C# and LINQ

Language Features for LINQ and Beyond

Overview

- **Digging Into C# Features For LINQ**
  - Extension methods
  - Lambda Expressions
  - Expression Trees
  - Query Expressions
  - Type Inference
  - Anonymous Types
  - Partial Methods

Some C# History

Integrating data queries into C# has been a goal for years.

"LINQ" on the whiteboard

```
sequence<Employee> scotts =
    employees.where(Name == "Scott");
```

"LINQ" in C# 2.0

```
IEnumerable<Employee> scotts =
    EnumerableExtensions.Where(employees,
          delegate(Employee e)
          {
              return e.Name == "Scott";
          });
```

Syntax Problems

- Code doesn't look like a query
- Static classes hide operations
- Anonymous methods are verbose
- Projected types require definitions
- Type names clutter the code

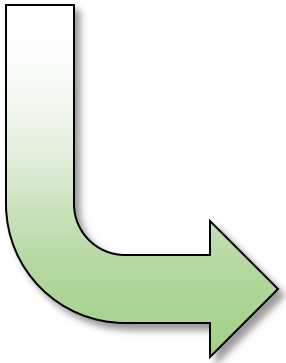
"LINQ" in C# 2.0

```
IEnumerable<Employee> scotts =
    EnumerableExtensions.Where(employees,
          delegate(Employee e)
          {
              return e.Name == "Scott";
          });
```

Evolving The Language

"LINQ" in C# 2.0

```
IEnumerable<Employee> scotts =
    EnumerableExtensions.Where(employees,
          delegate(Employee e)
          {
              return e.Name == "Scott";
          });
```



Today's LINQ

```
var scotts =
     from e in employees
     where e.Name == "Scott"
     select e;
```

Extension Methods

- **Create the illusion of new methods on an existing type**
  - Even sealed classes and interfaces

```
// Extending string via static methods (pre C# 3.0)
public static class StringUtils
{
    static public double ToDouble(string data)
    {
        double result = double.Parse(data);
        return result;
    }
}
```

```
string text = "43.35";
double data = StringUtils.ToDouble(text);
```

Defining Extension Methods

- First parameter of an extension method uses *this* modifier
- Can invoke static method with instance syntax

```
public static class StringExtensions
{
    static public double ToDouble(this string data)
    {
        double result = double.Parse(data);
        return result;
    }
}
```

```
string text = "43.35";
double data = text.ToDouble();
```

Using Extension Methods

- **Must define inside a non-generic, static class**
- **Extension methods are still external, static methods**
  - No access to private state or methods of target object
- **Cannot hide, replace, or override instance methods**
  - Compiler only looks for extension methods when it finds no compatible instance method
- **Must import namespace for extension method**
  - Namespace design is important

Extension Methods and LINQ

- **System.Linq defines extension methods for IEnumerable<T> and IQueryable<T>**
  - Standard query operators like Select, OrderBy, Where, and many more

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TSource> Where<TSource>(
                                this IEnumerable<TSource> source,
                                Func<TSource, bool> predicate) …
    }
    ...
}
```

```
string[] cities = {"Boston", "Los Angeles",
                   "Seattle", "London", "Hyderabad"};

IEnumerable<string> filteredList =
    cities.Where(delegate(string s)
                        { return s.StartsWith("L"); });
```

Shrinking Delegate Creation

Named method

```
IEnumerable<string> filteredList =
                cities.Where(StartsWithL);

public bool StartsWithL(string name)
{
    return name.StartsWith("L");
}
```

Anonymous method

```
IEnumerable<string> filteredList =
    cities.Where(delegate(string s)
                        { return s.StartsWith("L"); });
```

Lambda Expression

```
IEnumerable<string> filteredList =
    cities.Where(s => s.StartsWith("L"));
```

Lambda Expression Essentials

