Școala
informală
de IT

Language Integrated Query in .NET (LINQ)

Școala Informală de IT

Agenda

- LINQ Building Blocks
- Sequences
- Query Operators and Expressions
- Query Expression Trees
- LINQ to Objects
- Querying Collections
- Projection, Conversion, Aggregation
- Sorting, Grouping, Joins, Nested Queries

LINQ Building Blocks

LINQ Building Blocks

- Software developers spend a lot of time to obtain and manipulate data
- Data can be stored in
  - Collections
  - Databases
  - XML documents
  - etc…
- As of .NET 3.5 developers can use LINQ – a simplified approach to data manipulation

LINQ Building Blocks(2)

- LINQ is a set of extensions to .NET Framework
  - Encompasses language-integrated query, set, and transform operations
  - Consistent manner to obtain and manipulate "data" in the broad sense of the term
- Query expressions can be defined directly within the C# programming language
  - Used to interact with numerous data types
  - Converter to expression trees at compile time and evaluated at runtime

Why LINQ?

- LINQ offers syntax highlighting that proves helpful to find out mistakes during design time.
- LINQ offers IntelliSense which means writing more accurate queries easily.
- Writing codes is quite faster in LINQ and thus development time also gets reduced significantly.
- LINQ makes easy debugging due to its integration in the C# language.
- Viewing relationship between two tables is easy with LINQ due to its hierarchical feature and this enables composing queries joining multiple tables in less time.

Why LINQ? (2)

- LINQ allows usage of a single LINQ syntax while querying many diverse data sources and this is mainly because of its unitive foundation.
- LINQ is extensible that means it is possible to use knowledge of LINQ to querying new data source types.
- LINQ offers the facility of joining several data sources in a single query as well as breaking complex problems into a set of short queries easy to debug.
- LINQ offers easy transformation for conversion of one data type to another like transforming SQL data to XML data.

LINQ Building Blocks (3)

- LINQ allows query expressions to manipulate:
  - Any object implementing `IEnumerable<T>`
  - Collections of objects
  - Relational databases
  - XML documents
- The query expressions are based on numerous SQL-like query operators
  - Intentionally designed to look and feel very similar to SQL expressions

LINQ Building Blocks (4)

- "LINQ" is the term used to describe this overall approach to data access
  - LINQ to Objects
    - LINQ over objects implementing `IEnumerable<T>`
  - LINQ to SQL and LINQ to Entities implement LINQ over relational data
  - LINQ to `DataSet` is a superset of LINQ to SQL
  - LINQ to XML is LINQ over XML documents

LINQ to *

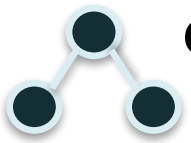
C# | VB.NET | Others...

.NET Language-Integrated Query (LINQ)

LINQ enabled data sources

LINQ enabled ADO.NET

LINQ to Objests | LING to DataSets | LINQ to SQL | LINQ to Entities | LINQ to XML



