Of Lambdas and LINQ

James Michael Hare
2012 Visual C# MVP
Application Architect
Scottrade

August 3rd, 2012

http://www.BlackRabbitCoder.net

Twitter: @BlkRabbitCoder

Contact Information

Me:

- Blog: http://www.BlackRabbitCoder.net
- Twitter: @BlkRabbitCoder

Information on Scottrade Careers:

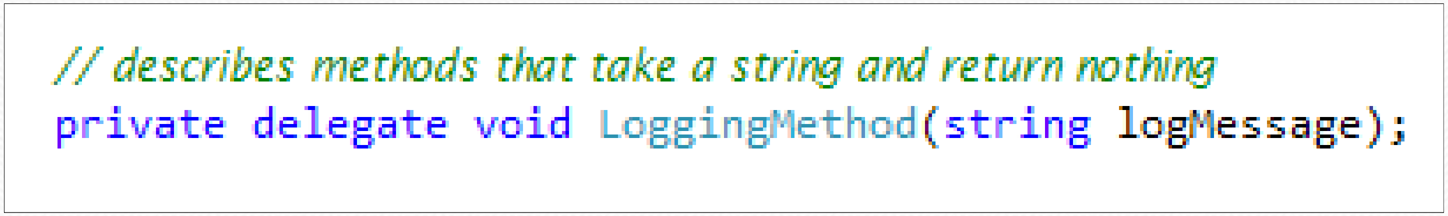
- http://jobs.scottrade.com
- Twitter: @scottradejobs

Introduction

- LINQ and lambdas are becoming much more prevalent in our codebases and it's important for all of our developers to have ***at least*** a basic understanding.
- This presentation is designed to be a brief introduction to both lambda expressions and the more common LINQ extension methods.
- In addition, we will give a brief overview of the optional query syntax.

A brief history…

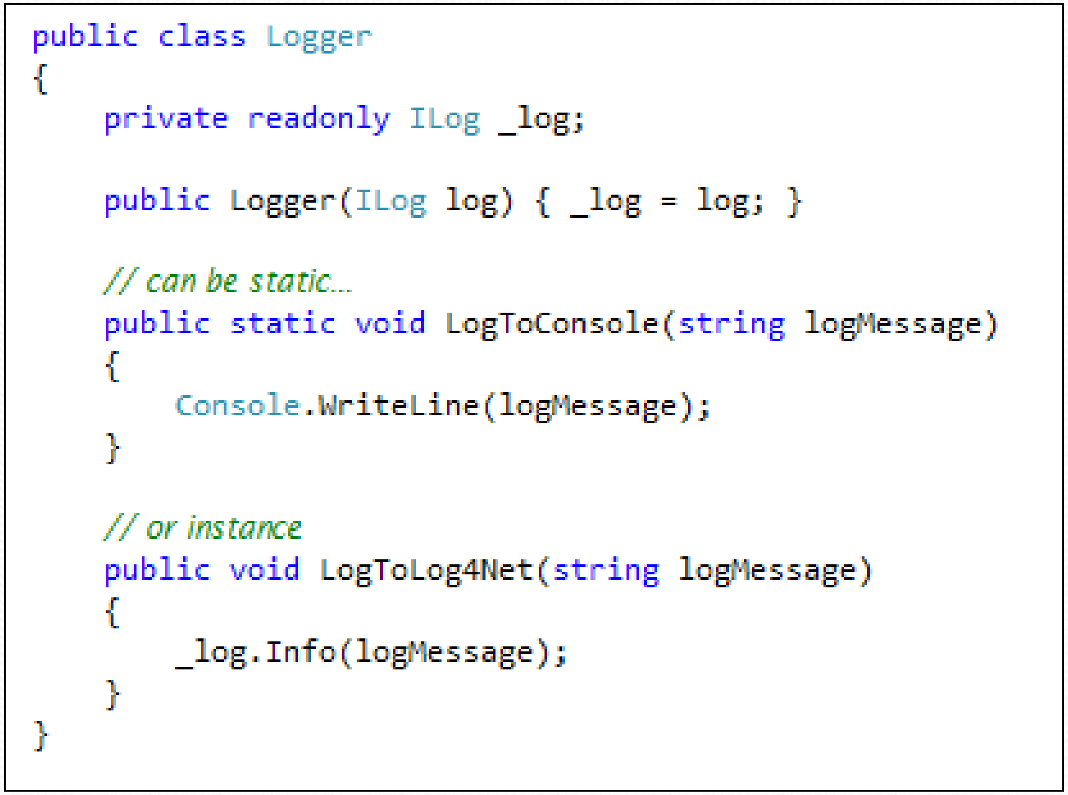
- Before we look at lambdas, let's take a look at how delegates work in C#.
- A delegate is similar to a pointer to a function in C++.
- Delegate describe the signature of a method.
- Define delegates using the keyword **`delegate`**.
- The following delegate describes methods that have no return value and take a string parameter:



```
// describes methods that take a string and return nothing
private delegate void LoggingMethod(string logMessage);
```

