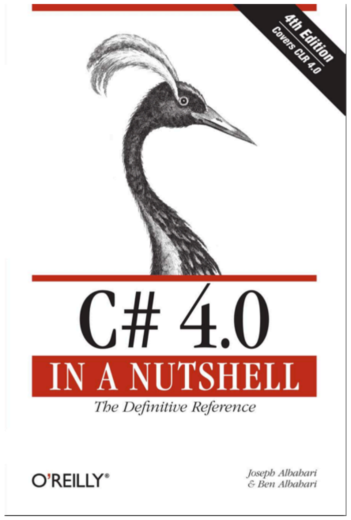
C# and .Net

Alexandre Bergel
abergel@dcc.uchile.cl
07/06/2011

Source



C# 4.0 in a Nutshell
Joseph Albahari & Ben Albahari
O'Reilly

Goal of this lecture

Introduction to C#, .Net and CLR

Emphasis on difference with Java

Understand the syntax and concept of C#

C# and the .Net Framework

- C# is
  - a general purpose language
  - type safe
  - an object-oriented programming language
- C# is “platform neutral, but work well with .Net”
- Mono (http://www.mono-project.com/) is an open-source and alternative to .Net
  - `mcs` to compile and `mono` to run
- Mono works on many platforms

C#: Simple example

```
using System;

class Test {

    static void Main() {

       int x = 12 * 30;

       Console.WriteLine(x);

    }

}
```

Namespace

```
using System;

namespace CC3002 {

  class Test { ... }

  class Test2 { ... }

  namespace NestedNamespace { ... }

}
```

Assembly

The C# compiler compiles source code (.cs file) into an *assembly*

An assembly can be a *library* (.dll) or an *application* (.exe)

Arithmetic overflow check operator

The checked operator tells the runtime to generate an `OverflowException` rather than failing silently

```
int a = 1000000;

int b = 1000000;

int c = check (a * b);

checked { ... c = a * b; ... }
```

Useful when precision matters

Value passing

By default arguments in Java and C# are *passed by value*

A copy of the value is created when passed to the method

Passing a reference-type argument by value *copies* the *reference*, but *not the object*

This is by far the most common case in today’s languages

The ref modifier

To *pass by reference*, C# provides the ref parameter modifier

```
class T {
  static void Foo (ref int p) { p++ ;}
  static void Main() {
    int x = 0;
    Foo(x);
    Console.writeLine(x); // Prints 1
  }
}
```

The ref modifier

```
class T {
  static void Swap(ref object o1, ref object o2){
    object t = o1;
    o1 = o2;
    o2 = t;
  }

  static void Main() {
    string s1 = “hello”;
    string s2 = “world”;
    Swap(s1, s2);
    Console.writeLine(s1); // “world”
    Console.writeLine(s2); // “hello”
  }
}
```

Value type

Value type

- built-in value type (primitive types)
- Custom value with struct

```
public struct Point { public int X, Y; }

Point p1 = new Point();

p1.X = 5;

Point p2 = p1; // Assignment cause copy
```

Optional parameters

Methods, constructors and indexers can declare *optional parameters*

```
void Foo (int x = 23) {Console.WriteLine(x); }

Foo();      // 23
```

Named arguments

Rather than identifying an argument by *position*, you can identity an argument by *name*

```
void Foo (int x, int y) { ... }

Foo (y:2, x:1); // eqv to Foo (1, 2)
```

Useful when calling COM APIs

Delegate

A delegate dynamically *wires up* a method *callers* to its *target* method

Similar to a *function pointer* in C or C++

Allow the programmer to encapsulate a *reference to a method*

Delegate type defines a *protocol* to which the caller and target will conform

Delegate instance is an object that *refers* to one (or more) *target methods*

Delegate

```
delegate int Transformer (int x);

class Test {
  static void Main() {
    // Create delegate instance
    Transformer t = Square;
    // Transformer t = new Transformer(Square);

    int result = t(3); // Invoke delegate
    Console.WriteLine(result); // 9
  }
  static int Square (int x) {return x*x;}
}
```

Multicast Delegates

All delegate instances have *multicast capability*.

```
SomeDelegate d = SomeMethod1;

d += SomeMethod2;
```

Invoking d will now *call both* `SomeMethod1` and `SomeMethod2`

Delegates are invoked in the *order* they are added

The **-** and **-=** operators remove a delegate

Multicast Delegates

If a multicast delegate has a *nonvoid return type*, the caller receives the return value from *the last* method to be invoked

