LINQ in C#

Language Integrated Query

Press Space for next page →

What is LINQ?

Language **IN**tegrated **Q**uery

- Introduced in C# 3.0 (.NET Framework 3.5)
- Query data from different sources with a unified syntax
- Works with collections, databases, XML, and more
- Brings SQL-like query capabilities to C#
- Makes code more readable and concise

Why Use LINQ?

WITHOUT LINQ

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

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

WITH LINQ

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

var evenNumbers = numbers
    .Where(n => n % 2 == 0)
    .ToList();
```

LINQ Syntax: Two Flavors

METHOD SYNTAX (FLUENT)

```
var result = numbers
    .Where(n => n > 5)
    .Select(n => n * 2)
    .OrderBy(n => n)
    .ToList();
```

More common, chainable

QUERY SYNTAX (SQL-LIKE)

```
var result = (from n in numbers
              where n > 5
              orderby n
              select n * 2)
             .ToList();
```

Familiar to SQL developers

Basic LINQ Operations

Common operations you'll use daily

- **Where** - Filter elements
- **Select** - Transform/project elements
- **OrderBy/OrderByDescending** - Sort elements
- **First/FirstOrDefault** - Get first element
- **ToList/ToArray** - Convert to collection
- **Count** - Count elements
- **Any/All** - Check conditions

Where - Filtering

Filter collections based on conditions

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

// Get even numbers
var evenNumbers = numbers.Where(n => n % 2 == 0);
// Result: { 2, 4, 6, 8, 10 }

// Get numbers greater than 5
var largeNumbers = numbers.Where(n => n > 5);
// Result: { 6, 7, 8, 9, 10 }

// Combine conditions
var result = numbers.Where(n => n > 3 && n < 8);
// Result: { 4, 5, 6, 7 }
```

Select - Projection

Transform elements into a different form

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

// Double each number
var doubled = numbers.Select(n => n * 2);
// Result: { 2, 4, 6, 8, 10 }

// Convert to strings
var strings = numbers.Select(n => n.ToString());
// Result: { "1", "2", "3", "4", "5" }

// Create anonymous objects
var objects = numbers.Select(n => new {
    Original = n,
    Squared = n * n
});
// Result: { {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25} }
```

Working with Objects

LINQ shines with complex objects

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

List<Person> people = new List<Person>
{
    new Person { Name = "Alice", Age = 30, City = "New York" },
    new Person { Name = "Bob", Age = 25, City = "Chicago" },
    new Person { Name = "Charlie", Age = 35, City = "New York" },
    new Person { Name = "Diana", Age = 28, City = "Chicago" }
};
```

Filtering Objects

```
// People over 28
var adults = people.Where(p => p.Age > 28);
// Alice (30), Charlie (35)

// People in New York
var newYorkers = people.Where(p => p.City == "New York");
// Alice, Charlie

// Combine conditions
var result = people.Where(p => p.Age > 25 && p.City == "Chicago");
// Diana (28, Chicago)

// Complex conditions
var filtered = people.Where(p =>
    p.Name.StartsWith("A") || p.Age < 30
);
// Alice, Bob, Diana
```

Chaining Operations

The real power of LINQ

```
// Filter, transform, and sort
var result = people
    .Where(p => p.Age > 25)           // Filter
    .OrderBy(p => p.Age)              // Sort
    .Select(p => p.Name)              // Project
    .ToList();                        // Execute
// Result: ["Bob", "Diana", "Alice", "Charlie"]

// Get names of people in New York, sorted
var names = people
    .Where(p => p.City == "New York")
    .OrderBy(p => p.Name)
    .Select(p => p.Name)
    .ToList();
// Result: ["Alice", "Charlie"]
```

OrderBy and Sorting

```
// Sort by age ascending
var byAge = people.OrderBy(p => p.Age);
// Bob (25), Diana (28), Alice (30), Charlie (35)

// Sort by age descending
var byAgeDesc = people.OrderByDescending(p => p.Age);
// Charlie (35), Alice (30), Diana (28), Bob (25)

// Multiple sort criteria
var sorted = people
    .OrderBy(p => p.City)        // First by city
    .ThenBy(p => p.Age);         // Then by age
// Bob (Chicago, 25), Diana (Chicago, 28),
// Alice (New York, 30), Charlie (New York, 35)
```