Defining matching methods

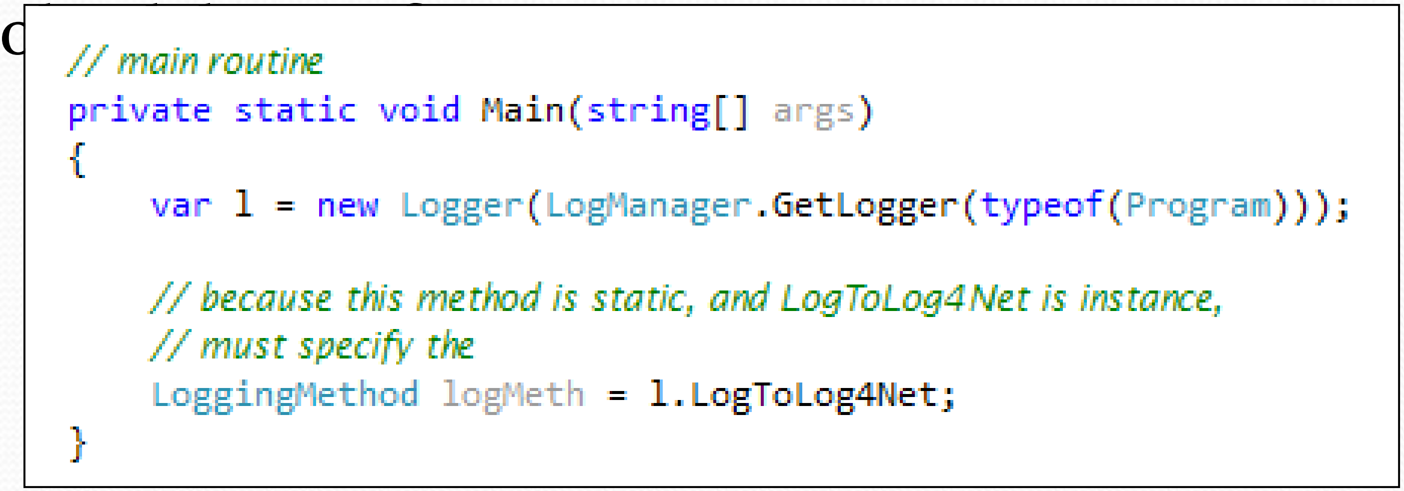
- Method can be static or instance.
- These methods match our **`LoggingMethod`** delegate



```
public class Logger
{
    private readonly ILog _log;

    public Logger(ILog log) { _log = log; }

    // can be static...
    public static void LogToConsole(string logMessage)
    {
        Console.WriteLine(logMessage);
    }

    // or instance
    public void LogToLog4Net(string logMessage)
    {
        _log.Info(logMessage);
    }
}
```

Assigning instance methods

- Delegate instances can be assigned method references.
- If instance method assigned, provide the instance to invo

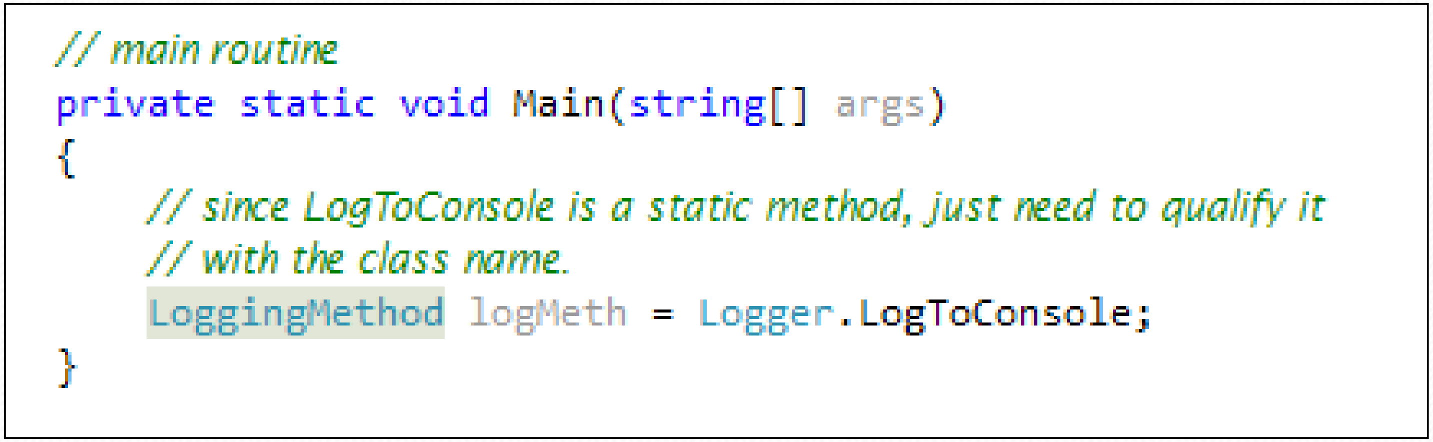


```
// main routine
private static void Main(string[] args)
{
    var l = new Logger(LogManager.GetLogger(typeof(Program)));

    // because this method is static, and LogToLog4Net is instance,
    // must specify the
    LoggingMethod logMeth = l.LogToLog4Net;
}
```

- The instance may be omitted if assigned from an instance member of the same instance.

Assigning static methods

- Since static methods require no instance to be invoked upon, just the class name.

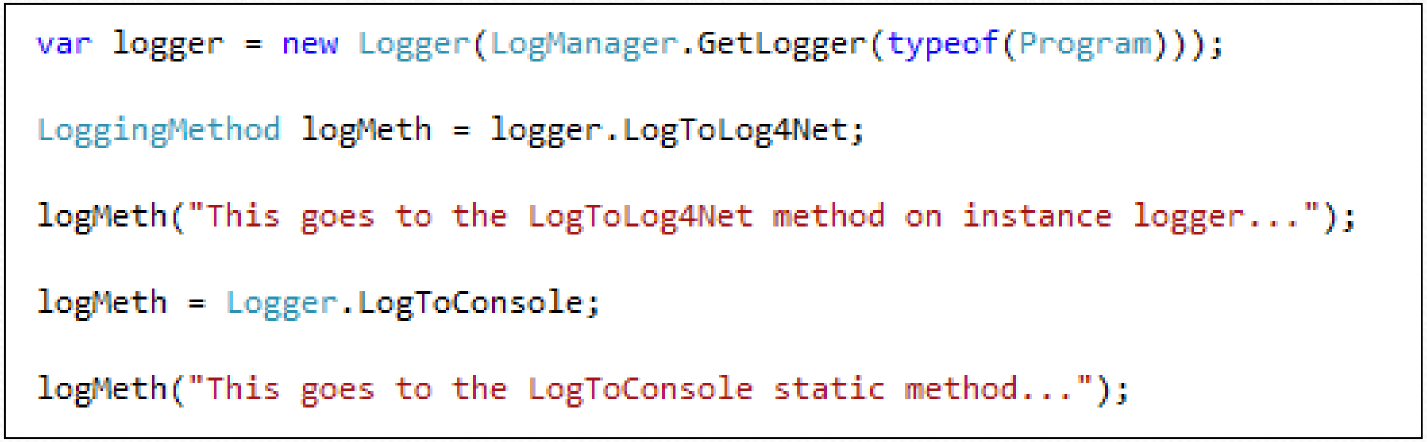


```
// main routine
private static void Main(string[] args)
{
    // since LogToConsole is a static method, just need to qualify it
    // with the class name.
    LoggingMethod logMeth = Logger.LogToConsole;
}
```

- The class name may be omitted if the current class is the class that contains the method to be assigned.

