Leveraging C# Extensibility Points

Part 1

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Compiling with Patterns

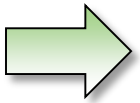
- **Syntactic sugar**
  - Language features defined in terms of other, existing features
  - Expansion of shorthand syntax into patterns
    - Sometimes using interfaces
    - Often *just* a pattern

- **Examples in C#**
  - "foreach statements" leverage "enumeration pattern"
    - GetEnumerator, MoveNext, Current, Dispose
  - "query expressions" leverage "query pattern"
    - Where, Select, SelectMany, GroupBy, OrderBy, ThenBy, Join, GroupJoin
  - "await expressions" leverage "awaiter pattern"
    - GetAwaiter, IsCompleted, OnCompleted
  - "list initialization expressions" leverage "collection pattern"
    - IEnumerable, Add

Foreach Statements

- **Common misconception**
  - "foreach" requires IEnumerable or IEnumerable<T>
  - Only depends on the enumeration pattern
- **Similarity to duck typing**
  - "If it walks and quacks like a duck…
    - … it's a duck"
  - If it has GetEnumerator, MoveNext, Current…
    - … it ought to be *enumerable*

```
foreach (T item in items)
{
  Foo(item);
}
```



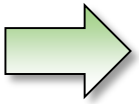
```
using (var e = items.GetEnumerator()) {
  // Closure scoping of item!
  while (e.MoveNext()) {
    T item = e.Current;
    Foo(item);
  }
}
```

Foreach Statements

- **Typing**
  - User specifies range loop variable type
  - Compiler discovers Current property type
  - Conversion are considered
    - Inheritance from C# 1.0 days prior to generics

```
var xs = new List<object>
  { 1, 2, 3 };

foreach (int x in xs) {
  …
}
```



```
using (var e = items.GetEnumerator()) {
  while (e.MoveNext()) {
    int item = (int)e.Current;
    Foo(item);
  }
}
```

- **Caveats**
  - Conversion can fail
  - IDisposable interface method call can cause boxing

Operator Overloading

- **Various operators can be overloaded**
  - Arithmetic, relational, logical, conversions, etc.
  - No “evil” overloading, e.g. comma or dot
  - Just methods in disguise, e.g. op_Addition

- **Short-circuiting logic**
  - && and || operators
  - ?: conditional “ternary” operator
  - Use truthiness unary operators “true” and “false”

```
x || y     ➔  Tx.op_True(x)  ? x : (x | y)
x && y     ➔  Tx.op_False(x) ? x : (x & y)
b ? x : y  ➔  Tx.op_True(x)  ? x : y
```

Operator Overloading

- **Building a small DSL**
  - Domain-specific language
  - Internal DSL piggybacks on host language syntax
  - Evaluation of the expression returns a data structure
    - Can analyze, interpret, compile, etc.

```
class Bool {
  public static Bool operator &(Bool a,
                                Bool b) {
    return new And(a, b);
  }

  public static bool false(Bool b) {
    return false;
  }
}
```

```
a && b
  ==
op_False(a) ? a : (a & b)
```

```
class And : Bool {
  public And(Bool a, Bool b) {
    Left = a; Right = b;
  }

  public Bool Left
    { get; private set; }
  public Bool Right
    { get; private set; }
}
```

Operator Overloading

- **Nullability for value types**
  - Introduced in C# 2.0
  - Ternary logic
    - If any operand is null, return null
      - Boolean for relational operators
    - Otherwise, evaluate operator
  - Lifting of operators
    - Existing operators augmented with nullable support
    - Operator overloads for nullable variants take precedence

```
struct Num
{
  public static Num operator +(Num a, Num b)
  { … }
}
```

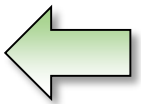
```
Num? a = …;
Num? b = …;
Num? c = a + b;
```

Lifted + operator

Operator Overloading

- **Nullability for value types**
  - Introduced in C# 2.0
  - Ternary logic
    - If any operand is null, return null
      - Boolean for relational operators
    - Otherwise, evaluate operator
  - Lifting of operators
    - Existing operators augmented with nullable support
    - Operator overloads for nullable variants take precedence