First, FirstOrDefault, Single

Getting individual elements

```
// First - throws if empty
var first = people.First();
// Returns: Alice

// FirstOrDefault - returns null if empty
var firstOver40 = people.FirstOrDefault(p => p.Age > 40);
// Returns: null

// First with condition
var firstInChicago = people.First(p => p.City == "Chicago");
// Returns: Bob

// Single - throws if 0 or more than 1 result
var single = people.Single(p => p.Name == "Alice");
// Returns: Alice

// SingleOrDefault - null if empty, throws if multiple
var singleOver40 = people.SingleOrDefault(p => p.Age > 40);
// Returns: null
```

Any and All

Check conditions across collections

```
// Any - check if at least one matches
bool hasAdults = people.Any(p => p.Age >= 30);
// true (Alice is 30)

bool hasTeens = people.Any(p => p.Age < 20);
// false

bool hasAny = people.Any();
// true (list is not empty)

// All - check if all match
bool allAdults = people.All(p => p.Age >= 18);
// true (all are 18+)

bool allOver30 = people.All(p => p.Age >= 30);
// false (Bob is 25)
```

Count and Sum

Aggregate operations

```
// Count all
int total = people.Count();
// 4

// Count with condition
int chicagoCount = people.Count(p => p.City == "Chicago");
// 2

// Sum ages
int totalAge = people.Sum(p => p.Age);
// 118 (30 + 25 + 35 + 28)

// Average age
double avgAge = people.Average(p => p.Age);
// 29.5

// Min and Max
int youngest = people.Min(p => p.Age);  // 25
int oldest = people.Max(p => p.Age);    // 35
```

GroupBy - Grouping Data

Group elements by a key

```
// Group by city
var byCity = people.GroupBy(p => p.City);

foreach (var group in byCity)
{
    Console.WriteLine($"{group.Key}:");
    foreach (var person in group)
    {
        Console.WriteLine($"  {person.Name} ({person.Age})");
    }
}

// Output:
// New York:
//   Alice (30)
//   Charlie (35)
// Chicago:
//   Bob (25)
//   Diana (28)
```

GroupBy with Aggregates

```
// Count people per city
var cityCount = people
    .GroupBy(p => p.City)
    .Select(g => new {
        City = g.Key,
        Count = g.Count()
    });
// { City: "New York", Count: 2 },
// { City: "Chicago", Count: 2 }

// Average age per city
var avgAgeByCity = people
    .GroupBy(p => p.City)
    .Select(g => new {
        City = g.Key,
        AvgAge = g.Average(p => p.Age)
    });
// { City: "New York", AvgAge: 32.5 },
// { City: "Chicago", AvgAge: 26.5 }
```

Join Operations

Combine data from multiple sources

```
public class Order
{
    public int Id { get; set; }
    public string PersonName { get; set; }
    public decimal Amount { get; set; }
}

List<Order> orders = new List<Order>
{
    new Order { Id = 1, PersonName = "Alice", Amount = 100 },
    new Order { Id = 2, PersonName = "Bob", Amount = 200 },
    new Order { Id = 3, PersonName = "Alice", Amount = 150 }
};
```

Inner Join Example

```
// Join people with their orders
var joined = people.Join(
    orders,                             // Collection to join
    person => person.Name,              // Key from first
    order => order.PersonName,          // Key from second
    (person, order) => new              // Result selector
    {
        Name = person.Name,
        Age = person.Age,
        OrderAmount = order.Amount
    }
);

// Result:
// { Name: "Alice", Age: 30, OrderAmount: 100 }
// { Name: "Alice", Age: 30, OrderAmount: 150 }
// { Name: "Bob", Age: 25, OrderAmount: 200 }
```