Invoking a delegate

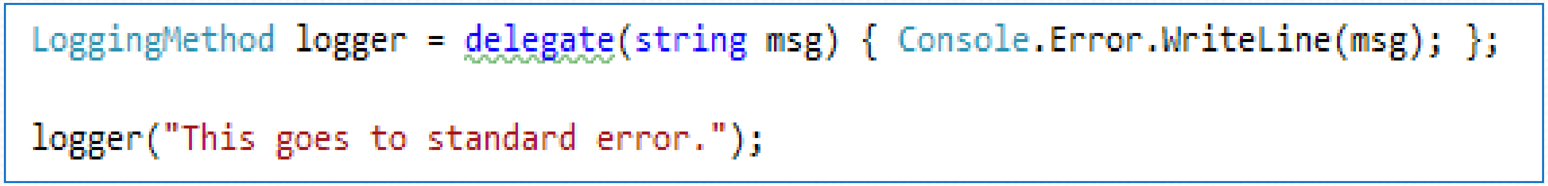
- The beauty of delegates is that they allow methods to be assigned to variables.
- This means you can easily change the behavior of a piece of code without needing to subclass.
- You invoke by calling the delegate like the method:



```
var logger = new Logger(LogManager.GetLogger(typeof(Program)));

LoggingMethod logMeth = logger.LogToLog4Net;

logMeth("This goes to the LogToLog4Net method on instance logger...");

logMeth = Logger.LogToConsole;

logMeth("This goes to the LogToConsole static method...");
```

Anonymous methods

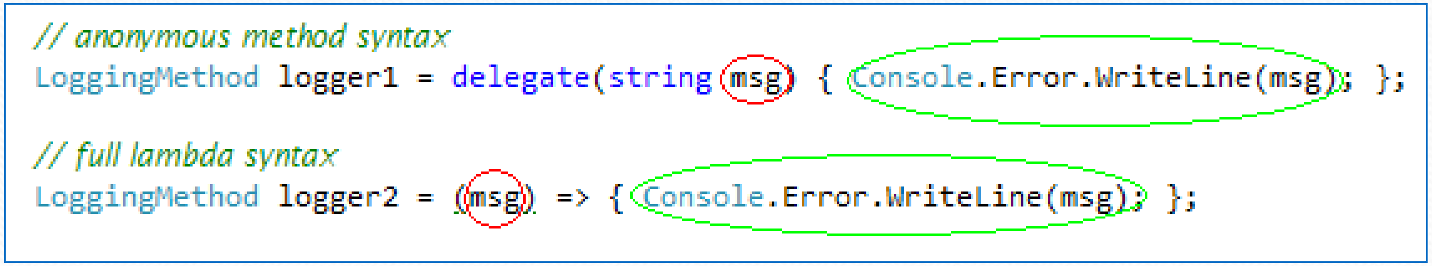
- The only problem with early .NET was that all delegate targets had to be methods.
- This means you’d have to create full methods for even the simplest task.
- The anonymous method syntax introduced in .NET 2.0 allowed writing methods on the fly at the point they would be assigned to a delegate:



```
LoggingMethod logger = delegate(string msg) { Console.Error.WriteLine(msg); };

logger("This goes to standard error.");
```

Lambda expressions

- .NET 3.0 added lambda expression support for writing anonymous methods.
- Lambda expression syntax can be seen as a more concise form of the anonymous method syntax:



```
// anonymous method syntax
LoggingMethod logger1 = delegate(string msg) { Console.Error.WriteLine(msg); };

// full lambda syntax
LoggingMethod logger2 = (msg) => { Console.Error.WriteLine(msg); };
```

- The full lambda expression syntax is:

```
(parameters) => { statement(s) }
```

- The lambda operator “=>” is pronounced “goes to”.

More on lambda parameters

- Specifying type is optional when can be inferred

```
(string s) => { return s.Length; }
(s) => { return s.Length; }
```

- If no parameters, the parenthesis are empty:

```
() => { statement(s) }
```

- If one parameter, the parenthesis are optional:

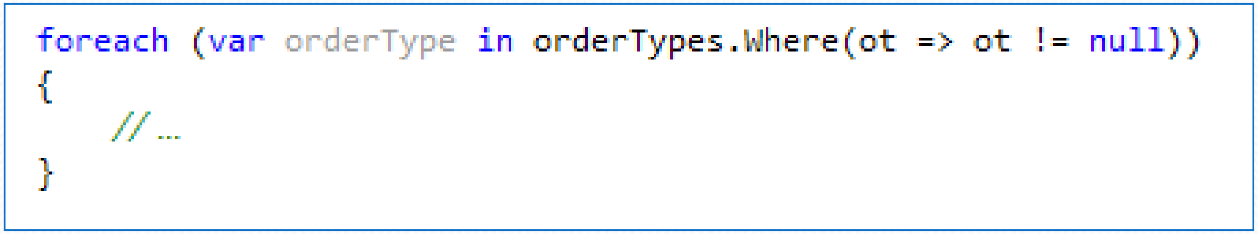
```
(s) => { return s.Length; }
s => { return s.Length; }
```

- If several parameters, use parenthesis and commas:

```
(x,y) => { return x > y; }
```