Objects

Relational Data

XML

Query Operations

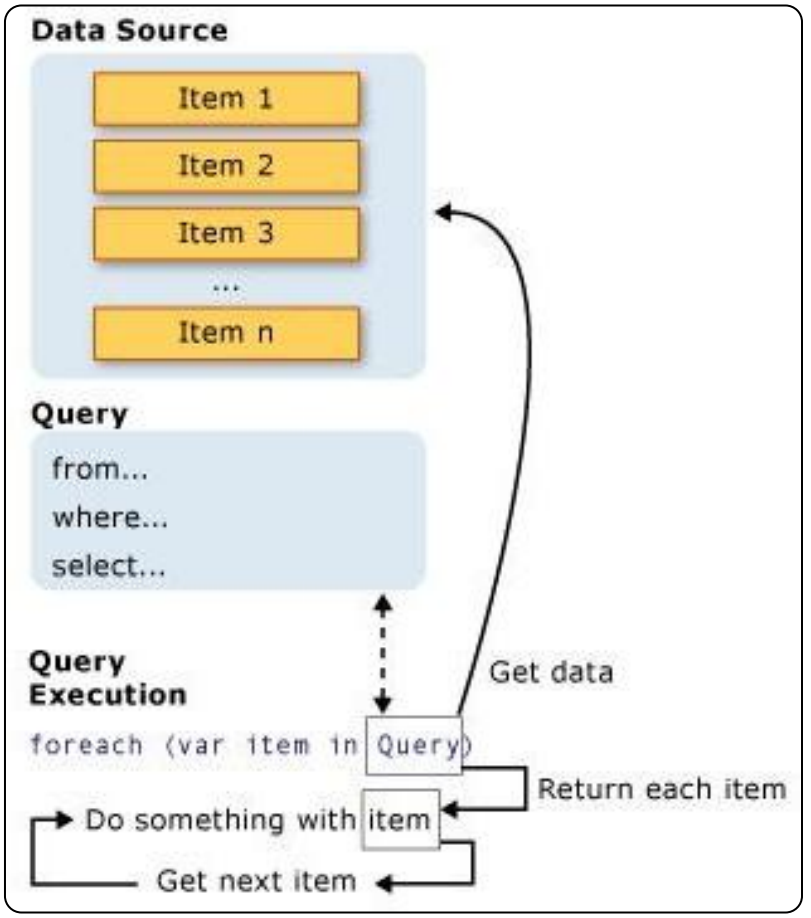
- All LINQ query operations consist of three distinct actions:
  - Obtain the data source
  - Create the query
  - Execute the query

```
public Beagle(string
name, string ability) :
base(name)
{...}
```

Query Operations

- Simple Inheritance example(1)

```
public class Dog
{
  public string Name { get; set; }

  public Dog(string name)
  {
    this.Name = name;
  }

  public void Bark()
  {
    Console.WriteLine("Dog barks...");
  }
}
```



Școala
informală
de IT

LINQ Sequences

IEnumerable<T> and Sequences

- The interface `IEnumerable<T>` is universal LINQ data source
  - Implemented by arrays and all .NET generic collections
  - Enables enumerating over a collection of elements
- A `sequence` in LINQ means a collection implementing the `IEnumerable<T>` interface
- Any variable declared as `IEnumerable<T>` for type `T` is considered a sequence of type `T`

IEnumerable<T> and Sequences

- Most of the Standard Query Operators are extension methods in the static class `System.Linq.Enumerable`
  - Prototyped with an `IEnumerable<T>` as their first argument
  - E.g. `Min(IEnumerable<T>)`, `Where(IEnumerable<T>, Func<T, bool>)`
- Use the `Cast` or `OfType` operators to perform LINQ queries on legacy, non-generic .NET collections

Query Operators and Expressions

LINQ Query Expressions

- When you have a collection of data, a common task is to extract a subset of items based on a given requirement
  - You want to obtain only the items with names that contain a number
  - Or don't have embedded spaces
- LINQ query expressions can greatly simplify the process
- Query expressions are written in a declarative query syntax introduced in C# 3.0

LINQ Query Expressions (2)

- LINQ query expressions are written in a declarative SQL-like syntax
- Example: extracting a subset of array containing items with names of more than 6 characters:

```
string[] games = {"Morrowind", "BioShock", "Daxter",
  "The Darkness", "Half Life", "System Shock 2"};
IEnumerable<string> subset =
  from g in games
  where g.Length > 6
  orderby g
  select g;
foreach (string s in subset)
  Console.WriteLine("Item: {0}", s);
```

Școala
informală
de IT

Query Expressions

- WORKSHOP -

LINQ Query Expressions (3)

- In LINQ a query is a basic language construction
  - Just like classes, methods and delegates in C#
- Query expressions are used to query and transform data from any LINQ-enabled data source
- A LINQ query is not executed until
  - You iterate over the query results
  - You try to access any of the elements in the result set