SelectMany - Flattening

Flatten nested collections

```
public class Department
{
    public string Name { get; set; }
    public List<string> Employees { get; set; }
}

var departments = new List<Department>
{
    new Department {
        Name = "IT",
        Employees = new List<string> { "Alice", "Bob" }
    },
    new Department {
        Name = "HR",
        Employees = new List<string> { "Charlie", "Diana" }
    }
};

// Flatten all employees
```

SelectMany with Projection

```
// Get employees with their departments
var employeeDepts = departments.SelectMany(
    dept => dept.Employees,
    (dept, employee) => new
    {
        Department = dept.Name,
        Employee = employee
    }
);

// Result:
// { Department: "IT", Employee: "Alice" }
// { Department: "IT", Employee: "Bob" }
// { Department: "HR", Employee: "Charlie" }
// { Department: "HR", Employee: "Diana" }
```

Distinct and Union

Set operations

```
List<int> numbers1 = new List<int> { 1, 2, 3, 4, 5 };
List<int> numbers2 = new List<int> { 4, 5, 6, 7, 8 };

// Distinct - remove duplicates
var withDupes = new List<int> { 1, 2, 2, 3, 3, 3 };
var unique = withDupes.Distinct();
// Result: { 1, 2, 3 }

// Union - combine and remove duplicates
var union = numbers1.Union(numbers2);
// Result: { 1, 2, 3, 4, 5, 6, 7, 8 }

// Intersect - common elements
var intersect = numbers1.Intersect(numbers2);
// Result: { 4, 5 }

// Except - in first but not second
var except = numbers1.Except(numbers2);
// Result: { 1, 2, 3 }
```

Skip and Take - Pagination

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

// Take first 5
var first5 = numbers.Take(5);
// Result: { 1, 2, 3, 4, 5 }

// Skip first 5, take rest
var last5 = numbers.Skip(5);
// Result: { 6, 7, 8, 9, 10 }

// Pagination: page 2, size 3
int pageNumber = 2;
int pageSize = 3;
var page = numbers
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize);
// Result: { 4, 5, 6 }
```

TakeWhile and SkipWhile

Conditional take/skip

```
List<int> numbers = new List<int> { 1, 3, 5, 2, 4, 6, 7, 9 };

// Take while condition is true
var takeWhile = numbers.TakeWhile(n => n < 6);
// Result: { 1, 3, 5, 2, 4 }
// Stops at 6 even though 7, 9 are also available

// Skip while condition is true
var skipWhile = numbers.SkipWhile(n => n < 6);
// Result: { 6, 7, 9 }

// Practical example: skip header rows
var data = new List<string> { "Header1", "Header2", "Data1", "Data2" };
var dataOnly = data.Skip(2);  // Skip first 2 rows
```

Reverse

Reverse the order

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var reversed = numbers.Reverse();
// Result: { 5, 4, 3, 2, 1 }

// Useful with OrderBy
var people = /* ... */;
var youngestFirst = people.OrderBy(p => p.Age).Reverse();
// Same as: people.OrderByDescending(p => p.Age)
```

Zip - Combine Collections

Pair elements from two sequences

```
var names = new List<string> { "Alice", "Bob", "Charlie" };
var ages = new List<int> { 30, 25, 35 };

var combined = names.Zip(ages, (name, age) => new
{
    Name = name,
    Age = age
});

// Result:
// { Name: "Alice", Age: 30 }
// { Name: "Bob", Age: 25 }
// { Name: "Charlie", Age: 35 }

// Stops at shortest collection
var moreNames = new List<string> { "Diana", "Eve" };
var result = moreNames.Zip(ages, (n, a) => $"{n}: {a}");
// Result: ["Diana: 30", "Eve: 25", "Charlie: 35"]
```

Aggregate - Custom Aggregation

Reduce a sequence to a single value

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

// Sum using Aggregate
var sum = numbers.Aggregate((total, next) => total + next);
// Result: 15

// Product
var product = numbers.Aggregate((total, next) => total * next);
// Result: 120

// With seed value
var sumPlus100 = numbers.Aggregate(100, (total, next) => total + next);
// Result: 115