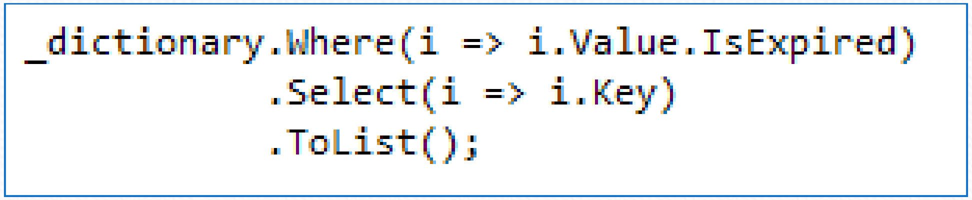
Lambda parameter naming

- The standard convention is a short identifier whose meaning can be inferred from context:



```
foreach (var orderType in orderTypes.Where(ot => ot != null))
{
    //…
}
```

- Names are local to the lambda and can be reused if desired to represent an item flowing down a chain.



```
_dictionary.Where(i => i.Value.IsExpired)
           .Select(i => i.Key)
           .ToList();
```

More on lambda body

- If body more than one statement, must separate with semicolons and enclose in braces:

```
s => {
        Console.WriteLine(s);
        Console.Out.Flush();
     }
```

- If one statement only, can omit semicolon and braces:

```
s => { Console.Error.WriteLine(s); }
s => Console.Error.WriteLine(s)
```

- If only an expression to evaluate, can omit **return**:

```
s => return s.Length
s => s.Length
```

The generic delegates

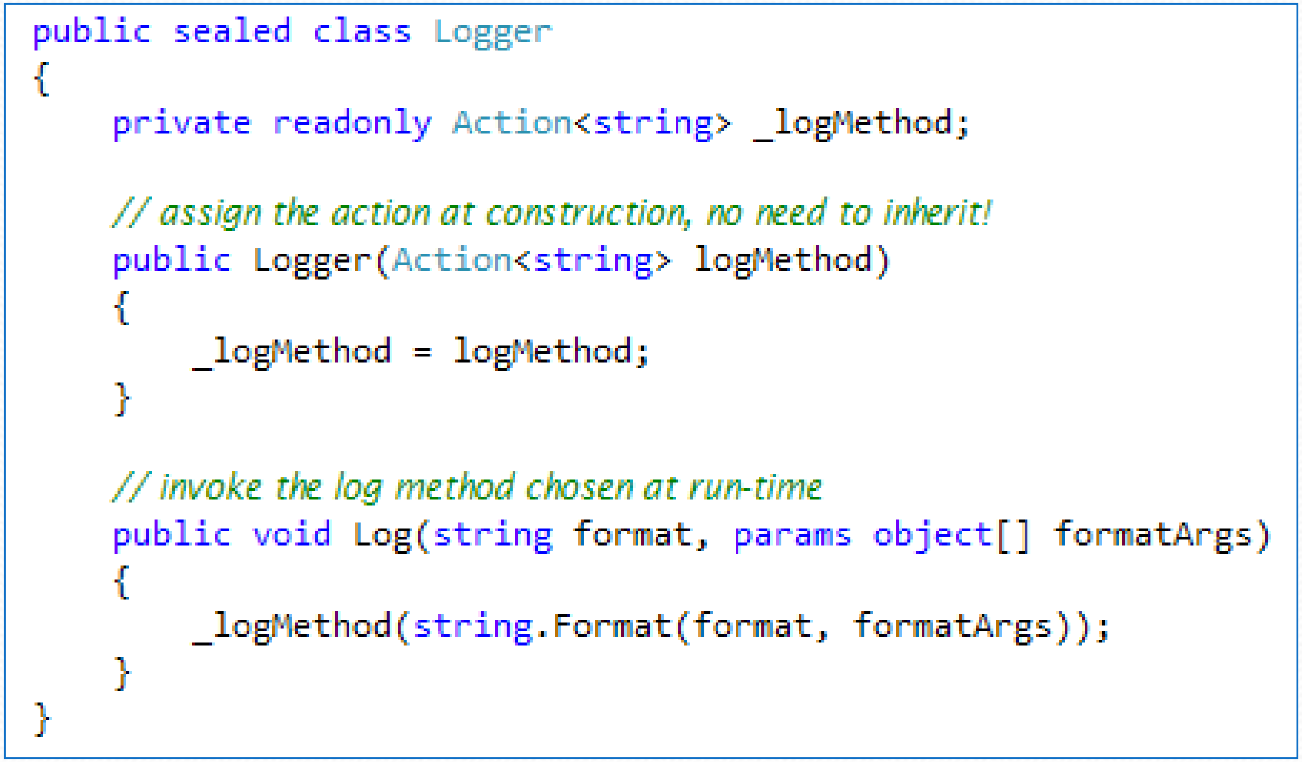
- Delegates can be useful for specifying pluggable behavior at runtime instead of compile time.
- This eliminates much of the need of inheritance in places it was used to specify different behaviors.
- .NET 2.0 added some common generic delegates:
  - **Predicate<T>** - *bool fx(T)* – specifies a condition to be run against an item which returns **true** or **false**.
  - **Action<T>** - *void fx(T)* – specifies an action to be performed on an item, no result.
  - **Action** has other versions as well for varying number of parameters from 0 to 16.

More generic delegates

- .NET 3.5 added a new generic delegate for more general delegate specification:
  - **Func<TResult>** - *TResult fx()* – specifies a delegate that takes no parameters and returns a **TResult**.
  - **Func<T, TResult>** - *TResult fx(T)* – specifies a delegate that takes on parameter and returns a **TResult.**
  - **Func<T1, T2, TResult>** - *TResult fx(T1, T2)* – specifies a delegate that takes 2 parameters and returns **TResult.**
  - **Func<T1…, TResult>** can support up to 16 parameters.
- **Func<T, bool>** is equivalent to **Predicate<T>**.