```
IEnumerable<string> filteredList =
    cities.Where(s => s.StartsWith("L"));
```

- **Takes a functional view of the world**
- **Concise syntax for defining an anonymous function**
  - Doesn't require the *delegate* keyword
  - Doesn't require the *return* keyword
  - Compiler uses type inference whenever possible
- **Introduces the *goes to* operator =>**
  - Left hand side is function signature
  - Right hand side is an expression or statement block

Constructing Lambda Expressions

- **Parentheses and types are often optional in signature**
  - No parentheses required when using a single, implicitly typed parameter
- **Statement blocks possible using { and }**
  - Can introduce local variables
  - But – lambda expressions are best kept short

```
IEnumerable<string> filteredList =
    cities.Where((string s) =>
                        { string temp = s.ToLower();
                          return temp.StartsWith("L");
                        });
```

Invoking Lambda Expressions

- **Must first assign lambda to compatible delegate type**
- **Built-in Func and Action delegates available**
  - We rarely need to define custom delegates

```
Func<int, int> square = x => x * x;
Func<int, int, int> mult = (x, y) => x * y;
Action<int> print = x => Console.WriteLine(x);

print(square(mult(3, 5))); // displays 225;
```

Lambdas for LINQ

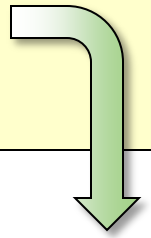
```
IEnumerable<string> filteredList =
    cities.Where(s => s.StartsWith("L"));
```

- **What if *`cities`* is not an in-memory collection?**
  - LINQ works with databases, for example
  - Does filtering occur inside the database, or inside our app domain?

Code as Data

- Lambda expressions as delegates become opaque code
- The alternative is Expression<TDelegate>

```
Expression<Func<int, int>> squareExpression = x => x * x;
Expression<Func<int, int, int>> multExpression = (x, y) => x * y;
Expression<Action<int>> printExpression = x => Console.WriteLine(x);

Console.WriteLine(squareExpression);
Console.WriteLine(multExpression);
Console.WriteLine(printExpression);
```

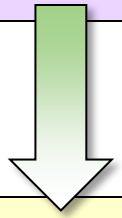


```
x=>(x*x)
(x,y)=>(x*y)
x=>WriteLine(x)
```

Expression Trees

- **C# treats Expression<TDelegate> as a special type**
  - Instead of generating MSIL, compiler generates an expression tree

```
Expression<Func<int, int>> squareExpression = x => x * x;
```



```
ParameterExpression x;
Expression<Func<int, int>> squareExpression =
Expression.Lambda<Func<int, int>>(
   Expression.Multiply(x = Expression.Parameter(typeof(int), "x"), x),
    new ParameterExpression[] { x });
```

Using Expressions

- **Compile an expression before invoking**
  - Generating MSIL at runtime

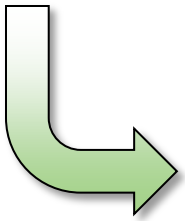
```
Expression<Func<int, int>> squareExpression = x => x * x;
Func<int, int> square = squareExpression.Compile();

int y = 3;
int ySquared = square(y);

