C#/.Net Project Cluster

Other new C# 2.0 features

and

Simple WinForms user interfaces

Peter Sestoft
KVL and IT University of Copenhagen

C#/.Net project cluster

Wednesday 4 May 2005

- Iterators: the `yield` statement.
- Partial types
- Anonymous methods: `delegate` expressions.
- SQL-style nullable value types: `int?`, `bool?`, and so on.
- Graphical user interfaces (GUIs) with WinForms.

New in C# 2.0: Anonymous methods: `delegate` expressions

Advanced API's often have methods that take delegates as arguments, for instance:

```
class IntList {
  public IntList Filter(IntPredicate p);
  ...
}
delegate bool IntPredicate(int x);
```

The `Filter` method may return a list containing only those elements `x` for which `p` is true.

We can define a method `Even` that is true for even integers, make a delegate, and apply `Filter` to it:

```
static bool Even(int x) { return x%2 == 0; }
...
list.Filter(Even);
```

C# 2.0 allows us to define `Filter`'s delegate argument inline, as an anonymous method:

```
list.Filter(delegate(int x) { return x%2 == 0; });
```

An anonymous method `delegate(...) {...}` is an expression that evaluates to a delegate.

Like anonymous functions `(fn x => ...)` in Standard ML or `lambda` in Scheme or λ in the λ-calculus.

In Java one would use methods in anonymous inner classes, but they are (even) more verbose.

Using an anonymous method to specify sorting order

A quicksort method `Qsort` may take a delegate as argument to specify the sorting order:

```
public delegate int DComparer<T>(T v1, T v2);

private static void Qsort<T>(T[] arr, DComparer<T> cmp, int a, int b) {
  ...
  while (cmp(arr[i], x) < 0) i++;
  while (cmp(x, arr[j]) < 0) j--;
  ...
}
```

The `Qsort` method may be called with a delegate created from a method:

```
static int StringReverseCompare(String s1, String s2) {
  return String.Compare(s2, s1);
}
...
Qsort(sa, StringReverseCompare, 0, sa.Length-1);
```

Or it may be called with a delegate created by an anonymous method expression:

```
Qsort(sa,
      delegate(String s1, String s2) { return String.Compare(s2, s1); },
      0,
      sa.Length-1);
```

This is often convenient, but abuse leads to incomprehensibility.

An anonymous method can use the enclosing method’s variables

Assume the hypothetical `IntList` class has a method `Apply` that applies a delegate to all elements:

```
class IntList {
  public void Apply(IntApplier p);
  ...
}
delegate void IntApplier(int x);
```

Then we can write a method to compute the sum of all list elements, using an anonymous method:

```
static int Sum(IntList list) {
  int res = 0;
  list.Apply(delegate(int x) { res += x; });
  return res;
}
```

Note that the anonymous method uses the `Sum` method’s local variable `res`.

Powerful, but . . .

For this to be possible, the C# compiler must turn the `Sum` method into a member method of a new (hidden) class.

Now multiple threads can access a local variable of a method; otherwise unheard of. Could cause surprises.

Fancy uses of anonymous methods

A `Fun<A,R>` is a one-argument delegate, a `Fun<A1,A2,R>` is a two-argument delegate:

```
public delegate R Fun<A,R>(A x);
public delegate R Fun<A1,A2,R>(A1 x1, A2 x2);
```

Method `MakeAdder(x)` returns a delegate that returns the sum of `x` and its argument `y`:

```
public Fun<int,int> MakeAdder(int x) {
  return delegate(int y) { return x+y; }
}
```

We can use it like this:

```
Fun<int,int> addSeven = MakeAdder(7;
int z1 = addSeven(10), z2 = addSeven(35);
```

Just to scare you: Method `Curry` turns a two-argument delegate `f` into a delegate that returns a delegate:

```
public static Fun<A,Fun<B,C>> Curry<A,B,C>(Fun<A,B,C> f) {
  return delegate(A x) {
    return delegate(B y) {
      return f(x, y);
    };
  };
}
```

New in C# 2.0: Iterators and the `yield` statement

A C# enumerator is traditionally written as a (nested) class, just like a Java iterator.

This is cumbersome, and easy to get wrong.