Pluggable behavior

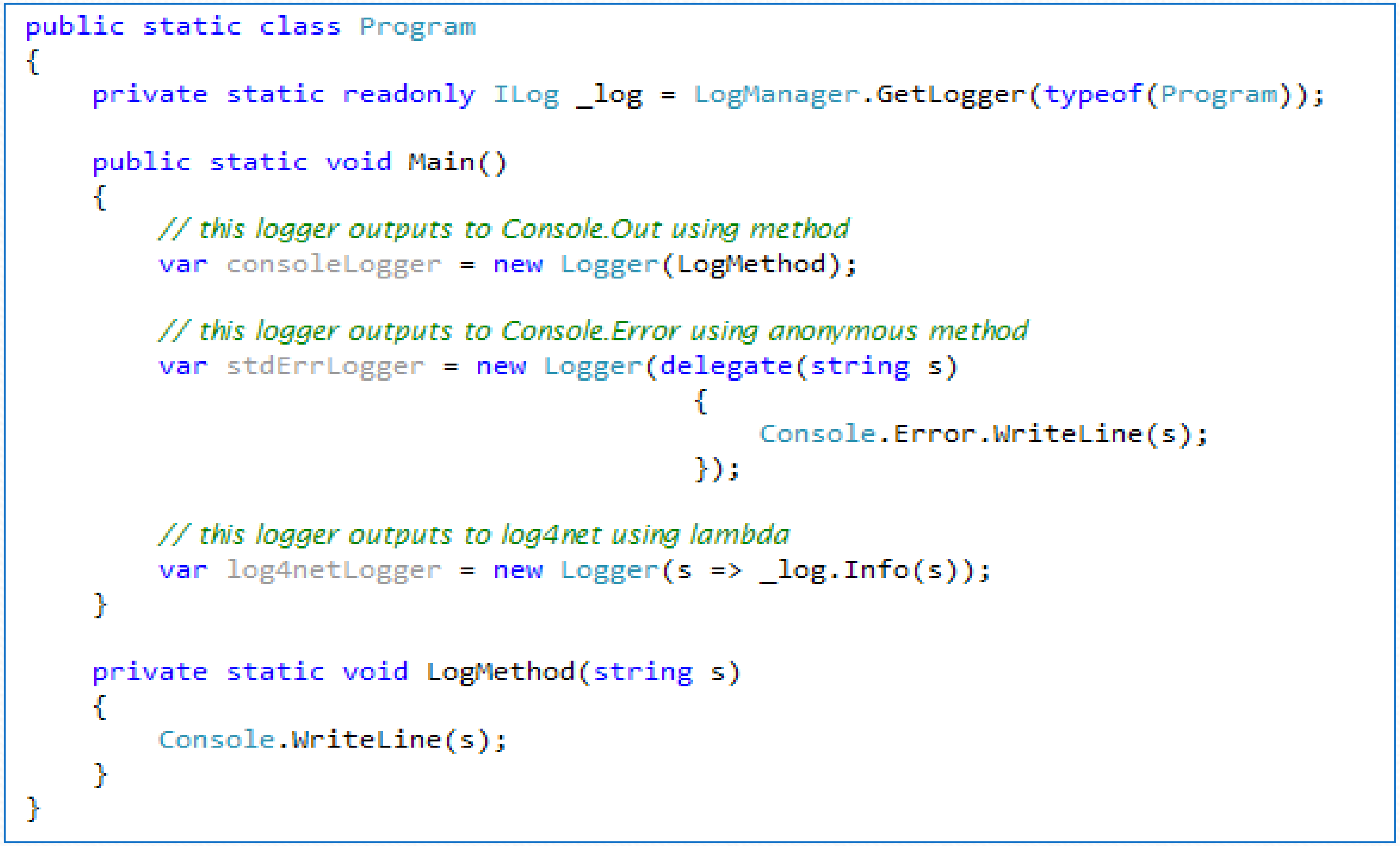
- Delegates allow you to create classes and algorithms with pluggable behavior without inheritance:



```
public sealed class Logger
{
    private readonly Action<string> _logMethod;

    // assign the action at construction, no need to inherit!
    public Logger(Action<string> logMethod)
    {
        _logMethod = logMethod;
    }

    // invoke the log method chosen at run-time
    public void Log(string format, params object[] formatArgs)
    {
        _logMethod(string.Format(format, formatArgs));
    }
}
```

Pluggable behavior

- Behaviors can be changed on the fly:



```
public static class Program
{
    private static readonly ILog _log = LogManager.GetLogger(typeof(Program));

    public static void Main()
    {
        // this logger outputs to Console.Out using method
        var consoleLogger = new Logger(LogMethod);

        // this logger outputs to Console.Error using anonymous method
        var stdErrLogger = new Logger(delegate(string s)
                                        {
                                            Console.Error.WriteLine(s);
                                        });

        // this logger outputs to log4net using lambda
        var log4netLogger = new Logger(s => _log.Info(s));
    }

    private static void LogMethod(string s)
    {
        Console.WriteLine(s);
    }
}
```

LINQ

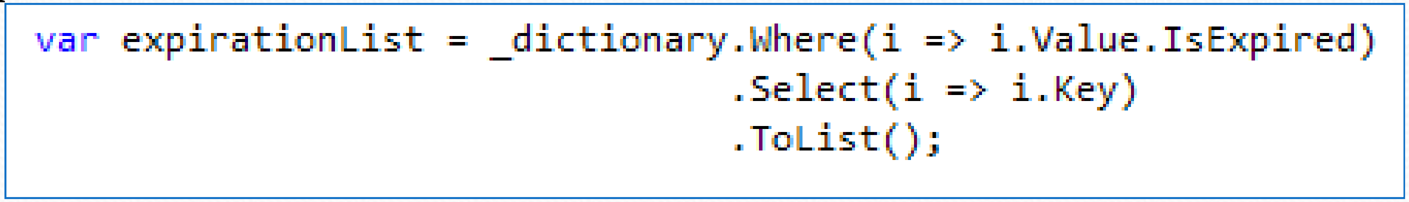
- LINQ = Language INtegrated Query.
- LINQ is a new query language and class libraries.
- LINQ class libraries can be used with or without the query syntax as desired.
- Most LINQ extension methods operate on sequences of data that implement **IEnumerable<T>**.
- Most LINQ extension methods specify behavior with generic delegates (**Action**, **Func**, etc).
- Lambda expressions are a perfect way to supply concise definitions to LINQ.