// String concatenation
var words = new List<string> { "Hello", "World", "LINQ" };
var sentence = words.Aggregate((curr, next) => curr + " " + next);
// Result: "Hello World LINQ"
```

Complex Example: E-Commerce

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

List<Product> products = new List<Product>
{
    new Product { Id = 1, Name = "Laptop", Price = 1200, Category = "Electronics", Stock = 5 },
    new Product { Id = 2, Name = "Mouse", Price = 25, Category = "Electronics", Stock = 50 },
    new Product { Id = 3, Name = "Desk", Price = 300, Category = "Furniture", Stock = 10 },
    new Product { Id = 4, Name = "Chair", Price = 150, Category = "Furniture", Stock = 15 },
    new Product { Id = 5, Name = "Monitor", Price = 400, Category = "Electronics", Stock = 8 }
};
```

E-Commerce Queries

```
// Products under $200
var affordable = products.Where(p => p.Price < 200);

// Group by category with totals
var categoryStats = products
    .GroupBy(p => p.Category)
    .Select(g => new
    {
        Category = g.Key,
        Count = g.Count(),
        TotalValue = g.Sum(p => p.Price * p.Stock),
        AvgPrice = g.Average(p => p.Price)
    });

// Most expensive product in each category
var mostExpensive = products
    .GroupBy(p => p.Category)
    .Select(g => g.OrderByDescending(p => p.Price).First());

// Low stock electronics
var lowStock = products
```

Deferred Execution

LINQ queries are lazy - they execute when enumerated

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

// Query defined but NOT executed yet
var query = numbers.Where(n =>
{
    Console.WriteLine($"Checking {n}");
    return n > 3;
});

Console.WriteLine("Query defined");

// NOW it executes (when enumerated)
foreach (var num in query)
{
    Console.WriteLine($"Result: {num}");
}

// Output:
// Query defined
// Checking 1
// Checking 2
// Checking 3
// Checking 4
```

Immediate Execution

Force immediate execution with ToList, ToArray, Count

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

// Immediate execution - runs NOW
var list = numbers.Where(n => n > 3).ToList();
// Query executes immediately

// Count also executes immediately
int count = numbers.Where(n => n > 3).Count();

// Deferred vs Immediate
numbers.Add(6);

var deferred = numbers.Where(n => n > 3);      // Not executed
var immediate = numbers.Where(n => n > 3).ToList();  // Executed now

numbers.Add(7);

Console.WriteLine(deferred.Count());    // 4 (includes 6 and 7)
Console.WriteLine(immediate.Count());   // 3 (only has 4, 5, 6)
```

LINQ to SQL / Entity Framework

LINQ works with databases too!

```
// Using Entity Framework
using (var context = new MyDbContext())
{
    // Query is translated to SQL
    var customers = context.Customers
        .Where(c => c.City == "New York")
        .OrderBy(c => c.Name)
        .ToList();

    // Generated SQL:
    // SELECT * FROM Customers
    // WHERE City = 'New York'
    // ORDER BY Name

    // Join across tables
    var orders = context.Orders
        .Join(context.Customers,
            o => o.CustomerId,
            c => c.Id,
            (o, c) => new { Order = o, Customer = c })
        .Where(x => x.Customer.City == "New York")
```

Best Practices

Use meaningful variable names in lambdas

Chain operations for readability

ToList() wisely - only when needed

Avoid side effects in lambda expressions

Consider performance for large datasets

Use AsEnumerable() to switch from DB to memory

Null checks with `?.` and `??`

Good vs Bad Examples

❌ BAD

```
var x = people.Where(p =>
  p.Age > 25).OrderBy(p =>
  p.Name).Select(p => p.Name)
  .ToList();

// Single letter variables
var r = people.Where(x =>
  x.Age > 25);

// Multiple ToList()
var list1 = people.Where(p =>
  p.Age > 25).ToList();
var list2 = list1.OrderBy(p =>
  p.Name).ToList();
```

