C#

Charles Calkins
OCI

.NET / C# History …

- 1997: Project Lightning / Project 42
  - July 2000: .NET project publicly announced at Professional Developers Conference
  - Jan 2002 released as .NET Framework 1.0
- Jan 1999: Project Cool
  - “C-like Object Oriented Language”
  - Named C# when announced at July 2000 PDC
  - Dec 2001: C# 1.0 standardized as ECMA-334
  - Released with .NET Framework 1.0

… .NET / C# History

C#	Date	.NET Framework	Visual Studio Version
1.0	Jan 2002	1.0	Visual Studio .NET 2002
1.2	Apr 2003	1.1	Visual Studio .NET 2003
2.0	Nov 2005	2.0	Visual Studio 2005
3.0	Nov 2007	3.5	Visual Studio 2008
4.0	Apr 2010	4.0	Visual Studio 2010

Mono

- Cross-platform .NET implementation
  - Linux, Windows, Mac OS X, BSD, Solaris, Wii, PlayStation 3, iPhone, Android (in beta)
  - x86, x86-64, IA64, PowerPC, SPARC(32), ARM, Alpha, s390, s390x
- Current public release v2.6 (Dec 2009)
  - C# 3.0, most of .NET 3.5 (even ASP.NET)
- Trunk
  - C# 4.0, some .NET 4.0

Terms …

- CLI – Common Language Infrastructure
  - Describes the executable code and runtime environment of the .NET Framework
  - Specification: ECMA-335, ISO/IEC 23271
- CTS – Common Type System
  - Set of types and operations shared between CTS-compliant programming languages

… Terms

- CIL – Common Intermediate Language
  - a.k.a. Microsoft Intermediate Language (MSIL) in old terminology
  - Platform-independent instruction set that executes in an environment supporting the CLI
    - CIL instruction set defined in ECMA-335, Partition III
- CLR – Common Language Runtime
  - a.k.a. Virtual Execution System (VES)
  - Virtual machine that executes CIL

C# 1.0 – Java-like

- Types
  - Value
    - int, double, bool, enum, struct, …
    - decimal (~28 sig digits, base 10 internal representation)
  - Reference
    - class, object, string, array, interface, delegate
- Single inheritance, implement multiple interfaces
- Garbage collection
- Reflection
- Exceptions
  - try / catch / finally

C# 1.0 – Delegate …

- An OO type-safe function pointer, in a way

```
delegate int D(int x);

int f1(int a) {
   System.Console.WriteLine("f1: {0}", a);
   return a;
}
int f2(int b) {
   b*=3;
   System.Console.WriteLine("f2: {0}", b);
   return b;
}
```

… C# 1.0 - Delegate

```
D d = f1; // instantiate, add f1 to invocation list
d += f2;  // add f2 to invocation list
d += f1;  // add f1 again
d += f2;  // add f2 again
Console.WriteLine("d(3): " + d(3)); // invoke
```

→ output →

```
f1: 3
f2: 9
f1: 3
f2: 9
d(3): 9
```

Return value of call is return value of last in invocation list

C# 2.0 – Anonymous Delegates

```
delegate int D();

D F() {
   int x = 0;
   D result = delegate { return ++x; };  // closure
   return result;
}
D d = F();
Console.WriteLine(d() + " " + d() + " " + d());
```

→ output → `1 2 3`

C# 2.0 – Generics

```
public class Stack<T>
{
   T[] items;
   public void Push(T item) {...}
   public T Pop() {...}
}

Stack<SomeType> stack = new Stack<SomeType>();
```

C# 2.0 – Generics with Constraints

```
public class EntityTable<K,E>
   where K: IComparable<K>, IPersistable
   where E: Entity, new()
{
   public void Add(K key, E entity)
   {
        ...
        if (key.CompareTo(x) < 0) {...}
        ...
   }
}
```