Example: Enumerate the integers $m, m + 1, \ldots, n$:

```
class MyTest {
  public static void Main(String[] args) {
    foreach (int i in FromTo(13, 17))
      Console.WriteLine(i);
  }

  public static IEnumerable<int> FromTo(int m, int n) {
    return new FromToEnumerable(m, n);
  }

  private class FromToEnumerable : IEnumerable<int> { ... }

  private class FromToEnumerator : IEnumerator<int> { ... }
}
```

The enumerable class and the enumerator class

```
private class FromToEnumerable : IEnumerable<int> {    // Static member class
  internal readonly int m, n;
  public FromToEnumerable(int m, int n) { this.m = m; this.n = n; }
  public IEnumerator<int> GetEnumerator() { return new FromToEnumerator(this); }
}
private class FromToEnumerator : IEnumerator<int> {    // Static member class
  private readonly FromToEnumerable eble;
  private int i;
  public FromToEnumerator(FromToEnumerable eble) { this.eble = eble; i = eble.m-1; }
  public int Current {
    get {
      if (eble.m <= i && i <= eble.n)
        return i;
      else
        throw new InvalidOperationException();
    }
  }
  public bool MoveNext() {
    if (i <= eble.n)
      i++;
    return i <= eble.n;
  }
  public void Dispose() { eble = null; }
}
```

C# 2.0: Writing an enumerable using the yield statement

With the yield statement, the FromTo method can be written like this:

```
public static IEnumerable<int> FromTo(int m, int n) {
  for (int i=m; i<=n; i++)
    yield return i;
}
```

The FromToEnumerable and FromToEnumerator classes are no longer needed!

An *iterator* method is one that contains at least one yield statement.and has return type IEnumerable<T> or IEnumerator<T>.

The yield statement can be used only in iterator methods.

There are two forms of the yield statement:

- yield return e; causes the next value of the enumerator to be that of e.
- yield break; signals that the enumerator has no more values.
  Same as returning from or reaching the end of the iterator method.

New in C# 2.0: Partial type declarations

In C# 2.0, a class, interface or struct may be declared in several parts, contained in separate source files.

Useful if one part is generated by a program generator, and another part contains manual adaptations.

Regenerating the generated part will not destroy the manual adaptations.

Example: Two files, each containing part of the declarations of interface I and class C:

```
partial interface I {                   |       partial interface I {
  void M2(C.S n);                       |         void M1(C.S n);
}                                       |       }
sealed partial class C : I {            |       public partial class C {
  public void M1(S n) {                 |         public partial struct S {
    if (n.x > 0)                        |           public int x;
      M2(n.Decr());                     |           public S Decr() { x--; return this; }
  }                                     |         }
  public partial struct S {             |         public void M2(S n) {
    public S(int x) { this.x = x; }     |           Console.WriteLine("n.x={0} ", n.x);
  }                                     |           M1(n);
  public static void Main() {           |         }
    C c = new C();                      |       }
    c.M1(new S(5));                     |
  }                                     |
}                                       |
```

A modifier on one part applies to all parts of a class, interface or struct.

New in C# 2.0: SQL-style nullable types

In SQL, any value, such as an integer, may be null.

Calculations preserve nulls, so $17 +$ null gives null.

C# will be used for stored procedures in Microsoft SQL Server. This requires support for null values.

If t is a value type, then t? is a *nullable type* over t. The notation t? is shorthand for Nullable<t>.

The nullable type t? has the values of t and the additional value null.

There is an implicit conversion from t to t?, and an explicit conversion (cast) from t? to t.

The usual arithmetic (+, -, *, . . .) and logical (&, |, !, . . .) operators are lifted to work on nullable simple values:

```
int? i1=11, i2=22, i3=null, i4=i1+i2, i5=i1+i3;              // 11 22 null 33 null
int i6 = (int)i1;                                 // Legal: cast from int? to int
int i7 = (int)i5;                                 // Legal but fails at run-time
int i8 = i1;                                      // Illegal, no implicit conversion
int?[] iarr = { i1, i2, i3, i4, i5 };
i2 += i1;                                         // Result 33 = 22 + 11
```

A nullable type Nullable<T> implements interface INullableValue.

If x has type Nullable<T> then x.HasValue means x!=null and x.Value of type T is defined only when x!=null.

The null-test operator ?? is a way to provide a fallback value

e1	e2	e1 ?? e2
null	v2	v2
v1	v2	v1

Assume that iarr of type int?[] holds { 11, 22, null, 33, null }.