The preceding methods are still called, but their return value is discarded

```
using System;

delegate int Del (int i);

class T {
  static void Main() {
    Del   d = D1;
    d += D2;
    d += D3;
    Console.WriteLine("Main " + d(5));
  }

  static int D1 (int i)
    { Console.WriteLine("D1 " + i); return 1; }
  static int D2 (int i)
    { Console.WriteLine("D2 " + i); return 2; }
  static int D3 (int i)
    { Console.WriteLine("D3 " + i); return 3; }
}
```

```
D1 5
D2 5
D3 5
Main 3
```

Delegates versus Interface

“A problem that can be *solved* with a *delegate* can also be solved with an *interface*”

A delegate design may be a *better choice than an interface* design if one or more of these conditions are true:

- the interface defines only a single method
- multicast capability is needed
- the subscriber needs to implement the interface multiple times

Events

When using delegates, two emergent roles appear

- *broadcaster*: a type that contains a delegate field. It decides when to broadcast, by invoking the delegate
- *subscribers*: the method target. A subscriber decides when to start and stop listening, by calling += and -=

Events are a language feature that formalizes this pattern

An event is a *construct* that exposes just the subset of delegate features required for the boardcaster

The goal of events is to *prevent subscribers from interfering with each other*

Events

```
public class Metronome {
  public event TickHandler Tick;
  ...
}
```

Code within `Metronome` has full access to `Tick` and can treat it as a delegate

Metronome Example

```
using System;

namespace eventExample {
  public class Metronome {
    public event TickHandler Tick;
    public delegate void TickHandler (Metronome m);
    public void Start () {
      while (true) {
        System.Threading.Thread.Sleep (2000);
        if (this.Tick!= null) {
          this.Tick (this);
        }
      }
    }
  }
  ...
```

Metronome Example

```
 ...
 public class Listener {
   public void Subscribe (Metronome m) {
     m.Tick += new Metronome.TickHandler(HeardIt);
   }
   private void HeardIt(Metronome m) {
     Console.WriteLine("Heard it!");
   }
 }
...
```

Metronome Example

```
  ...
  class Run {
    static void Main() {
      Metronome m = new Metronome();
      Listener l = new Listener();
      l.Subscribe(m);
      m.Start();
    }
  }
}
```

Lambda expression

```
using System;

delegate int Transformer (int i);
public class Lambda {
  static void Main() {
    Transformer sqr = x => x * x;
    Console.WriteLine(sqr(3));
  }
}
```

Lambda expression

A lambda expression has the following form

```
    (parameters) => expression-or-statement-
  block
    x => {return x * x; };
```

Used with two pre-defined delegates, Func and Action

```
    Func<int,int> sqr = x => x * x;
    Func<string,string,int> totalLength =
                (s1, s2) => s1.Length + s2.Length
    int total = totalLength (“hello”, “world”);
```

The compiler can usually infer the type of lambda

The using statement

Many classes encapsulate unmanaged resources, such as file handles, graphic handles, or database connections

Consider:

```
StreamReader reader;
try { reader = File.OpenText (“file.txt”); ... }
finally { if (reader != null) ((IDisposable)reader).Dispose();}
```

Shortened into:

```
using (StreamReader reader = File.OpenText (“file.txt”)) {...}
```

Exception

All exceptions in C# are *runtime exceptions*

There is no equivalent to Java’s compile-time checked exceptions

Atomicity pattern

```
using System;

class Transaction {
  static void Main() {
    Accumulator a = new Accumulator();
    try {
      a.Add (4, 5);  // Result is 9

      // Will cause OverflowException
      a.Add (1, int.MaxValue);
    }
    catch (OverflowException) {
       Console.WriteLine (a.Total); // 9
    }
  }
}
```

Atomicity pattern

```
class Accumulator {
  public int Total {get; private set; }

  public void Add (params int[] ints) {
    bool success = false;
    int totalSnapshot = Total;
    try {
      foreach (int i in ints) {
        checked {Total += i; }
      }
      success = true;
    }
    finally {
      if (! success) Total = totalSnapshot;
    }
  }
}
```

Accessors

The class Accumulator has a field Total

```
class Accumulator {

 public int Total {get; private set; }
 ...
}
```

This field can be accessed from other objects

```
    Console.WriteLine(new Accumulator().Total);
```

But it cannot be set. The following raises a compile time error

```
    new Accumulator().Total = 3
```

Variable number of arguments

C#, as well as Java, accepts a variable number of arguments when sending message

Variables are declared in a particular way

In C#, use the `params` keyword

```
public void Add (params int[] ints) { ...}

a.Add(1 , 2 , 3 , 4);
```

Variable number of arguments (Java)

```
public class T {

    public static int add(int... ints) {
        int result = 0;
        for (int i : ints)
            result += i;
        return result;
    }
    public static void main(String[] argv) {
        // prints 21
        System.out.println(add(1,2,3,4,5,6) + add());
    }
}
```

Extension Methods

Object-orientation provides many mechanism for supporting software evolution and extension

- e.g., late binding, polymorphic, class inheritance, reflection

C# offers the possibility to add methods to already existing class

- Already present in many other languages: CLOS, Objective-C, Pharo, Ruby, AspectJ, ...