✅ GOOD

```
var adultNames = people
    .Where(p => p.Age > 25)
    .OrderBy(p => p.Name)
    .Select(p => p.Name)
    .ToList();

// Descriptive names
var adults = people
    .Where(person => person.Age > 25);

// Single ToList() at end
var sortedAdults = people
    .Where(p => p.Age > 25)
    .OrderBy(p => p.Name)
    .ToList();
```

Performance Tips

Filter early - use Where before Select

Avoid multiple enumerations - use ToList() if iterating multiple times

Use appropriate collections - List vs Array vs IEnumerable

Indexed access - First() vs 0 when appropriate

Count vs Any - Use Any() for existence checks

Take + OrderBy - More efficient than OrderBy + Take for large sets

Performance Example

```
// ❌ Slower - select then filter
var result1 = people
    .Select(p => new { p.Name, p.Age })
    .Where(p => p.Age > 25)
    .ToList();

// ✅ Faster - filter then select
var result2 = people
    .Where(p => p.Age > 25)
    .Select(p => new { p.Name, p.Age })
    .ToList();

// ❌ Slow - Count() > 0
if (people.Where(p => p.Age > 25).Count() > 0)

// ✅ Fast - Any()
if (people.Any(p => p.Age > 25))
```

Common Pitfalls

1. MULTIPLE ENUMERATION

```
var query = people.Where(p => p.Age > 25);
int count = query.Count();  // Enumerates
var list = query.ToList();  // Enumerates again!

// Better:
var list = people.Where(p => p.Age > 25).ToList();
int count = list.Count;
```

2. MODIFYING DURING ENUMERATION

```
// ❌ Throws exception
foreach (var item in list)
    list.Remove(item);

// ✅ Use ToList() first
foreach (var item in list.ToList())
    list.Remove(item);
```

Real-World Example: Reporting

```
public class SalesReport
{
    public DateTime Date { get; set; }
    public string Product { get; set; }
    public decimal Amount { get; set; }
    public string Region { get; set; }
}

List<SalesReport> sales = LoadSales();

// Monthly sales by region
var monthlySales = sales
    .GroupBy(s => new { Month = s.Date.Month, s.Region })
    .Select(g => new
    {
        Month = g.Key.Month,
        Region = g.Key.Region,
        TotalSales = g.Sum(s => s.Amount),
        AvgSale = g.Average(s => s.Amount),
        Count = g.Count()
    })
```

Top Products by Region

```
// Top 5 products per region
var topProducts = sales
    .GroupBy(s => s.Region)
    .SelectMany(regionGroup => regionGroup
        .GroupBy(s => s.Product)
        .Select(productGroup => new
        {
            Region = regionGroup.Key,
            Product = productGroup.Key,
            TotalSales = productGroup.Sum(s => s.Amount)
        })
        .OrderByDescending(x => x.TotalSales)
        .Take(5)
    );

// Growth analysis - compare periods
var q1Sales = sales.Where(s => s.Date.Month <= 3).Sum(s => s.Amount);
var q2Sales = sales.Where(s => s.Date.Month > 3 && s.Date.Month <= 6)
                   .Sum(s => s.Amount);
var growth = ((q2Sales - q1Sales) / q1Sales) * 100;
```

LINQ with XML

```
XDocument doc = XDocument.Load("data.xml");

// Query XML
var names = doc.Descendants("Person")
    .Where(p => (int)p.Element("Age") > 25)
    .Select(p => (string)p.Element("Name"));

// Create XML
var xml = new XElement("People",
    people.Select(p => new XElement("Person",
        new XElement("Name", p.Name),
        new XElement("Age", p.Age)
    ))
);
```

LINQ to Objects Summary

- **Declarative** - Say what you want, not how
- **Composable** - Chain operations together
- **Lazy** - Deferred execution by default
- **Type-safe** - Compile-time checking
- **Versatile** - Works with any IEnumerable
- **Powerful** - Complex queries in few lines

Practice Exercises

Let’s work through some hands-on examples!