LINQ: Common behaviors

- Most LINQ extension methods cannot be called on a **null** sequence (throws **ArgumentNullException).**
- You can specify behaviors using any valid means (lambda expressions, anonymous methods, ordinary methods, or method groups).
- You can use the LINQ methods on any sequences that implement **IEnumerable<T>** including **List<T>**, **T[]**, **HashSet<T>**, iterators, etc.
- Many of the LINQ methods use deferred execution (may not compute query until the data is needed).

LINQ: Chaining

- LINQ operations can be chained together.
- Each method call is independent and applies to the previous result in the chain:



```
var expirationList = _dictionary.Where(i => i.Value.IsExpired)
                                .Select(i => i.Key)
                                .ToList();
```

  - ***Where()*** *narrows sequence to only those items whose* ***Value*** *property has an* ***IsExpired*** *property ==* ***true.***
  - ***Select()*** *transforms the expired items from the* ***Where()*** *to return a sequence containing only the* ***Key*** *properties.*
  - ***ToList()*** *takes the sequence of expired* ***Keys*** *from the* ***Select()*** *and returns then in a* ***List<string>.***

LINQ: Selecting and filtering

- **Where()** – Narrows sequence based on condition

```
orders.Where(o => o.Type == OrderType.Buy)
```

  - *Sequence only containing orders with order type of buy.*

- **Select()** – Transforms items in sequence or returns part(s) of items.

```
orders.Select(o => o.Type)
```

  - *Sequence only containing the types of the orders.*

- These two are sometimes confused, remember not to use Select to try to filter or you get a sequence of bool.

```
orders.Select(o => o.Type == OrderType.Buy)
```

LINQ: Consistency checks

- **All()** – **True** if all items satisfy a predicate:

  ```
  requests.All(r => r.IsValid)
  ```

  - *True if **IsValid** returns **true** for all items.*

  ```
  employees.All(e => e.Ssn != null)
  ```

  - *True if **Ssn** is not null for all items.*

- **Any()** – **True** if at least one satisfies predicate:

  ```
  requests.Any(r => !r.IsValid)
  ```

  - *True if any item returns **IsValid** of **false**.*

  ```
  results.Addresses.Any()
  ```

  - *True if Addresses contains at least one item.*

LINQ: Membership and count

- **Contains()** – **True** if sequence contains item.

  ```
  orderTypes.Contains(“Buy”)
  ```

  - *True if contains string “buy” based on default string comparer.*

  ```
  orderTypes.Contains(“buy”, comparer)
  ```

  - *True if contains “buy” based on given string comparer.*

- **Count()** – Number of items (or matches) in sequence.

  ```
  requests.Count(r => !r.IsValid)
  ```

  - *Count of items where **IsValid** returns false*

  ```
  requests.Count()
  ```

  - *Count of all items in sequence*

LINQ: Combine and reduce

- **Distinct()** – Sequence without duplicates.

  ```
  orderIds.Distinct()
  ```

  - *A list of all unique order ids based on default comparer.*

- **Concat()** – Concatenates sequences.

  ```
  defaultValues.Concat(otherValues)
  ```

  - *Sequence containing defaultValues followed by otherValues.*

- **Union()** – Concatenates without duplicates.

  ```
  defaultValues.Union(otherValues)
  ```

  - *Sequence with items from defaultValues followed by items from otherValues without any duplicates.*

LINQ: First item or match

- **First()** – Returns first item or match, throws if none.

```
orders.First()
```

  - *Returns first order, throws if none.*

```
orders.First(o => o.Type == OrderType.Buy)
```

  - *Returns first buy order, throws if no buy orders.*

- **FirstOrDefault()** – Returns first item, default if none.

```
orders.FirstOrDefault()
```

  - *Returns first order, default if no orders.*
  - *Default based on type (null for reference, 0 for numeric, etc)*

```
orders.FirstOrDefault(o => o.Id > 100)
```

  - *Returns first order with id > 100, or default if no match.*

LINQ: Other items and matches

- **Last() / LastOrDefault()**
  - Similar to First except returns last match or item in list.
  - Default version returns default of type if no match.
- **ElementAt() / ElementAtOrDefault()**
  - Returns element at the given position (zero-indexed).
  - Default version returns default of type if position > count.
- **Single() / SingleOrDefault()**
  - Very similar to First except ensures that only one item exists that matches.

LINQ: Set operations

- **Except()** – Set difference, subtracts sequences

  ```
  snackFoods.Except(healthyFoods)
  ```

  - *Returns first sequence minus elements in the second.*
  - *Sequence of snack foods that are not healthy foods.*
- **Intersect()** – Set intersection, common sequence

  ```
  snackFoods.Intersect(healthyFoods)
  ```

  - *Returns first sequence minus elements in the second.*
  - *Sequence of foods that are snack foods and healthy foods.*
- **Union()** – Combines sequences without duplicates.

  ```
  snackFoods.Union(healthyFoods)
  ```

  - *Returns both sequences combined with duplicates removed.*

LINQ: Aggregation operations

- **Sum()** – Adds all in from a sequence

```
orders.Sum(o => o.Quantity)
```

- **Average()** – Agerage of all items in a sequence

```
orders.Average(o => o.Quantity)
```

- **Min()**– Finds minimum item in sequence

```
orders.Min(o => o.Quantity)
```

- **Max()** – Finds maximum item in sequence

```
orders.Max(o => o.Quantity)
```