Console.WriteLine(ySquared); // prints 9
```

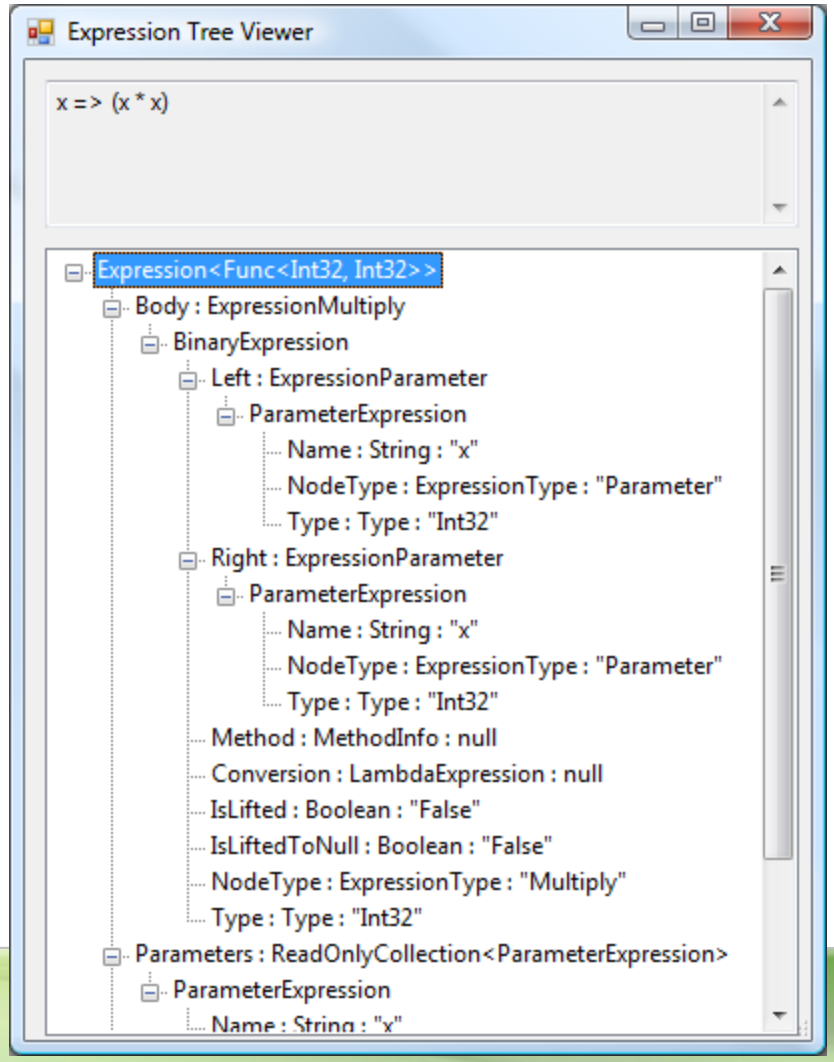
- **But the real power of an expression is through runtime analysis …**

Analyzing Trees

- **LINQ Providers can analyze expression**
  - LINQ to SQL will build SQL command after analysis

```
Expression<Func<int, int>>
  squareExpression = x => x * x;
```





In-memory LINQ Versus Remote LINQ

- IEnumerable<T> versus IQueryable<T>


```
namespace System.Linq {
    public static class Enumerable
    {
        public static IEnumerable<TSource> Where<TSource>(
                                this IEnumerable<TSource> source,
                                Func<TSource, bool> predicate) …
    }

    ...
}
```

delegate

```
namespace System.Linq {
    public static class Queryable
    {
        public static IQueryable<TSource> Where<TSource>(
                                this IQueryable<TSource> source,
                                Expression<Func<TSource, bool>> predicate)
            ...
    }

    ...
}
```

expression

Next Steps …

- **Extension methods give us standard operators**
  - As extension methods, standard operators can be redefined
- **Lambda expressions give us expressive logic**
  - Can use as delegates
  - Can use as expression trees
- **But - does this code look like a query?**

```
string[] cities = { "Boston", "Los Angeles",
                    "Seattle", "London", "Hyderabad" };

IEnumerable<string> filteredList =
    cities.Where(s => s.StartsWith("L"));
```

Query Expressions

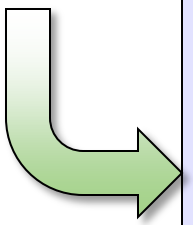
```
string[] cities = { "Boston", "Los Angeles",
                    "Seattle", "London", "Hyderabad" };

IEnumerable<string> filteredCities =
    from city in cities
    where city.StartsWith("L") && city.Length < 15
    orderby city
    select city;
```

- **Puts the "language integrated" into LINQ**
- **Begins with a *from* clause, ends with a *select* or *group***
  - Can use `from`, `let`, `where`, `orderby`, and `join`
- **Looks like a SQL query**
  - *from* logically comes first (also helps Intellisense)

Sweet and Sugary Syntax

- **Compiler transforms query expressions into a series of method calls with lambda expressions**

```
string[] cities = { "Boston", "Los Angeles",
                    "Seattle", "London", "Hyderabad" };

IEnumerable<string> filteredCities =
    from city in cities
    where city.StartsWith("L") && city.Length < 15
    orderby city
    select city;
```



