderBy, OrderByDescending)

LINQ provides `OrderBy` and `OrderByDescending` methods to sort collections in ascending or descending order, respectively. These methods take a key selector function to determine the sorting criteria.

Example:

```
var sortedArray = numbersArray.OrderBy(n => n);
var sortedList = numbersList.OrderByDescending(n => n);
```

In these examples, `OrderBy` sorts the elements of `numbersArray` in ascending order, while `OrderByDescending` sorts the elements of `numbersList` in descending order.

Projection (Select)

The `Select` method projects each element of a collection into a new form. This is commonly used to transform elements or to extract specific properties from complex objects.

Example:

```
var squaredNumbersArray = numbersArray.Select(n => n * n);
var squaredNumbersList = numbersList.Select(n => n * n);
```

In these examples, `Select` transforms each element in `numbersArray` and `numbersList` by squaring them.

Practical Examples

Let's put these operations together in practical scenarios.

Filtering and Sorting

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie", "
    David", "Eve" };

// Filter names starting with 'A' and sort them in ascending order
var filteredAndSortedNames = names
    .Where(name => name.StartsWith("A"))
    .OrderBy(name => name);

foreach (var name in filteredAndSortedNames)
{
    Console.WriteLine(name);
}
```

In this example:

- `Where` filters the list to include only names that start with 'A'.
- `OrderBy` sorts the filtered names in ascending order.

Projection and Filtering

```
List<Student> students = new List<Student>
{
    new Student { Name = "Alice", Age = 20, Score = 85 },
    new Student { Name = "Bob", Age = 22, Score = 78 },
    new Student { Name = "Charlie", Age = 21, Score = 90 }
};

// Project names of students who scored above 80
var highScorerNames = students
    .Where(student => student.Score > 80)
    .Select(student => student.Name);

foreach (var name in highScorerNames)
{
    Console.WriteLine(name);
}
```

In this example:

- `Where` filters the students to include only those with scores above 80.
- `Select` projects the names of the filtered students.

Combined Operations

```
int[] numbers = { 5, 3, 8, 6, 2, 7, 1, 4 };

// Filter even numbers, sort them in descending order, and project
   their squares
var processedNumbers = numbers
    .Where(n => n % 2 == 0)
    .OrderByDescending(n => n)
    .Select(n => n * n);

foreach (var num in processedNumbers)
{
    Console.WriteLine(num);
}
```

In this example:

- `Where` filters the array to include only even numbers.
- `OrderByDescending` sorts these numbers in descending order.
- `Select` projects the squares of the sorted numbers.

Using Lambda Expressions

Lambda expressions are a key feature in LINQ that provide a concise way to represent anonymous methods. They are integral to writing LINQ queries using method syntax. Understanding and effectively using lambda expressions is crucial for leveraging the full power of LINQ in C#.

A lambda expression is a short block of code that takes parameters (if any) and returns a value. They are used extensively in LINQ to represent functions that can be passed as arguments to LINQ methods. The syntax for lambda expressions is derived from mathematical notation, where the => symbol (known as the "goes to" operator) separates the parameters from the body of the expression.

Basic Syntax

Here's the basic syntax of a lambda expression:

```
(parameters) => expression_or_statement_block
```

For example:

```
x => x * x
```

This lambda expression takes one parameter, x, and returns its square.

Understanding Lambda Expressions

To understand lambda expressions better, let's break down their components and how they fit into LINQ method syntax:

Parameters

Enclosed in parentheses (), these represent the inputs to the lambda expression. If there are no parameters, you use empty parentheses (). For a single parameter, the parentheses can be omitted.

Lambda Operator

The =¿ operator separates the parameters from the body of the expression.

Expression or Statement Block

The right side of the => operator contains the expression or block of statements that are executed when the lambda is invoked. If the body consists of a single expression, it doesn't need curly braces . For multiple statements, you need curly braces and an explicit return statement.

Examples

Single Parameter Example:

```
// Single parameter without parentheses
x => x * x

// Single parameter with parentheses
(x) => x * x
```

Multiple Parameters Example:

```
// Multiple parameters
(x, y) => x + y
```

No Parameters Example:

```
// No parameters
() => Console.WriteLine("Hello, World!")
```

Statement Block Example:

```
// Multiple statements
(x, y) =>
{
    int sum = x + y;
    return sum * 2;
}
```

Writing Lambda Expressions in LINQ

Lambda expressions are used in LINQ method syntax to define the logic for filtering, projecting, grouping, and joining data. Here are some common LINQ methods that utilize lambda expressions:

Filtering with `Where`

The `Where` method filters a sequence based on a predicate defined by a lambda expression. For example, filtering a list of students to find those who scored above 80:

```
List<Student> students = GetStudents();

var highScorers = students.Where(student => student.Score > 80).ToList
    ();
```

In this example, `student => student.Score > 80` is the lambda expression that defines the filtering condition.

Projecting with `Select`

The `Select` method projects each element of a sequence into a new form. For example, projecting a list of students to get their names:

```
var studentNames = students.Select(student => student.Name).ToList();
```

Here, `student => student.Name` is the lambda expression that defines the projection.

Sorting with `OrderBy`

The `OrderBy` method sorts elements of a sequence based on a key. For example, sorting students by their score:

```
var sortedStudents = students.OrderBy(student => student.Score).ToList
    ();
```

In this case, `student => student.Score` is the lambda expression that defines the sorting key.

Grouping with `GroupBy`

The `GroupBy` method groups elements of a sequence based on a key. For example, grouping students by their age:

```
var groupedByAge = students.GroupBy(student => student.Age);
```

Here, `student => student.Age` is the lambda expression that defines the grouping key.

Joining with `Join`

The `Join` method joins two sequences based on matching keys. For example, joining students with their courses based on a common key:

```
List<Course> courses = GetCourses();

var studentCourses = students.Join(courses,
    student => student.CourseId,
    course => course.Id,
    (student, course) => new { student.Name, course.Title }).ToList();
```

In this example, the lambda expressions `student => student.CourseId` and `course => course.Id` define the matching keys, and `(student, course) => new { student.Name, course.Title }` defines the result selector.

Combining Queries

Combining queries in LINQ allows you to perform more complex data manipulations by joining, grouping, and aggregating data. These operations are essential for analyzing and transforming datasets effectively.

Joining Data

Joining data is a common operation when working with relational data. In LINQ, you can join two or more sequences (such as lists, arrays, or other collections) based on a common key. The Join method is used to achieve this, and it requires three key components:

- **Outer sequence:** The main data source.
- **Inner sequence:** The data source to join with the outer sequence.
- **Key selectors:** Functions to extract the join keys from the outer and inner sequences.
- **Result selector:** A function to create the result from the matched elements.

Example Narrative

"Imagine you have two lists: one containing students and another containing enrollments. Each enrollment links a student to a course by using the student's ID. You want to join these lists to retrieve a combined result of students along with their enrolled courses."

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
}

public class Enrollment
{
    public int StudentId { get; set; }
    public string Course { get; set; }
}

List<Student> students = new List<Student>
{
    new Student { StudentId = 1, Name = "Alice" },
    new Student { StudentId = 2, Name = "Bob" },
    new Student { StudentId = 3, Name = "Charlie" }
};

List<Enrollment> enrollments = new List<Enrollment>
{
    new Enrollment { StudentId = 1, Course = "Math" },
    new Enrollment { StudentId = 2, Course = "Science" },
    new Enrollment { StudentId = 1, Course = "History" }
};

var studentCourses = students.Join(
    enrollments,
    student => student.StudentId,
    enrollment => enrollment.StudentId,
    (student, enrollment) => new
    {
        StudentName = student.Name,
        Course = enrollment.Course
    });

foreach (var sc in studentCourses)
{
    Console.WriteLine($"{sc.StudentName} is enrolled in {sc.Course}");
}
```

In this example:

- The Join method links the students and enrollments lists on the `StudentId` property.
- The result selector creates an anonymous object containing the student's name and the course they are enrolled in.
- The `foreach` loop iterates over the joined results, printing each student's name and their courses.