Query Operators

- Query operators in C# are keywords like:
  - `from`, `in`, `where`, `orderby`, `select`, …
- For each standard query operator a corresponding extension method exists
  - E.g. `where` → `Where(IEnumerable<T>)`
- At compile time the C# compiler translates query expressions into expression trees
  - Expression trees are sequences of method calls (from `System.Linq.Enumerable`)

Query Operators - Syntax

- **The basic syntax of LINQ queries is:**

```
IEnumerable<string> subset =
  from g in games select g;
```

```
var subset =
  games.Select(g => g);
```

  - **This selects all elements in games data source**

- **You can apply criteria by the operator where**

  - **Any valid C# boolean expression can be used**

```
IEnumerable<string> subset =
  from g in games
  where g.Price < 20
  select g;
```

```
var subset =
  games.Select(g => g).
  Where(g => g.Price <
20);
```

Query Operators (2)

- Two sets of LINQ standard operators
  - Operating on `IEnumerable<T>`
  - Operating on `IQueryable<T>`
- LINQ query operators are shorthand versions for various extension methods
  - Defined in `System.Linq.Enumerable` type
  - Example:

```
IEnumerable<string> subset =
  games.Where(g => g.Price < 2
```

```
var subset =
  from g in games
  where g.Price < 20
  select g;
```

Query Operators (3)

- The standard query operators provide query capabilities including
  - Filtering – `where`
  - Projection – `select`, `selectMany`
  - Aggregation – `Sum`, `Max`, `Count`, `Average`
  - Sorting – `orderby`
  - Grouping – `groupby`
  - … and many more

Standard Query Operators - Example

```
string[] games = {"Morrowind", "BioShock","Half Life",
  "The Darkness","Daxter", "System Shock 2"};

// Build a query expression using extension methods
// granted to the Array via the Enumerable type

var subset = games.Where(game => game.Length > 6).
  OrderBy(game => game).Select(game => game);

foreach (var game in subset)
  Console.WriteLine(game);
Console.WriteLine();
```

```
var subset =
  from g in games
  where g.Length > 6
  orderby g
  select g;
```

Școala
informală
de IT

Standard Query Operators

- WORKSHOP -

Școala
informală
de IT

Query Expression Trees

Query Expression Trees

- A query expression tree is an efficient data structure representing a LINQ expression
  - Type of abstract syntax tree used for storing parsed expressions from the source code
  - Lambda expressions often translate into query expression trees
- `IQueryable<T>` is interface implemented by query providers (e.g. LINQ to SQL, LINQ to XML, LINQ to Entities)
- `IQueryable<T>` objects use expression trees

Query Expression Trees (2)

- **LINQ queries can be performed over two standard .NET interfaces:**
  - `IEnumerable<T>`
    - **At compile time IL is emitted**
  - `IQueryable<T>`
    - **At compile time a query expression tree is emitted**
  - **Both are evaluated at runtime**

Query Expression Trees (3)

- **When any element of the `IQueryable<T>` result is being accessed for the first time**
  - **A query is generated from the expression tree and is executed**

```
int[] nums = new int[] {
  6, 2, 7, 1, 9, 3 };
var numsLessThanFour =
  from i in nums
  where i < 4
  select i;
foreach (var item in numsLessThanFour)
  Console.WriteLine(item);
```

Variable is of type `IQueryable<int>`

Query is generated and executed here

Expression Trees - Benefits

- `IQueryable<T>` uses expression trees which provide it mechanisms:
  - For smart decisions and optimizations when query is generated
    - Based on analysis of expression trees
  - Optimizing multiple nested or complex queries
  - Combining multiple queries into very efficient single one

LINQ to Objects

LINQ to Objects

- LINQ to Objects refers to using LINQ queries directly over IEnumerable<T> collection
  - Without the an intermediate LINQ provider or API, such as LINQ to SQL or LINQ to XML
  - Applicable to any enumerable collection
- The old school data retrieval approach
  - Write complex foreach loops that specify how to retrieve data from a collection
- The LINQ approach – write declarative code that describes what to be retrieved