- **Aggregate()** – Performs custom aggregation

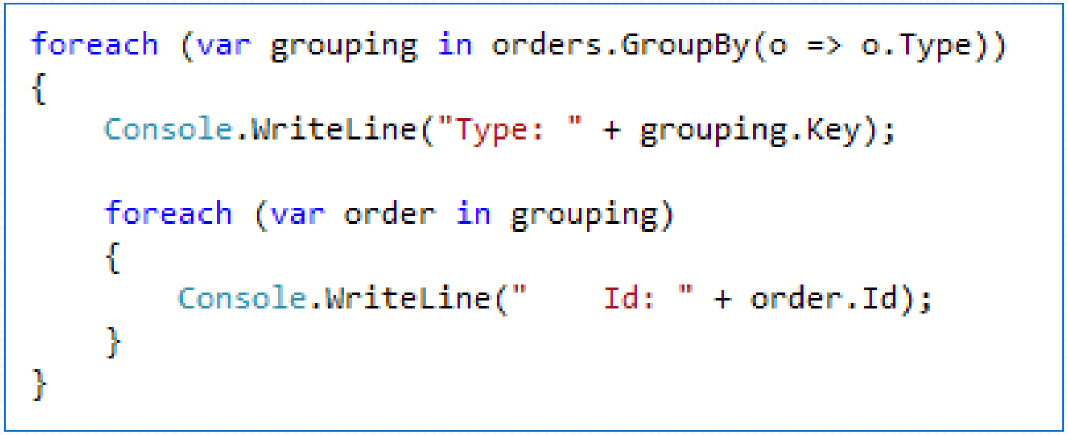
```
orders.Aggregate(0.0, (total, o) =>
    total += o.Quantity * o.Price)
```

LINQ: Grouping

- **GroupBy()** – Groups a sequence by criteria

  `orders.GroupBy(o => o.Type)`

  - *Organizes the sequence into groups of sequences.*
  - *In this case, organizes orders into sequences by order type.*
  - *Each element in resulting sequence is grouping with* ***Key*** *being the group criteria and the grouping being a sequence:*



```
foreach (var grouping in orders.GroupBy(o => o.Type))
{
    Console.WriteLine("Type: " + grouping.Key);

    foreach (var order in grouping)
    {
        Console.WriteLine("    Id: " + order.Id);
    }
}
```

LINQ: Ordering

- **OrderBy()** – Orders in ascending order

```
orders.OrderBy(o => o.Id)
```

  - *Orders in ascending order by order id.*

- **OrderByDescending()** – Orders in descending order

```
Orders.OrderByDescending(o => o.Id)
```

  - *Orders in descending order by order id.*

- **ThenBy()** – Adds additional ascending criteria

```
Orders.OrderBy(o => o.Id).ThenBy(o => o.Type)
```

  - *Orders in ascending order by id and when ids are same ordered by order type as a secondary ordering.*

- **ThenByDescending()** – As above, but descending.

LINQ: Exporting to Collection

- **ToArray()** – Returns sequence as an array.

  ```
  orders.OrderBy(o => o.Id).ToArray()
  ```

  - *Retrieves orders ordered by id into an array.*

- **ToList()** – Returns sequence as a **List**

  ```
  Orders.OrderBy(o => o.Id).ToList()
  ```

  - *Retrieves orders ordered by id into a List<Order>.*

- **ToDictionary()** – Returns sequence as **Dictionary**

  ```
  Orders.ToDictionary(o => o.Id)
  ```

  - *Retrieves orders in a dictionary where the Id is the key.*

- **ToLookup()** – Returns sequence of **IGrouping**

  ```
  Orders.ToLookup(o => o.Type)
  ```

  - *Retrieves orders into groups of orders keyed by order type.*

Query Syntax

- The query syntax is an alternative way to write LINQ expressions.
- Microsoft tends to recommend the query syntax as it tends to be easier to read.
- Ultimately, it is reduced to the same method calls we’ve already seen, so can use either.
- There are some methods which are not directly callable using the query syntax.

Query Syntax

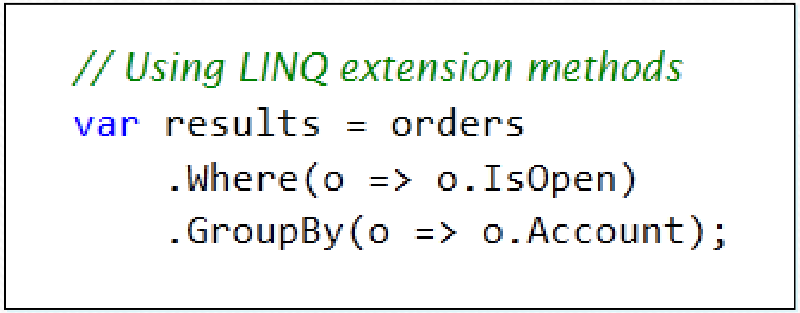
- All queries start with the **from** clause in the form:
  - `from o in orders`
    - *Declares variable representing current and source sequence.*
    - *Similar to **foreach(var o in orders)** except deferred.*
- All queries must end with a **select** or **group** clause:
  - `from o in orders select o.Id`
    - *Projects output same as the **Select()** extension method.*
  - `from o in orders group o by o.Id`
    - *Groups output same as the **GroupBy()** extension method.*

Query Syntax

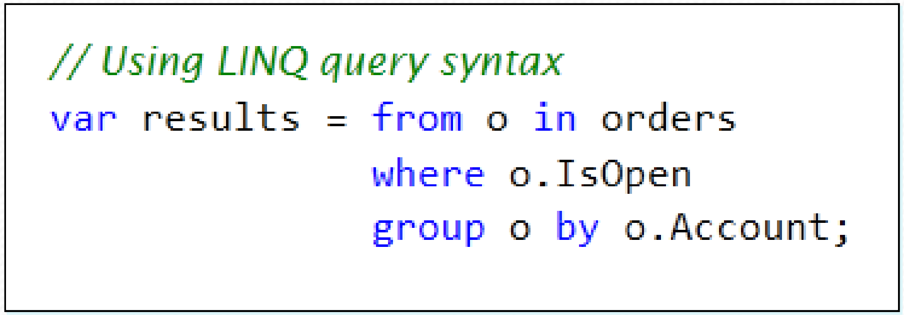
- In between the **from** and the **select/group** clauses, you can filter, order, and join as desired:
  - **where** – *filters to only matching objects, like* ***Where().***
  - **into** – *allows storage of results from a* ***groupby****,* ***join****, or* ***select*** *for use in later clauses.*
  - **order** – *sorts the results, can use* ***ascending*** *or* ***descending*** *modifiers, like the* ***OrderBy()*** *method family.*
  - **join** – *joins two sequences together, like* ***Join().***
  - **let** – *allows storage of result from sub-expression for use in later clauses.*