Grouping Data

Grouping data is useful when you need to organize elements into groups based on a key. The GroupBy method in LINQ performs this operation, and it allows you to specify:

- **Key selector:** A function to extract the grouping key from each element.
- **Element selector (optional):** A function to select the elements to be included in each group.

Example Narrative

"Consider you have a list of students along with their grades in various subjects. You want to group the students by their grades to see which students achieved the same grade."

```
public class StudentGrade
{
    public string Name { get; set; }
    public string Subject { get; set; }
    public string Grade { get; set; }
}

List<StudentGrade> studentGrades = new List<StudentGrade>
{
    new StudentGrade { Name = "Alice", Subject = "Math", Grade = "A" },
    new StudentGrade { Name = "Bob", Subject = "Math", Grade = "B" },
    new StudentGrade { Name = "Charlie", Subject = "Math", Grade = "A"
   },
    new StudentGrade { Name = "Alice", Subject = "Science", Grade = "B"
    },
    new StudentGrade { Name = "Bob", Subject = "Science", Grade = "A" }
};

var gradeGroups = studentGrades.GroupBy(
    sg => sg.Grade,
    sg => sg.Name,
    (grade, names) => new
    {
        Grade = grade,
        Students = names.Distinct()
    });

foreach (var group in gradeGroups)
{
    Console.WriteLine($"Grade: {group.Grade}");
    foreach (var student in group.Students)
    {
        Console.WriteLine($"  Student: {student}");
    }
}
```

In this example:

- The GroupBy method groups `studentGrades` by the `Grade` property.
- The element selector extracts student names.
- The result selector creates an anonymous object containing the grade and a distinct list of student names.
- The `foreach` loop iterates over the groups, printing each grade and the students who received it.

Aggregating Data

Aggregation operations in LINQ compute a single value from a collection of values. Common aggregation methods include `Count`, `Sum`, `Average`, `Min`, and `Max`.

Example Narrative

"Suppose you have a list of transactions, and you want to calculate the total amount of sales, the average sale amount, and the highest sale amount."

```
public class Transaction
{
    public int Id { get; set; }
    public decimal Amount { get; set; }
}

List<Transaction> transactions = new List<Transaction>
{
    new Transaction { Id = 1, Amount = 100.50m },
    new Transaction { Id = 2, Amount = 200.75m },
    new Transaction { Id = 3, Amount = 50.25m }
};

int transactionCount = transactions.Count();
decimal totalSales = transactions.Sum(t => t.Amount);
decimal averageSales = transactions.Average(t => t.Amount);
decimal highestSale = transactions.Max(t => t.Amount);

Console.WriteLine($"Total Transactions: {transactionCount}");
Console.WriteLine($"Total Sales: {totalSales:C}");
Console.WriteLine($"Average Sale Amount: {averageSales:C}");
Console.WriteLine($"Highest Sale Amount: {highestSale:C}");
```

In this example:

- `Count` returns the number of transactions.
- `Sum` calculates the total sales amount.
- `Average` computes the average sale amount.
- `Max` finds the highest sale amount.
- The results are printed to the console, formatted as currency.

Best Practices for Writing Efficient LINQ Queries

Understand Deferred Execution

Deferred execution means that the evaluation of a LINQ query is delayed until you actually iterate over the query variable. This can be advantageous as it allows for query construction without immediate execution.

```
var query = students.Where(s => s.Score > 80);
// The query is not executed here.
var highScorers = query.ToList(); // The query is executed here.
```

Deferred execution enables you to compose queries dynamically and only execute them when necessary. This can help avoid unnecessary computations, especially if the query construction changes based on program logic.