LINQ to Objects - Advantages

- **LINQ queries offer three main advantages over traditional `foreach` loops**
  - **They are more concise and easy-to-read**
    - **Especially when filtering by multiple conditions**
  - **Provide powerful filtering, ordering, and grouping capabilities**
  - **Can be ported to other data sources with little or no modification**

LINQ to Objects - Example

- **LINQ to Objects is performing SQL-like queries on in-memory data collections and arrays**

```
string[] presidents = { "Adams", "Arthur", "Buchanan",
  "Bush", "Carter","Cleveland","Clinton", "Coolidge",
  "Eisenhower", "Fillmore", "Ford", "Garfield","Grant",
  "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
  "Jefferson", "Johnson", "Kennedy", "Lincoln",
  "Madison", "McKinley", "Monroe", "Nixon", "Pierce",
  "Polk", "Reagan", "Roosevelt", "Taft", "Taylor",
  "Truman", "Tyler", "Van Buren", "Washington",
  "Wilson"};
string president =
  presidents.Where(p => p.StartsWith("Lin")).First();
Console.WriteLine(president);
```

```
string president =
  (from p in presidents
  where p.StartsWith("Lin")
  select p).First();
```

Școala
informală
de IT

LINQ to Objects

- WORKSHOP -

Examples

- **Counting the Occurences of a Word in a String**

```
string text = "Historically, the world of data …";
…
string searchTerm = "data";
string[] source = text.Split(
  new char[] { '.', '?', '!', ' ', ';', ':', ',' },
  StringSplitOptions.RemoveEmptyEntries);

// Use ToLower() to match both "data" and "Data"
var matchQuery =
  from word in text
  where word.ToLower() == searchTerm.ToLower()
  select word;
int wordCount =
  matchQuery.Count();
```

```
int wordCount = text.Select(
  w => w.toLower() ==
  searchTerm.ToLower()).Count();
```

Count the Occurences of a Word in a String

- WORKSHOP -

Querying Collections

Querying Collections

- What can we query?
  - Not everything can be queried by LINQ to Objects
  - The objects need to be a collection
  - It must implement the `IEnumerable<T>` interface
- The good news
  - Almost all standard collections in .NET Framework implements `IEnumerable<T>`

Querying Collections (2)

- **What can be queried using LINQ to Objects?**
  - **Arrays –** `T[]`
  - **Generic lists –** `List<T>`
  - **Generic dictionaries –** `Dictionary<K,V>`
  - **Strings –** `string`
  - **Other collections that implements** `IEnumerable<T>`

Querying Arrays

- Any kind of arrays can be used with LINQ
  - Can be even an untyped array of objects
  - Queries can be applied to arrays of custom objects
  - Example:

```
var titles =
  from b in books
  where b.Title.Contains("Action")
  select b.Title;
```

```
Book[] books = {
  new Book { Title="LINQ in Action" },
  new Book { Title="LINQ for Fun" },
  new Book { Title="Extreme LINQ" } };
var titles = books
  .Where(book => book.Title.Contains("Action"))
  .Select(book => book.Title);
```

Querying Arrays

- The previous example can be adapted to work with a generic list
  - `List<T>`, `LinkedList<T>`, `Queue<T>`, `Stack<T>`, `HashSet<T>`, etc.

```
List<Book> books = new List<Book>() {
  new Book { Title="LINQ in Action" },
  new Book { Title="LINQ for Fun" },
  new Book { Title="Extreme LINQ" } };
var titles = books
  .Where(book => book.Title.Contains("Action"))
  .Select(book => book.Title);
```

Querying Generic Lists

- WORKSHOP -

Querying Strings

- Although `System.String` may not be perceived as a collection at first sight
  - It actually is a collection, because it implements `IEnumerable<char>`
- String objects can be queried with LINQ to Objects, like any other collection

```
var count = "Non-letter characters in this string: 8"
  .Where(c => !Char.IsLetter(c))
  .Count();
Console.WriteLine(count);

// The result is: 8
```