`new()` → assert that E has a public, parameterless constructor so can call

```
   new E()
```

C# 2.0 – Generic Methods

```
void PushMany<T>(Stack<T> stack, params T[] values)
{
   foreach (T value in values) stack.Push(value);
}

Stack<int> stack = new Stack<int>();
PushMany<int>(stack, 1, 2, 3, 4);
```

or, knows T by “type inferencing”:

```
PushMany(stack, 1, 2, 3, 4);
```

`params` → parameter array (C# 1.0)

C# 2.0 - Iterators

- Implement IEnumerable or IEnumerable<T> (GetEnumerator())

```
class C : IEnumerable {
   int[] a = { 1, 2, 3, 4 };
   public IEnumerator GetEnumerator() {
      for (int i = a.Length - 1; i >= 0; i--)
         yield return a[i];
   }
}

foreach (int i in new C()) Console.Write(i + " ");
```

→ output → `4 3 2 1`

C# 2.0 – Nullable Types

- Like a value type, plus a boolean indicating if it was null

```
int? a = null;  // nullable type
int? b = a;
int x = b ?? 3;  // null coalescing operator
b = 7;
int y = b ?? 3;
Console.WriteLine("{0},{1},{2},{3}", a, b, x, y);
```

→ output → ,7,3,7

C# 3.0 – Extension Methods …

- Old way: declare separate method

```
public static string ToTitleCase(string str) {
    if (str == null) return null;
    else return
        System.Globalization.CultureInfo.
        CurrentUICulture.TextInfo.ToTitleCase(str);
}

string s = "this is a string";
Console.WriteLine(ToTitleCase(s));
```

→ output → `This Is A String`

… C# 3.0 – Extension Methods …

- New way: extension method

```
// not-nested static class, "this" in argument list
public static class MyExtensions {
   public static string ToTitleCase(this string str) {
      if (str == null) return null;
      else return
         System.Globalization.CultureInfo.
         CurrentUICulture.TextInfo.ToTitleCase(str);
   }
}
string s = "this is a string";
Console.WriteLine(s.ToTitleCase());
```

… C# 3.0 – Extension Methods

- Works on interfaces, too – not just classes

```
public static class MyExtensions {
   public static IEnumerable<int>
      WhereEven(this IEnumerable<int> values) {
   foreach (int i in values)
      if (i % 2 == 0)
         yield return i;
   }
}
int[] a = { 1, 2, 3, 4 }; // arrays implement IEnumerable<T>
foreach (int i in a.WhereEven()) Console.Write(i+" ");
```

→ output → `2 4`

C# 3.0 – Lambda Methods …

- To make Where() generic, pass delegate

```
public static class MyExtensions {
   public delegate bool Criteria<T>(T value);

   public static IEnumerable<int>
      Where<T>(this IEnumerable<int> values,
         Criteria<T> criteria) {
   foreach (T item in values)
      if (criteria(item))
         yield return item;
   }
}
```

… C# 3.0 – Lambda Methods

- Old way: anonymous delegate

```
string[] names = new string[] { "Bill", "Jane",
   "Bob", "Frank" };
IEnumerable<string> Bs = names.Where<string>(
   delegate(string s) { return s.StartsWith("B"); }
   );
```

- New way: lambda method

```
IEnumerable<string> Bs = names.Where<string>(
   s => s.StartsWith("B")  // (a,b,…) => {…}
   );
```

C# 3.0 – Automatic Properties

- Old way: backing field

```
string name_;
public string Name {
   get { return name_; }
   set { name_ = value; }
}
```

- New way: automatic property

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

C# 3.0 – Object Initializer

- Old way: set properties

```
class Person {
   public string Name { get; set; }
   public int Age { get; set; }
}
Person p = new Person();
p.Name = "Bob";
p.Age = 40;
```

- New way: object initializer

```
Person p = new Person() { Name = "Bob", Age = 40 };
```