Minimize Data Transfer

Apply filters as early as possible to reduce the amount of data being processed in subsequent operations. This reduces memory usage and improves performance.

```
var highScorers = students
    .Where(s => s.Score > 80) // Filter early
    .Select(s => new { s.Name, s.Score }); // Select relevant fields
```

Select only the fields you need. This minimizes the amount of data transferred and processed, which can significantly improve performance.

```
var studentNames = students
    .Select(s => s.Name); // Select only the necessary field
```

Use Efficient Data Structures

Select appropriate data structures based on your query requirements. For example, use Dictionary for fast lookups and List for ordered collections.

```
Dictionary<int, Student> studentDictionary = students.ToDictionary(s =>
    s.StudentId);
```

Enumerating a collection multiple times can be inefficient. Use methods like ToList or ToArray to cache results if you need to enumerate a collection more than once.

```
var highScorers = students.Where(s => s.Score > 80).ToList();
// Use highScorers list multiple times without re-executing the query.
```

Avoiding Common Pitfalls

Avoid Using LINQ to Modify Data

LINQ is designed for querying data rather than modifying it. Avoid using LINQ queries to update collections or databases. Instead, use appropriate update methods or commands.

```
// Inefficient way to modify data
students.Where(s => s.Score < 50).ToList().ForEach(s => s.Score += 10);

// Better approach
foreach (var student in students.Where(s => s.Score < 50))
{
    student.Score += 10;
}
```

Be Mindful of Deferred Execution

Be aware that deferred execution can lead to multiple query executions if the query variable is iterated over multiple times. Cache the results if necessary to avoid this.

```
var query = students.Where(s => s.Score > 80);
var count = query.Count(); // Executes the query
var highScorers = query.ToList(); // Executes the query again

// Better approach
var cachedResults = query.ToList();
var count = cachedResults.Count;
var highScorers = cachedResults;
```

Avoid Using Heavy Operations Inside Queries

Avoid performing expensive operations, such as complex calculations or database calls, inside LINQ queries. Move these operations outside the query to improve performance.

```
// Inefficient
var results = students.Where(s => ExpensiveOperation(s.Score)).ToList()
    ;

// Better approach
var processedScores = students.Select(s => new { s, IsHigh =
    ExpensiveOperation(s.Score) }).ToList();
var results = processedScores.Where(ps => ps.IsHigh).Select(ps => ps.s)
    .ToList();
```

Be Cautious with Large Data Sets in Memory

Be cautious when working with large data sets in memory, as this can lead to high memory consumption and possible performance degradation. Consider streaming data or processing it in chunks.

```
// Inefficient for large data sets
var largeDataSet = Enumerable.Range(1, int.MaxValue).ToList();

// Better approach: Process in chunks
var chunkSize = 1000;
for (int i = 0; i < largeDataSet.Count; i += chunkSize)
{
    var chunk = largeDataSet.Skip(i).Take(chunkSize);
    ProcessChunk(chunk);
}
```

Avoid Complex Nested Queries

Complex nested queries can be hard to read and maintain. Simplify them by breaking them into smaller, more manageable pieces or by using intermediate variables.

```
// Complex nested query
var complexQuery = students
    .Where(s => enrollments.Any(e => e.StudentId == s.StudentId && e.
   Course == "Math"))
    .Select(s => new
    {
        s.Name,
        MathCourse = enrollments.Where(e => e.StudentId == s.StudentId
   && e.Course == "Math").Select(e => e.Course).FirstOrDefault()
    }).ToList();

// Better approach
var mathEnrollments = enrollments.Where(e => e.Course == "Math").ToList
   ();
var simplifiedQuery = students
    .Where(s => mathEnrollments.Any(e => e.StudentId == s.StudentId))
    .Select(s => new
    {
        s.Name,
        MathCourse = mathEnrollments.Where(e => e.StudentId == s.
   StudentId).Select(e => e.Course).FirstOrDefault()
    }).ToList();
```