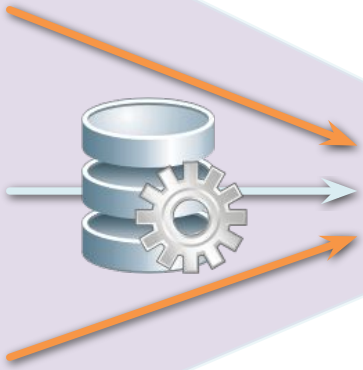
```
var count =
  (from c in "Non-letter..."
  where !Char.IsLetter(c)
  select c).Count();
```

LINQ Operations

Aggregation Operations

- An aggregation operation computes a single value from a collection of values
- Example of aggregation of field over a sequence of employees

Name	Salary
Bay Ivan	12500,00
Bat Rambo	13500,00
Baba Yaga	43300,00
Kiro the King	29800,00
Bay Mangal	25000,00
…	…



MAX(Salary)
125500,00

Aggregation Methods

- `Average()`
  - Calculates the average value of a collection
- `Count()`
  - Counts the elements in a collection
- `Max()`
  - Determines the maximum value in a collection
- `Sum()`
  - Sums the values in a collection

Aggregation Methods - Example

- **Count(<condition>)**

```
double[] temperatures =
  {28.0, 19.5, 32.3, 33.6, 26.5, 29.7};
int highTempCount = temperat
Console.WriteLine(highTempCo
// The result is: 2
```

```
var highTemp =
  (from p in temperatures
   where p > 30
   select p).Count();
```

- **Max()**

```
var highTemp =
  (from p in temperatures
   select p).Max();
```

```
double[] temperatures =
  {28.0, 19.5, 32.3, 33.6,
double maxTemp = temperatures.Max();
Console.WriteLine(maxTemp);
// The result is: 33.6
```

Projection

- Projection refers to the act of transforming the elements of a collection into a different type
  - The resulting type is defined by the developer
- Projection operators in LINQ
  - `Select` – projects single values that are based on a transform function
  - `SelectMany` – projects collections of values into a new collection containing all values

Projections - Example

- `Select(<transform-function>)`

```
List<string> words =
  new List<string>() { "an", "apple", "a", "day" };
var query =
  from word in words
  select word.Substring(0, 1);

foreach (string s in query)
{
  Console.Write("{0} ",s);
}
// The result is: a a a d
```

```
var query =
  words.Select(w =>
    w.Substring(0,1));
```

Projections - Example (2)

- **SelectMany(<multi-value-function>)**

```
string[] sentence = new string[] {
  "The quick brown",
  "fox jumped over",
  "the lazy dog"};

// SelectMany returns nine strings
// (sub-iterates the Select result)
IEnumerable<string> allWords =
  sentence.SelectMany(segment => segment.Split(' '));

foreach (var word in allWords)
  Console.WriteLine(" {0}", word);

// Result: The quick brown fox jumped over the lazy
dog
```

Projections

- WORKSHOP -

Conversions

- Converting a collection to a different type
  - Can change the type of the collection
  - Can change the type of the elements
- Conversion operations in LINQ queries are useful in a variety of applications
- For example:
  - `Enumerable.AsEnumerable<TSource>`
  - `Enumerable.OfType<(TResult)>`
  - `Enumerable.ToArray(TSource)`

Conversion Methods

- If start with "As"
  - Change the static type of the source collection but do not enumerate it
- If start with "To"
  - Enumerate the source collection and turn each item into the corresponding collection type

```
string[] towns =
  {"Sofia", "Plovdiv", "Varna", "Bourgas", "Pleven"};
List<string> list = towns.ToList();
```

Sorting

- A sorting operation orders the elements of a sequence based on one or more attributes
- Standard query operator
  - `OrderBy(...)`
  - `OrderByDescending(...)`
  - `ThenBy(...)` – performs a secondary sort in ascending order
  - `ThenByDescending(...)`
  - `Reverse(...)`