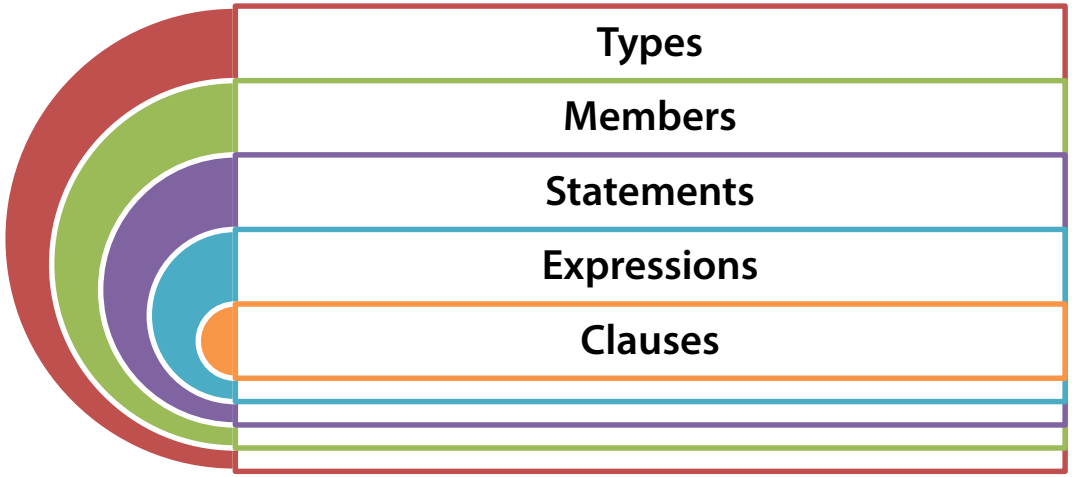
```
Num? a = …;
Num? b = …;
Num? c = !a.HasValue || !b.HasValue
       ? null
       : (a.Value + b.Value);
```



```
Num? a = …;
Num? b = …;
Num? c = a + b;
```

Query Expressions

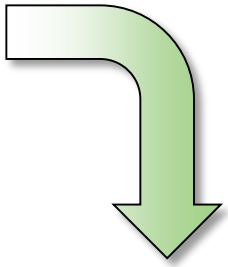
- **Query expressions (LINQ) consist of**
  - One or more *from* clauses
  - Joins using *join* clauses
  - Filters using *where* clauses
  - Sorting using *orderby* clauses
  - Projections using *select* (or *let*) clauses
  - Groupings using *group by* clauses
  - Termination with a *select* or *group by* clause
    - Continuations using *into*



Query Expressions

- **The query pattern**
  - Each clause translates in a (fluent) method call pattern
  - Query expressions = syntactic sugar

```
from p in products
where p.Price > 24.95m
orderby p.Price descending,
        p.Name
select new { p.Name, p.Price }
```



Can be assigned to an expression tree

Just like *any* other ordinary method call

```
products
.Where(p => p.Price > 24.95m)
.OrderByDescending(p => p.Price)
.ThenBy(p => p.Name)
.Select(p => new { p.Name, p.Price })
```

Query Expressions

- **LINQ to ***
  - Can implement the query pattern on any type
    - Language has no dependencies on particular BCL types
  - Fluent method pattern using
    - Instance methods, or
    - Extension methods
  - Built-in query provider implementations using IQueryable<T>
    - E.g. LINQ to SQL
- **Overload the pattern yourself**
  - Restrict query operators available for domain
  - Or go crazy ☺

```
static class RegExStringExtensions {
    public static MatchCollection Where(
        this string text,
        Func<object, RegEx> predicate) { … }
}
```

Query Expressions

- **How query providers work (typically)**
  - Analysis of expression trees at runtime
  - Only lambda expression can be captured
    - Queryable uses a clever trick to capture the entire query expression