Extension Methods

```
using System;
// a static class cannot be instantiated
public static class StringHelper
{
  // “string” is the equivalent of “String” in Java
  public static bool IsCapitalized (this string s) {
    if (string.IsNullOrEmpty(s))
      return false;
    else
      return char.IsUpper (s[0]);
  }

  static void Main() {
    Console.WriteLine("Hello".IsCapitalized());
  }
}
```

Extension Methods

An extension method can be used *as if it is defined on the* corresponding *class*

It has to be *under* the *current scope*

Conflicts are not allowed

An instance method will always *take precedence* over an extension method

Can apply to interfaces

Dynamic binding

Way to *check types* at *execution* rather than at compilation

Useful when at compile time *you* know that a certain function, member or operation exists, but the *compiler* does not

```
dynamic d = "hello";

Console.WriteLine (d.ToUpper()); // HELLO

// Compiles OK but gives runtime error

Console.WriteLine (d.Foo());
```

Dynamic binding

Calling an object *dynamically* is useful in scenarios that would otherwise require *complicated reflection* code

Dynamic binding is also *useful* when *interoperating* with *dynamic languages* (e.g., IronPython or IronRuby) and *COM components*

TryInvokeMember

```
using System;
using System.Dynamic;

public class Test {
  static void Main () {
    dynamic d = new Duck ();
    d.Quack();
    d.Waddle();
  }
}
```

TryInvokeMember

```
public class Duck : DynamicObject {
  public override bool TryInvokeMember
             ( InvokeMemberBinder binder,
               object[] args,
               out object result) {
    Console.WriteLine
        (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

The Duck class does not have a Quack method. Instead, it uses *custom bindings* to *intercept* and interpret all method calls

Implicit conversions

```
int i = 7;
dynamic d = i;
int j = d;

int i = 7;
dynamic d = i;
short j = d;  // throws RuntimeBinderException
```

Unsafe code and pointers

- C# supports *direct memory manipulation* via *pointers*
    - within blocks of code marked unsafe
    - compiled with the /unsafe compiler option
- Pointer types are primarily useful for interoperability with C APIs
- Useful for accessing memory outside the managed heap or for performance-critical hotspots

Unsafe code and pointers

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Framework Overview

- Almost all the capabilities of the .Net Framework are exposed via types
- Contains
  - all the essential types
  - user interface technologies
  - backend technologies
  - distributed system technologies

Language Integrated Query

LINQ is a set of language and framework feature for writing structured type-safe queries over collections

Enable to query any collections that implements IEnumerable<T> (array, list, XML DOM, SQL Server)

Linq

```
using System;
using System.Collections.Generic;
using System.Linq;

class LinqDemo {
  static void Main () {
    string[] names = { "Tom", "Dick", "Harry" };
    IEnumerable<string> filteredNames =
           names.Where(n => n.Length >= 4);

    foreach (string name in filteredNames)
      Console.WriteLine(name);
  }
} // print “Dick” and “Harry”
```

Query expression syntax

C# embeds a syntax to easily query collections

```
IEnumerable<string> filteredNames = names
            .Where   (n => n.Contains (“a”))
            .OrderBy (n => n.Length)
            .Select  (n => n.ToUpper());
```

Deferred Execution

query operators are not *executed* when constructed, but when *enumerated*

```
var numbers = new List<int> ();
numbers.Add (1);

var query = numbers.Select (n => n * 10);
numbers.Add (2);
foreach (var n in query)
  Console.Write (n + “ | “);

// 10 | 20 |
```

What you should know!

What is an assembly?

What is the difference between a library and an application?

What is the difference between passing by value and passing by reference?

What is an optional parameter?

What is a delegate?

What is a lambda?

What is a method extension?

Can you answer these questions?

When passing by reference is useful?

Which design pattern delegates and events help implement?

How would you implement lambdas in Java?

How method extension help to get a better distribution of responsibilities?

License