C# 3.0 – Type Inference

- Use `var` instead of explicit type

```
// now takes the type DateTime
var now = new DateTime(2001, 1, 1);

// legal - DayOfYear is a property of DateTime
int dayOfYear = now.DayOfYear;

// compiler error - Substring() is not a method of
   DateTime
string test = now.Substring(1, 3);
```

C# 3.0 – Anonymous Types

- Combine object initializers with type inference – compiler generates internal class definition (IntelliSense works, too)

```
var x = new { TheDate = new DateTime(2010, 1, 1),
   Name = "Bob" };
Console.WriteLine("{0}, {1}, {2}", x.TheDate, x.Name,
   x.GetType().Name);  // type name by reflection
```

→ output →

```
1/1/2010 12:00:00 AM, Bob, <>f__AnonymousType0`2
```

Putting It Together …

- First, define a generalized projection (mapping) operator as an extension method

```
public static class MyExtensions {
    public static IEnumerable<TDest>
        Select<T, TDest>(this IEnumerable<T> values,
        Func<T, TDest> transformation) {
    foreach (T item in values)
        yield return transformation(item);
    }
}
```

`Func<T1, T2>` → C# 3.0 built-in two parameter delegate

```
public delegate TResult Func<in T, out TResult>(T arg);
```

… Putting It Together …

- Then, query a data structure via a chain of extension methods of IEnumerable<T>, map result to anonymous type

```
string[] nameData = new string[] { "Steve", "Jimmy",
   "Celine", "Arno" };

// filter out Jimmy, map to anonymous type
var people = nameData.Where(str => str != "Jimmy")
   .Select(str => new {
      Name = str,
      LettersInName = str.Length,
      HasLongName = (str.Length > 5)
   });
```

… Putting It Together

```
foreach (var person in people)
   Console.WriteLine(
      "{0} has {1} letters in their name. {2}",
         person.Name, person.LettersInName,
         person.HasLongName ? "That's long!" : "");
```

→ output →

```
Steve has 5 letters in their name.
Celine has 6 letters in their name. That’s long!
Arno has 4 letters in their name.
```

C# 3.0 Language Integrated Query (LINQ)

- C# 3.0 has syntax to do the same thing – a "query expression"

```
var people = from str in nameData
             where str != "Jimmy"
             select new {
                Name = str,
                LettersInName = str.Length,
                HasLongName = (str.Length > 5)
             };
```

C# 3.0 LINQ to Objects

```
// find big processes, sort by name
var procs = from proc in
   System.Diagnostics.Process.GetProcesses()
   where proc.WorkingSet64 > 1024 * 1024
   orderby proc.ProcessName
   select new { ID = proc.Id, proc.ProcessName,
      WorkingSet = proc.WorkingSet64 };

foreach (var p in procs)
   Console.WriteLine("{0} {1} {2}", p.ID,
   p.ProcessName, p.WorkingSet);
```

C# 3.0 LINQ

- LINQ to Objects
  - LINQ with any collection supporting IEnumerable or IEnumerable<T>
- LINQ to XML
  - LINQ with an XML file / in-memory structure
- LINQ to SQL
  - LINQ with SQL Server
- More
  - Custom LINQ providers, such as SQLite

C# 4.0 Parallel LINQ

```
// http://download.geonames.org/export/dump/allCountries.zip
var lines = System.IO.File.ReadLines(@"C:\AllCountries.txt");
var q = from line in lines.AsParallel() // Is faster: 2:39 vs. 3:14
   let fields = line.Split(new char[] { '\t' })
   let elevation =
       string.IsNullOrEmpty(fields[elevationColumn]) ?
       0 : int.Parse(fields[elevationColumn])
   where elevation > 8000 // elevation in m
   orderby elevation descending
   select new {
       name = fields[nameColumn] ?? "",  // null test
       elevation = elevation,
       country = fields[countryColumn]
   };
```