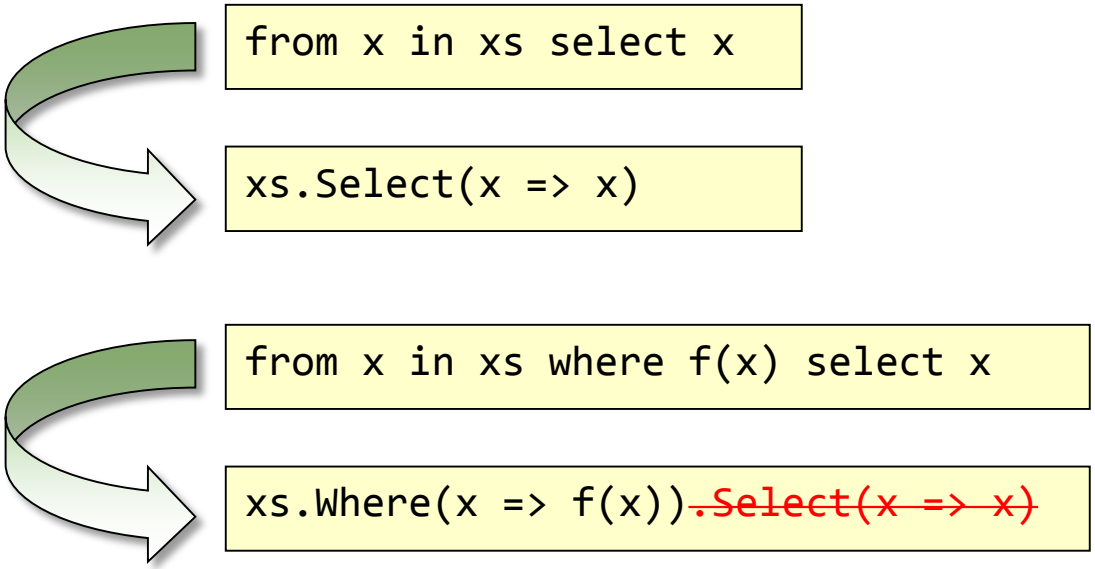
```
public static IQueryable<T> Where<T>(
    this IQueryable<T> source,
    Expression<Func<T, bool>> predicate)
{
    var whereOfT = (MethodInfo)MethodBase.GetCurrentMethod();
    var method = whereOfT.MakeGenericMethod(typeof(T));
    return source.Provider.CreateQuery<T>(
        Expression.Call(method, source.Expression, predicate)
    );
}
```

Assignment of lambda to expression tree

IQueryable<T> has an Expression

Query Expressions

- **Some language tidbits**
  - Final projection is erased, iff
    - It's the identity function "x => x"
    - It's preceded by a clause other than from



  - Guarantees hiding of the source type

Query Expressions

- **Transparent identifiers**
  - Keep track of "scope" in a query by flowing objects
    - Anonymous type instances
    - No user-visible identifier (*transparent*)

Where predicate has 1 parameter

```
from p in products
from o in GetOffers(p)
let q = p.Price
let d = o.Discount
where q * d > 10m
select p.Name
```

How is scope for {p, o, q, d} tracked?

```
products
.SelectMany(p => GetOffers(p), (p, o) => new { p, o })
.Select(t => new { t, q = t.p.Price })
.Select(t => new { t, d = t.t.o.Discount })
.Where(t => t.t.q * t.d > 10m)
.Select(t => t.t.t.p.Name)
```

Summary

- **Syntactic sugar**
  - Language features defined in terms of other features
- **The foreach keyword**
  - Enumeration pattern is more than just IEnumerable
  - GetEnumerator, MoveNext, Current, Dispose
- **Operator overloading**
  - Just methods in disguise
  - Useful trick to build DSLs
- **Query expressions**
  - Query pattern is more than just IEnumerable<T>, IQueryable<T>
  - Where, Select, OrderBy, ThenBy, GroupBy, SelectMany, Join, GroupJoin
  - Blends nicely with extension methods, expression trees