Exercise 1: Student Grades

Calculate average grade per student and find top performers

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

List<StudentGrade> grades = new List<StudentGrade>
{
    new StudentGrade { Name = "Alice", Subject = "Math", Score = 95 },
    new StudentGrade { Name = "Alice", Subject = "Science", Score = 88 },
    new StudentGrade { Name = "Bob", Subject = "Math", Score = 78 },
    new StudentGrade { Name = "Bob", Subject = "Science", Score = 92 },
    new StudentGrade { Name = "Charlie", Subject = "Math", Score = 85 },
    new StudentGrade { Name = "Charlie", Subject = "Science", Score = 90 }
};

// TODO: Calculate average score per student
// TODO: Find top 2 students by average score
```

Exercise 1: Solution

```
// Calculate average score per student
var studentAverages = grades
    .GroupBy(g => g.Name)
    .Select(group => new
    {
        Name = group.Key,
        AverageScore = group.Average(g => g.Score)
    })
    .OrderByDescending(s => s.AverageScore);

// Result:
// Alice: 91.5
// Charlie: 87.5
// Bob: 85.0

// Top 2 students
var topStudents = studentAverages.Take(2).ToList();

foreach (var student in topStudents)
{
    Console.WriteLine($"{student.Name}: {student.AverageScore:F1}");
}
```

Exercise 2: Product Catalog

Filter products by price and group by category

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

List<Product> products = new List<Product>
{
    new Product { Name = "Laptop", Price = 999, Category = "Electronics" },
    new Product { Name = "Mouse", Price = 25, Category = "Electronics" },
    new Product { Name = "Desk", Price = 350, Category = "Furniture" },
    new Product { Name = "Chair", Price = 200, Category = "Furniture" },
    new Product { Name = "Monitor", Price = 300, Category = "Electronics" },
    new Product { Name = "Lamp", Price = 45, Category = "Furniture" }
};

// TODO: Find products under $100
// TODO: Count products per category
// TODO: Find most expensive product in each category
```

Exercise 2: Solution

```
// Products under $100
var affordable = products
    .Where(p => p.Price < 100)
    .OrderBy(p => p.Price);
// Mouse ($25), Lamp ($45)

// Count per category
var categoryCount = products
    .GroupBy(p => p.Category)
    .Select(g => new { Category = g.Key, Count = g.Count() });
// Electronics: 3, Furniture: 3

// Most expensive per category
var mostExpensive = products
    .GroupBy(p => p.Category)
    .Select(g => new
    {
        Category = g.Key,
        Product = g.OrderByDescending(p => p.Price).First()
    });
// Electronics: Laptop ($999)
// Furniture: Desk ($350)
```

Exercise 3: Employee Data

Analyze salaries by department

```
public class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public decimal Salary { get; set; }
}

List<Employee> employees = new List<Employee>
{
    new Employee { Name = "Alice", Department = "IT", Salary = 80000 },
    new Employee { Name = "Bob", Department = "IT", Salary = 75000 },
    new Employee { Name = "Charlie", Department = "HR", Salary = 60000 },
    new Employee { Name = "Diana", Department = "HR", Salary = 65000 },
    new Employee { Name = "Eve", Department = "Sales", Salary = 70000 },
    new Employee { Name = "Frank", Department = "Sales", Salary = 72000 }
};

// TODO: Average salary per department
// TODO: Highest paid employee per department
// TODO: Total payroll per department
```

Exercise 3: Solution

```
// Average salary per department
var avgSalary = employees
    .GroupBy(e => e.Department)
    .Select(g => new
    {
        Department = g.Key,
        AvgSalary = g.Average(e => e.Salary)
    });
// IT: $77,500, HR: $62,500, Sales: $71,000

// Highest paid per department
var topPaid = employees
    .GroupBy(e => e.Department)
    .Select(g => new
    {
        Department = g.Key,
        Employee = g.OrderByDescending(e => e.Salary).First()
    });

// Total payroll per department
var payroll = employees
    .GroupBy(e => e.Department)
    .Select(g => new
    {
```