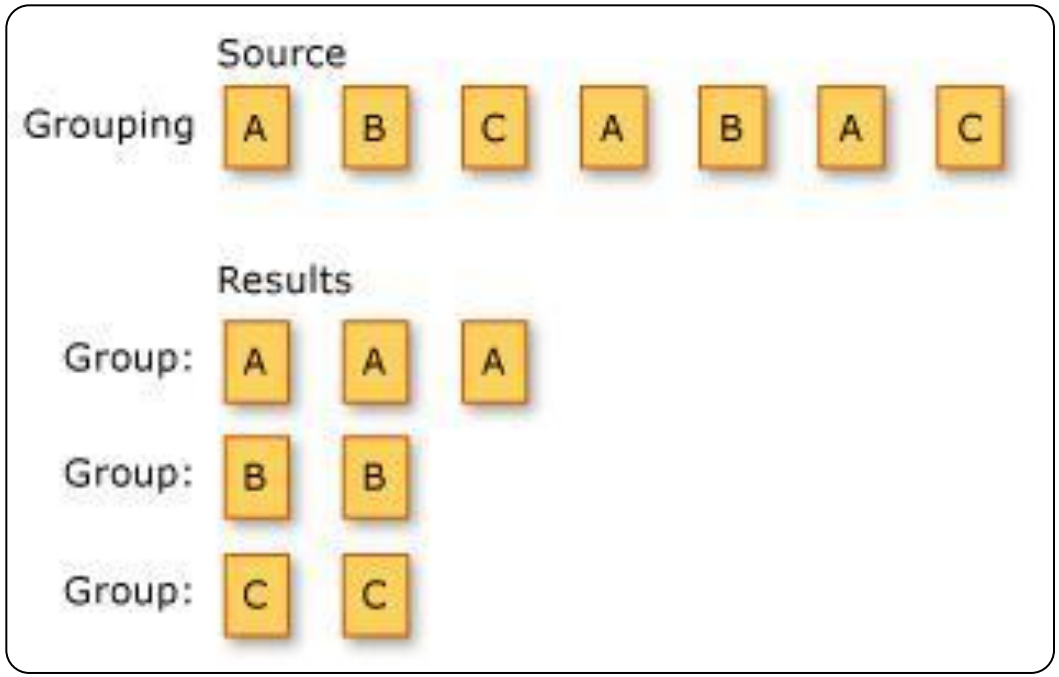
Sorting - Example

```
string[] words = { "Cluj", "Oradea",
  "Baia Mare", "Arad", "Timisoara" };
IEnumerable<string> query =
  from word in words
  orderby word.Length, word.Substring(0, 1) descending
  select word;

foreach (string str in query)
  Console.WriteLine(str);
/* The result is:
  Arad
  Baia Mare
  Cluj
  Oradea
  Timisoara
*/
```

```
var query =
  words.Select(word => word).
  OrderBy(word => word.Length).
  ThenByDescending(
    word => word.Substring(0, 1));
```

Grouping

- **Operation of putting data into groups**
  - **The elements in each group share a common value for some attribute**
- **Example**



Creating Groups and Maps

- `GroupBy()`
  - Groups elements that share a common attribute, called `key`
  - Each group is represented by a sequence of `IGrouping(TKey,TElement)` objects
- `ToLookup()`
  - Inserts elements into a `Lookup(TKey, TElement)` based on a key selector function
  - `Distinct()`
  - Returns distinct elements form a collection

Group By - Examples

```
var people = new[] {
  new { Name = "Ionel", Town = "Oradea"},
  new { Name = "Alin", Town = "Cluj"},
  new { Name = "Marin", Town = "Cluj"},
  new { Name = "Gigel", Town = "Oradea"}
};

var peopleByTowns =
  from p in people
  group p by p.Town;

foreach (var town in peopleByTowns)
{
  Console.Write("Town {0}: ", town.Key);
  foreach (var person in town)
    Console.Write("{0} ", person.Name);
  Console.WriteLine();
}
```

```
var peopleByTowns =
  people.GroupBy(t => t.Town);
```