Same Query, Different Syntax

- These two queries are the same, just different syntax:



```
// Using LINQ extension methods
var results = orders
    .Where(o => o.IsOpen)
    .GroupBy(o => o.Account);
```



```
// Using LINQ query syntax
var results = from o in orders
              where o.IsOpen
              group o by o.Account;
```

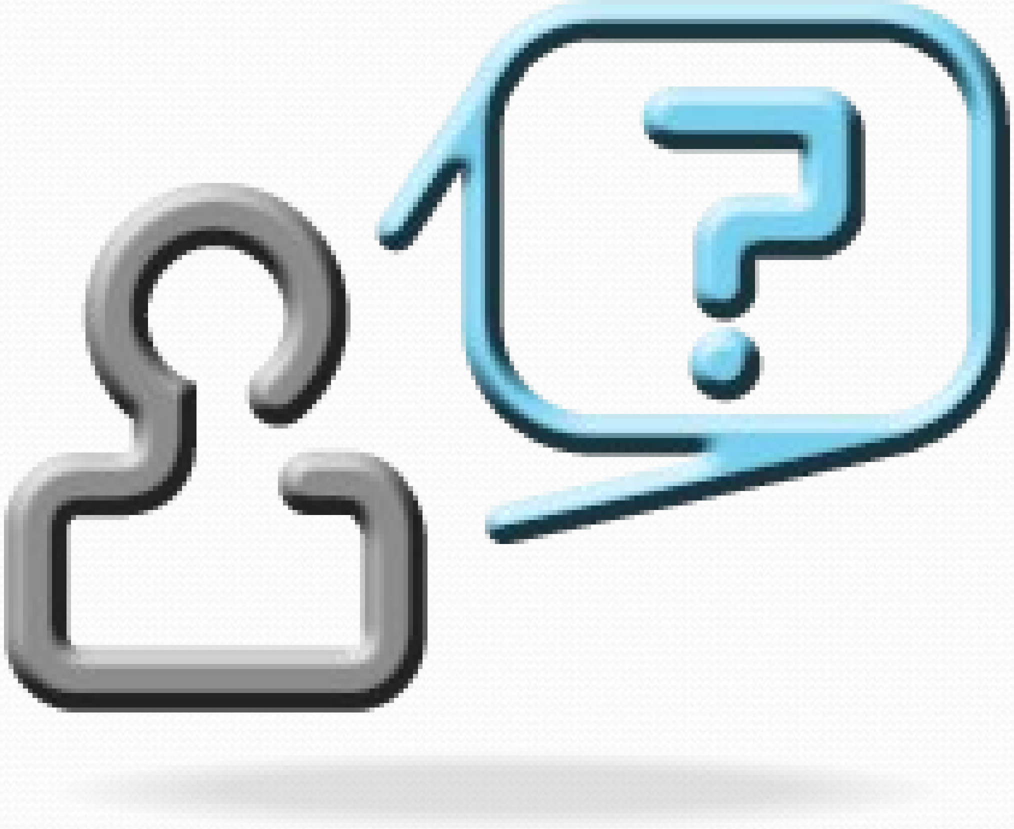
Pros and cons

- Pros:
  - Lambda expressions are much more concise than full methods for simple tasks.
  - LINQ expressions simplifies implementation by using .NET libraries that are optimized and fully tested.
  - Once you learn the lambdas and LINQ, ultimately the code is much more readable and easier to maintain.
- Cons:
  - The lambda syntax and LINQ methods take some getting used to at first *(short ramp-up time)*.

Summary

- Delegates help create more modular code without the need for inheritance.
- Lambdas are just an easy way to assign an anonymous method to a delegate.
- LINQ introduces a great class library for performing common operations on sequences of data.
- Using lambdas and LINQ can help reduce code size and improve code readability and maintainability.
- Care should be taken to make lambda expressions concise and meaningful while still readable.

Questions?



Platinum Sponsors

Microsoft

OAKWOOD

Aspect

signature CONSULTANTS

SYLLOGISTEKS

Gold Sponsors

Talentporte the source for IT talent

perceptivesoftware DEVELOPER NETWORK

Missouri State UNIVERSITY

TDK technologies

LRS

Perficient

POWERDNN DOTNETNUKE DONE RIGHT!

INFRAGISTICS DESIGN / DEVELOP / EXPERIENCE

HELLO my name is busyevent

VantageLinks

Byrne Software TECHNOLOGIES, INC.

Scottrade

KELLYMITCHELL

ArchitectNow

FAST

Washington University in St. Louis Information Services & Technology

CENTRIQ TRAINING

SERVERSILO Real Servers, Real Value

NEXTGEN CREATIVE. COLLABORATIVE. CONTEMPORARY.

iBridge Solutions LLC

USI Ungerboeck Software INTERNATIONAL

EQUIFAX

AppDynamics APPLICATION MANAGEMENT FOR THE CLOUD GENERATION

DAUGHERTY Business Solutions

CAIT Center for the Application of Information Technology Washington University in St. Louis

twilio CLOUD COMMUNICATIONS

XIOLINK

ADAPTIVE SOLUTIONS GROUP

ComponentOne a division of GrapeCity

ADVANCED resources

telerik deliver more than expected

Covenant TECHNOLOGY PARTNERS

PR Preferred Resources

Silver Sponsors

TRANSITIONS

DOTNETNUKE

discount ASP.net ASP.NET Web Hosting Team Foundation Server Hosting

jacobson staffing CONNECTING PEOPLE AND TECHNOLOGY

LogicNP Software The power of Components

pluralsight hardcore developer training

stackoverflow

MINDSCAPE