```
IEnumerable<string> filteredCities =
    cities.Where(c => c.StartsWith("L") && c.Length < 15)
          .OrderBy(c => c)
          .Select(c => c);
```

Remaining Troubles

```
IEnumerable<string> filteredCities =
    from city in cities
    where city.StartsWith("L") && city.Length < 15
    orderby city
    select city;
```

- **Type names can clutter the query**
- **Projection is still difficult**
  - Query a collection of Employee and return an EmployeeSummary
  - Requires a new type (EmployeeSummary)
  - Requires a well-stocked EmployeeSummary constructor

Implicit Typing

- **C# 3.0 introduced the *var* keyword**
  - Unlike JavaScript – does not denote weak, dynamic, or loose typing
- **Compiler infers the type of the variable**

```
var name = "Scott";
var x = 3.0;
var y = 2;
var z = x * y;

// all lines print "True"
Console.WriteLine(name is string);
Console.WriteLine(x is double);
Console.WriteLine(y is int);
Console.WriteLine(z is double);
```

Restrictions For Implicit Typing

- **Must have a non-ambiguous initializer**
  - null is ambiguous
- **Variable is still strongly typed!**

```
// ERROR: implicitly typed local variables must be initialized
var i;

// ERROR: Implicitly-typed local variables cannot have multiple
declarators
var j, k = 0;

// ERROR: Cannot assign <null> to an implicitly-typed local variable
var n = null;

var number = "42";
// ERROR: Cannot implicitly convert type 'string' to 'int'
int x = number + 1;
```

Anonymous Types

- **Nameless classes created with an *object initializer***
- **Specify properties and their initial values**
  - Compiler creates class with read-only properties
  - Always derives from System.Object
- **Cannot use anonymous type as return value or parameter**
  - No type name!
  - Need to return or pass System.Object

```
var employee = new {
        Name = "Scott",
        Department = "Engineering"
    };
Console.WriteLine("{0}:{1}", employee.Name, employee.Department);
```

LINQ with var and Anonymous Types

```
var processList =
     from process in Process.GetProcesses()
     orderby process.Threads.Count descending,
             process.ProcessName ascending
     select new
     {
         process.ProcessName,
         ThreadCount = process.Threads.Count
     };

Console.WriteLine("Process List");
foreach (var process in processList)
{
    Console.WriteLine("{0,25} {1,4:D}",
        process.ProcessName,
        process.ThreadCount);
}
```

Initializers For Named Classes and Collections

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public Address HomeAddress { get; set; }
}

public class Address
{
    public string City { get; set; }
    public string Country { get; set; }
}
```

```
Employee employee = new Employee {
        ID = 1,
        Name = "Sami",
        HomeAddress = { City = "Sharpsburg", Country = "USA" }
    };
List<Employee> employees = new List<Employee>() {
    new Employee { ID=2, Name="...", HomeAddress= { City="...", Country="..." }},
    new Employee { ID=3, Name="...", HomeAddress= { City="...", Country="..." }},
    new Employee { ID=4, Name="...", HomeAddress= { City="...", Country="..." }}
};
```

Partial Methods

- **Extensibility mechanism for designer generated code**
  - LINQ to SQL designer uses partial methods
- **Implicitly private – no return values or out parameters**
- **Optimizations for unused methods**
  - Compiler removes method calls if no implementation is defined

```
public partial class Account
{
    public Account()
    {
        OnCreated();
    }

    partial void OnCreated();
}
```

```
public partial class Account
{
    partial void OnCreated()
    {
        Console.WriteLine(
            "Account created...");
    }
}
```

Summary

- **LINQ is the product of language and framework design**
  - Extension methods
  - Lambda expressions
  - Expression trees
  - Query expressions
- **Many of these LINQ oriented features useful for everyday code**
  - Functional programming
  - Reduced typed noise

References

- The Evolution of LINQ and Its Impact On The Design of C#
  http://msdn2.microsoft.com/en-us/magazine/cc163400.aspx
- Visual Studio 2008 Samples (Expression Tree Viewer)
  http://msdn2.microsoft.com/en-us/vcsharp/bb330936.aspx