C# 4.0 Default and Named Parameters

```
void M(int x = 9, string s = "A", DateTime dt =
  default(DateTime), Guid guid = new Guid()) {
  Console.WriteLine("x={0}, s={1}, dt={2},
      guid={3}", x, s, dt, guid);
}
// unnamed (must be first) and named parameters
M(3, dt: DateTime.Now);
```

→ output →

```
x=3, s=A, dt=8/31/2010 10:17:48 AM, guid=00000000-
  0000-0000-0000-000000000000
```

Default parameters must be compile-time constants (not Guid.NewGuid())

C# 4.0 Dynamic Type …

```
// want to parse this
string content = "FirstName,LastName,State\n" +
   "Troy,Magennis,TX\n" + "Janet,Doherty,WA";

public class CsvParser : IEnumerable {
   List<string> headers_;
   string[] lines_;
   public CsvParser(string csvContent) {
      lines_ = csvContent.Split('\n');
      // split header row into field names
      if (lines_.Length > 0)
         headers_ = lines_[0].Split(',').ToList();
   }
```

… C# 4.0 Dynamic Type …

```
    public IEnumerator GetEnumerator() {
    // return data lines (skip header)
    bool header = true;
    foreach (var line in lines_)
       if (header)
          header = false;
       else
          yield return new CsvLine(line, headers_);
    }
}
```

… C# 4.0 Dynamic Type …

```
public class CsvLine : System.Dynamic.DynamicObject {
    string[] lineContent_;
    List<string> headers_;

    public CsvLine(string line, List<string> headers)
    {
        lineContent_ = line.Split(',');
        headers_ = headers;
    }
```

… C# 4.0 Dynamic Type …

```
    public override bool TryGetMember(
          GetMemberBinder binder,  // method invoked
          out object result) {
       result = null;
       // find the header index, get column value
       int index = headers_.IndexOf(binder.Name);
       if (index >= 0 && index < lineContent_.Length) {
          result = lineContent_[index];
          return true;
       }
       return false;
    }
}
```

→ Override **`TryInvokeMember()`** if method invoked has arguments

… C# 4.0 Dynamic Type

```
string content = "FirstName,LastName,State\n" +
                 "Troy,Magennis,TX\n" +
                 "Janet,Doherty,WA";
var q = from dynamic c in new CsvParser(content)
        where c.State == "WA" // “State” from CSV
        select c;
foreach (var c in q) {
   Console.WriteLine("{0}, {1} ({2})", c.LastName,
      c.FirstName, c.State);
}
```

→ output →

```
Doherty, Janet (WA)
```

Careful – **`dynamic`** generates run-time, not compile-time, errors

References …

- History
  - http://jameskovacs.com/2007/09/07/cnet-history-lesson/
  - http://www.theregister.co.uk/2000/09/12/official_microsofts_csharp_is_cool/
  - http://www.code-magazine.com/Article.aspx?quickid=0501091
- Specifications
  - C#: http://www.ecma-international.org/publications/standards/Ecma-334.htm
  - CLI: http://www.ecma-international.org/publications/standards/Ecma-335.htm
  - 2.0 features: http://download.microsoft.com/download/9/8/f/98fdf0c7-2bbd-40d3-9fd1-5a4159fa8044/csharp%202.0%20specification_sept_2005.doc

… References

- Mono
  - http://www.mono-project.com/
- System.Data.SQLite (ADO.NET provider for SQLite)
  - http://sqlite.phxsoftware.com/
- Books
  - Richter. *CLR via C#, 3rd ed*, Microsoft Press, 2010
  - Sanderson. *Pro ASP.NET MVC 2 Framework, 2nd ed*, Apress, 2010
  - Magennis. *LINQ to Objects Using C# 4.0*, Addison-Wesley, 2010
  - Albahari and Albahari. *C# 4.0 in a Nutshell, 4th ed*, O'Reilly Media, 2010