Exercise 4: Order Processing

Analyze customer orders

```
public class Order
{
    public string Customer { get; set; }
    public decimal Amount { get; set; }
    public DateTime Date { get; set; }
}

List<Order> orders = new List<Order>
{
    new Order { Customer = "Alice", Amount = 100, Date = new DateTime(2024, 1, 15) },
    new Order { Customer = "Bob", Amount = 250, Date = new DateTime(2024, 1, 16) },
    new Order { Customer = "Alice", Amount = 150, Date = new DateTime(2024, 1, 20) },
    new Order { Customer = "Charlie", Amount = 300, Date = new DateTime(2024, 1, 22) },
    new Order { Customer = "Bob", Amount = 120, Date = new DateTime(2024, 1, 25) },
    new Order { Customer = "Alice", Amount = 200, Date = new DateTime(2024, 2, 1) }
};

// TODO: Total spent per customer
// TODO: Top 2 customers by spending
// TODO: Average order value
```

Exercise 4: Solution

```
// Total spent per customer
var customerTotal = orders
    .GroupBy(o => o.Customer)
    .Select(g => new
    {
        Customer = g.Key,
        TotalSpent = g.Sum(o => o.Amount),
        OrderCount = g.Count()
    })
    .OrderByDescending(c => c.TotalSpent);
// Alice: $450 (3 orders)
// Charlie: $300 (1 order)
// Bob: $370 (2 orders)

// Top 2 customers
var topCustomers = customerTotal.Take(2);

// Average order value
var avgOrderValue = orders.Average(o => o.Amount);
// $186.67

// Orders in January
var januaryOrders = orders
    .Where(o => o.Date.Month == 1)
```

Exercise 5: Text Analysis

Analyze word frequency

```
string text = "the quick brown fox jumps over the lazy dog the fox was quick";
string[] words = text.Split(' ');

// TODO: Count how many times each word appears
// TODO: Find the 3 most common words
// TODO: Find the longest word
// TODO: Count words starting with a specific letter
```

Exercise 5: Solution

```
// Word frequency
var wordCount = words
    .GroupBy(w => w)
    .Select(g => new
    {
        Word = g.Key,
        Count = g.Count()
    })
    .OrderByDescending(w => w.Count);
// "the": 3, "fox": 2, "quick": 2, others: 1

// Top 3 most common words
var topWords = wordCount.Take(3);

// Longest word
var longestWord = words
    .OrderByDescending(w => w.Length)
    .First();
// "quick" or "brown" or "jumps" (5 letters)

// Words starting with 't'
var tWords = words
    .Where(w => w.StartsWith("t"))
```

Exercise 6: Time Series

Aggregate daily sales data

```
public class Sale
{
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}

List<Sale> sales = new List<Sale>
{
    new Sale { Date = new DateTime(2024, 1, 1), Amount = 100 },
    new Sale { Date = new DateTime(2024, 1, 1), Amount = 150 },
    new Sale { Date = new DateTime(2024, 1, 2), Amount = 200 },
    new Sale { Date = new DateTime(2024, 2, 1), Amount = 300 },
    new Sale { Date = new DateTime(2024, 2, 1), Amount = 250 },
    new Sale { Date = new DateTime(2024, 2, 15), Amount = 180 }
};

// TODO: Total sales per day
// TODO: Total sales per month
// TODO: Best sales day
```

Exercise 6: Solution

```
// Total sales per day
var dailySales = sales
    .GroupBy(s => s.Date.Date)
    .Select(g => new
    {
        Date = g.Key,
        Total = g.Sum(s => s.Amount),
        Count = g.Count()
    })
    .OrderBy(d => d.Date);
// Jan 1: $250 (2 sales), Jan 2: $200 (1 sale), etc.

// Total sales per month
var monthlySales = sales
    .GroupBy(s => new { s.Date.Year, s.Date.Month })
    .Select(g => new
    {
        Year = g.Key.Year,
        Month = g.Key.Month,
        Total = g.Sum(s => s.Amount)
    });
// Jan 2024: $450, Feb 2024: $730

// Best sales day
```

Additional Resources

DOCUMENTATION

- Microsoft LINQ Documentation
- 101 LINQ Samples

TOOLS

- **LINQPad** - Interactive LINQ playground
- **Visual Studio** - IntelliSense for LINQ
- **ReSharper** - LINQ refactoring suggestions