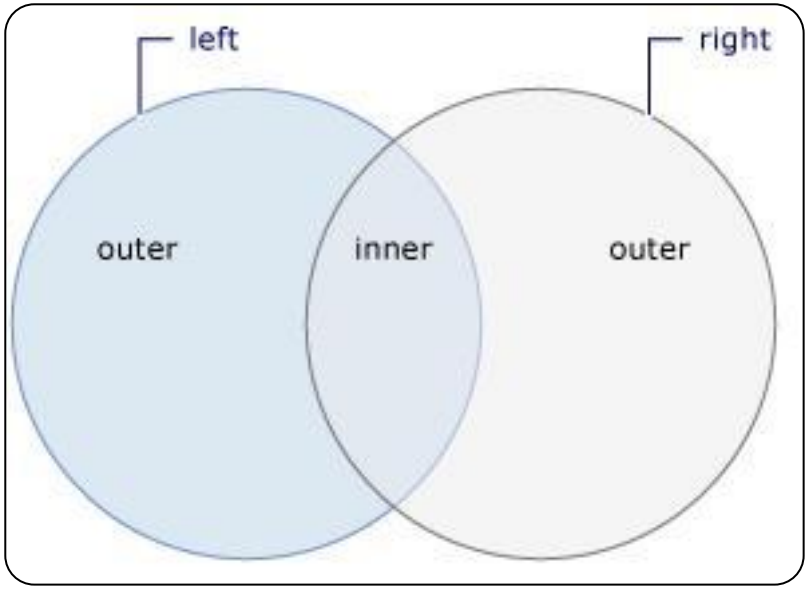
Group By - Examples (2)

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numberGroups =
  from n in numbers
  group n by n % 3;

foreach (var g in numb
{
  Console.Write("Remainder: {0} -> ", g.Key);
  foreach (var n in g)
    Console.Write("{0} ", n);
  Console.WriteLine();
}
// Remainder: 2 -> 5 8 2
// Remainder: 1 -> 4 1 7
// Remainder: 0 -> 3 9 6 0
```

```
var numberGroups =
  numbers.GroupBy(n => n % 3);
```

Joins

- Action of relating or associating one data source object with a second data source object
- The two data source objects are associated through a common value or attribute



Join Methods

- `Join`
  - Joins two sequences based on key selector function
  - And extracts the joined pairs of values
- `GroupJoin`
  - Joins two sequences based on key selector functions
  - And groups the resulting matches for each element

Join - Example

```
var owners = new[] {
  new { Name = "Koko", Town = "Plovdiv"},
  new { Name = "Pepi", Town = "Sofia"},
};

var pets = new[] {
  new { Name = "Sharo", Owner = owners[0] },
  new { Name = "Rex", Owner = owners[1] },
  new { Name = "Poohy", Owner = owners[0] },
};

var petsWithOwners =
  from o in owners
  join p in pets on o.Name equals p.Owner.Name
  select new { Owner = o.Name, Pet = p.Name };

foreach (var p in petsWithOwners)
  Console.WriteLine("{0} owned by {1}", p.Pet, p.Owner);
```

```
var petsWithOwners = owners.Join(pets,
  (o => o.Name), (p => p.Owner.Name),
  (o, p) => new {o.Name, p.Name });
```

Joins

- WORKSHOP -

Nested Queries

- The queries can be nested
- For example:
  - Suppose we have collections of `Person` and collections of `Role` objects
  - We want get all roles for given person (ID = 1)

```
var query = people
  .Where(p => p.ID == 1)
  .SelectMany(p => roles
    .Where(r => r.ID == p.RoleID)
    .Select(r =>
      new { p.FirstName, p.LastName, r.Role }));
```

Nested Queries

- WORKSHOP -