C# 3.0 Language Features

C# Programming

March 12, 2007
Previous C# Releases

• 1.0 – 2001
• 1.1 – 2003
• 2.0 – 2005
  – Generics
  – Anonymous methods
  – Iterators with yield
  – Static classes
  – Covariance and contravariance for delegate types
C# 3.0 “Orcas”

• The next release of Visual Studio is code-named Orcas
• Expected late 2007 or early 2008
• Includes new C# 3.0 features
• No runtime changes in C# 3.0
• So all 2.0 and 3.0 binaries will be compatible
• Primary new features target querying data and functional programming paradigms
C# 3.0 Features

• Implicitly-typed local variables
• Extension methods
• Lambda expressions
• Object initializers
• Anonymous types
• Implicitly-typed arrays
• Query expressions (LINQ)
• Expression trees
Language Integrated Query

• The query expression is a new syntactical construct (in C# and VB) designed to allow accessing relational, XML, and object data in the same way

• LINQ is the primary new feature in C# 3.0

• Many of the other new language features are used in query expressions, so we will begin by surveying these smaller additions
Local Variables

When a local variable’s type can be inferred from the initializer, it can be declared var

- `var i = 5;`
- `var s = "Hello";`
- `var d = 1.0;`
- `var numbers = new int[] {1, 2, 3};`
- `var orders = new Dictionary<int, Order>();`
Local Variables

The previous implicitly-typed declarations are equivalent to:

• int i = 5;
• string s = “Hello”;
• double d = 5.0;
• int[] numbers = new int[] {1, 2, 3};
• Dictionary<int, Order> orders = new Dictionary<int, Order>();
Local Variables

Restrictions on implicitly-typed variables:

• Declaration must include an initializer
• Compile-time type of initializer cannot be null
Local Variables

• Implicitly-typed local variables can also be used within using statements for resource acquisition...
  
  using (var stream = new StreamReader(@"C:\file.txt") { ... }

• ...and within foreach statements
  
  foreach (var n in numbers) { ... }
Object Initializers

- Allows initialization of objects with record-like syntax

```java
public class Point {
    int x, y;
    public int X
    { get { return x; } set { x = value; } }
    public int Y
    { get { return y; } set { y = value; } }
}
```
Object Initializers

- To create and initialize:
  ```csharp
  var p = new Point { X = 1, Y = 2 };
  ```
- This has the same effect as:
  ```csharp
  var p = new Point();
  p.X = 1;
  p.Y = 2;
  ```
- Order of members doesn’t matter
  ```csharp
  var p = new Point { Y = 2, X = 1 };
  ```
Object Initializers

• Consider a class definition with members that are reference types

```csharp
public class Rectangle {
    Point P1, P2;
}
```

• Can be created and instantiated with:

```csharp
var r = new Rectangle {
    P1 = new Point { X = 1, Y = 2 },
    P2 = new Point { X = 3, Y = 4 }
};
```
Object Initializers

• If P1 and P2 are instantiated in the Rectangle class:

```java
public class Rectangle {
    Point p1 = new Point(), p2 = new Point();
}
```

• Then the object initializer looks like:

```java
var r = new Rectangle {
    P1 = { X = 1, Y = 2 },
    P2 = { X = 3, Y = 4 }
};
```
Collection Initializers

• Objects of type ICollection<> can now also be initialized with elements

```csharp
List<int> digits = new List<int>();
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```
public class Contact {
    string Name;
    List<string> Numbers = new List<string>();
}

var contacts = new List<Contact> {
    new Contact {
        Name = "Foo",
        Numbers = {"123"} ,
    },
    new Contact {
        Name = "Bar",
        Numbers = {"234", "345"} 
    }
};
Anonymous Types

• An anonymous type can be built using an anonymous object initializer
• This type cannot be referenced in the program text
• This nameless class inherits from object
• The only operations allowed on an object of anonymous type are reads/writes of its members
• The anonymous type:

```csharp
new { p1 = e1, p2 = e2 }
```

• Gets compiled to:

```csharp
class __Anonymous1 {
    private T1 f1 = e1;
    private T2 f2 = e2;
    public T1 p1 = { get { return f1; } set { f1 = value; } }
    public T2 p2 = { get { return f2; } set { f2 = value; } }
}
```
Anonymous Types

• Within the same program, two anonymous object initializers with the same structure produce instances of the same type

```csharp
var o1 = new { X = 1, Y = 0 };
var o2 = new { X = 2, Y = 0};
o1.GetType() == o2.GetType()  // True
```

• But order of fields does matter

```csharp
var o3 = new { Y = 1, X = 1};
o1.GetType() == o3.GetType()  // False
```
Implicitly-typed Arrays

• Type is inferred from the elements in the array initializer
• There must be a unique (non-null) type to which each element is equal or implicitly convertible
• Otherwise the array creation fails at compile-time
Implicitly-typed Arrays

```javascript
var a = new[] { 1, 10, 100, 1000}; //int[

var b = new[] {1, 1.5, 2, 2.5 }; //double[

var c = new[] {“hello”, null}; //string[

var d = new[] {1, “one”, 2, “two”}; //ERROR
```
Implicitly-typed Arrays

• Anonymous object initializers + implicitly-typed arrays allow creation of anonymously typed data structures:

```csharp
var contacts = new[] {
    new {
        Name = "Foo",
        Numbers = new[] {"123"} },
    new {
        Name = "Bar",
        Numbers = new[] {"234", "345"} }
};
```
Extension Methods

• An extension is syntactic sugar that makes a static method defined in class A to look like an instance method of class B
• Abusing this feature can lead to obfuscated code
• But, it makes query expressions possible to write
Extension Methods

• To define an extension method, the `this` keyword is added to the first argument of the method
• For example, to add an extension method to the string class:

```csharp
public static class StringUtils {
    public static ChopN (this string s, int n) {
        if (s.Length < n) return s;
        else return s.Substring(0, n);
    }
}
Extension Methods

• If we now import the namespace containing the StringUtils class, the ChopN method will appear to be a method of the string class
  “12345”.ChopN(3); // “123”

• Extension methods are brought into scope with lower precedence than regular instance methods
Lambda Expressions

• Anonymous methods were added in C# 2.0
• They can be used where delegate values are expected

delegate void MyDel(string s);
...
MyDel myDel = new MyDel(DisplayName);
void DisplayName(string s) {
    Console.WriteLine(s);
}
Lambda Expressions

• If DisplayName is not needed anywhere else, an anonymous method can be defined instead

```csharp
MyDel myDel = delegate(string s) {
    Console.WriteLine(s)
};
```

• This syntax gets clunky
Lambda Expressions

• Lambda expressions are a concise, functional syntax for writing anonymous methods

```csharp
var myDel = string s => Console.Write(s);
```

• Parameter types can be inferred

```csharp
var myDel = s => Console.WriteLine(s);
```
Lambda Expressions

- Parameter lists can be explicitly or implicitly typed
- The body of a lambda can be either an expression or statement block
- Lambdas with expression bodies get converted to expression trees
- Lambdas with statement blocks get compiled into IL code
Lambda Expressions

- $x \Rightarrow x + 1$
- $x \Rightarrow \text{return } x + 1;$
- $(\text{int } x) \Rightarrow x + 1$
- $(\text{int } x) \Rightarrow \text{return } x + 1;$
- $(x, y) \Rightarrow x \ast y$
- $() \Rightarrow \text{Console.WriteLine(“Hello world”);}$
Lambda Expressions

• This syntax makes the use of polymorphic functions – like Filter and Map – more elegant

• Assuming we have generic implementations of these defined as extension methods:

```csharp
List<int> nums = new List<int>
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; 
var even = nums.Filter(n => n%2 == 0);
var sqrt = nums.Map(n => Math.sqrt(n));
```
public static IEnumerable<T> Filter<T>(this IEnumerable<T> list, Func<T, bool> test) {
    foreach (var item in list) {
        if (test(item))
            yield return item;
    }
}

public static IEnumerable<V> Map<T, V>(this IEnumerable<T> list, Func<T, V> convert) {
    foreach (var item in list) {
        yield return convert(item);
    }
}
Query Expressions

- SQL-like query syntax added to C# and VB
- This is purely syntactic sugar; all of it is rewritten to normal method invocations
- Designed to provide the same syntax for querying various types of data – relational, hierarchical, objects
Query Expressions

- There are no semantics for query expressions
- Instead, they get translated to methods that perform the query evaluation
- For example, some of the methods include Where<>, Select<>, Join<>
Query Expressions

```csharp
var odds =
    from n in nums
    where n % 2 == 1
    select n;

• Translates to:
    nums.Where(n => n % 2 == 1);
```
Query Expressions

```csharp
var oddsinvs =
    from n in nums
    where n % 2 == 1
    select new { n, inv=(double)1/n);
```

• Translates to:

```csharp
nums
    .Where(n => n % 2 == 1)
    .Select(n=>new {n, inv=(double)1/n});
```
Query Expressions

• The previous translations suggest that Where and Select are methods of List<T>
• They are actually extension methods defined in the System.Query namespace
• By using extension methods, query operations can be implemented for any type, whether the implementation is available or not
• If it is, you can provide implementations of the query methods as instance methods
Query Expressions

• The set of libraries that facilitate queries on XML is called XLinq
• These libraries duplicate much of the existing XML processing libraries
• In addition, they provide the methods required for querying
• Consider the previous example of contacts:

```xml
<contacts>
  <contact>
    <name>Foo</name>
    <numbers>...</numbers>
  </contact>
  <contact>
    <name>Bar</name>
    <numbers>...</numbers>
  </contact>
</contacts>
```
Query Expressions

• We can query this document with the same syntax:

```csharp
var xml = XElement.Load(path + @“contacts.xml”);
var names =
    from c in xml.Elements(“contact”) select c.Element(“name”).Value;
```
Query Expressions

• The DLinq namespace provides the classes for query relational data
• A class structure needs to be defined that matches the structure of the database
• For example, a table in the database gets declared as a class, and the columns get declared as members of the class
Query Expressions

[Table(Name="Customers")]
public class Customer
{
    [Column(Id=true)]
    public string CustomerID;
    [Column]
    public string City;
}
DataContext db = new DataContext(dbLocation);
Table<Customer> Customers = db.GetTable<Customer>();

var q =
    from c in Customers
    where c.City == "London"
    select c;
Query Expressions

• Queries are not executed immediately
• They are executed lazily when the results are needed
• This also allows composing queries
  
  ```csharp
  if (orderByLocation) {
    q = from c in q
    orderby c.Country, c.City
    select c;
  }
  else if (orderByName) {
    q = from c in q
    orderby c.ContactName
    select c;
  }
  ```