Compute the product of the non-null elements (namely, $7986 = 11 \cdot 22 \cdot 33$):

```
int prod = 1;
for (int i=0; i<iarr.Length; i++)
  prod *= iarr[i] ?? 1;
```

Print the non-null elements greater than 11 (namely, 22 33):

```
for (int i=0; i<iarr.Length; i++)
  if (iarr[i] > 11)                     // true if non-null and > 11
    Console.Write("[{0}] ", iarr[i]);
```

Print the elements different from 11 (namely, 22 null 33 null):

```
for (int i=0; i<iarr.Length; i++)
  if (iarr[i] != 11)                    // true if null or != 11
    Console.Write("[{0}] ", iarr[i]);
```

Convenient, but now we have *both* the null reference, and the absent value null of a value type.

The `bool?` type and three-valued logic

The nullable type `bool?` has three values: `false`, `true`, and `null` (= don't know).

Most lifted operators (+, *, ^, <, . . .) are `null`-strict: they give the result `null` if any argument is `null`.

But the lifted strict logical operators (&) and (|) produce `true` or `false` whenever possible:

x&y	null	false	true
null	null	false	null
false	false	false	false
true	null	false	true

x|y	null	false	true
null	null	null	true
false	null	false	true
true	true	true	true

The `null` value is considered false in conditional expressions (`?:`) and in conditional statements (`if`, `while`, `do-while` and `for`).

Consequence: it no longer holds that `(e1 ? e2 : e3)` and `((!e1) ? e3 : e2)` are equivalent.

Basics of graphical user interfaces (GUI) in .Net

The current technology for making GUIs in .Net is called WinForms.

See namespaces System.Drawing and System.Windows.Forms and their neighbours.

GUI components — forms, buttons, menus, tables, textboxes — are created as objects.

It is similar in many respects to Java's Swing library (but seems to have little automatic layout management).

The next version of Microsoft Windows, codenamed Longhorn, has a new GUI system called Avalon.

See `http://msdn.microsoft.com/longhorn/`

Avalon is declarative and uses XAML, an XML-language, to describe the the structure and functionality of GUIs.

The rendering model is very similar to Scalable Vector Graphics (SVG) from WWW Consortium.

WinForms will remain supported also in Longhorn, and Avalon components can be included in a WinForms GUI.

But Avalon is recommended for Longhorn-only development.

WinForms example (file `Theatre.cs`)

A Form in WinForms is a windows that can contain other components; it corresponds to a JFrame in Java Swing:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class MyTest {
  public static void Main(String[] args) {
    Form form = new Form();
    form.Text = "Inferial Bio";
    TheatrePanel panel = new TheatrePanel(10, 15);
    form.Controls.Add(panel);
    form.ClientSize = panel.Size;
    form.StartPosition = FormStartPosition.CenterScreen;
    form.ShowDialog();
  }
}
```

The form has a single 'control', a TheatrePanel (see next slide).

A panel on which to draw the cinema seats

We declare a TheatrePanel to display the seating in a cinema. It is a subclass of Panel.

A Panel can contain other panels, buttons and so on, and one can paint on it. Similar to JPanel.

```
public class TheatrePanel : Panel {
  private int sw = 20, sh = 20;
  private bool[,] seats;

  public TheatrePanel(int rows, int cols) {
    this.seats = new bool[rows,cols];  // false = free, true = sold
    this.BackColor = Color.White;
    this.Size = new Size(seats.GetLength(1) * sw, seats.GetLength(0) * sh);
    // Use double buffering in graphics to avoid flickering on repaint:
    this.SetStyle(ControlStyles.AllPaintingInWmPaint
                | ControlStyles.UserPaint
                | ControlStyles.OptimizedDoubleBuffer, true);
  }
  protected override void OnPaint(PaintEventArgs e) {
    ... called when the TheatrePanel needs to be redrawn ...
  }
  protected override void OnMouseClick(MouseEventArgs e) {
    ... called when a mouse click happens within the panel ...
  }
}
```

The `seats` array represents the state of cinema seats (false = free, true = sold).

Drawing the cinema's seats

The `OnPaint` method is called (by the window system) when the TheatrePanel needs to be redrawn.

As in Java, drawings are made on the panel's Graphics object.

We draw a free seat as a green blob, a sold seat as a red blob.

```
protected override void OnPaint(PaintEventArgs e) {
  if (seats != null) {
    Graphics g = e.Graphics;
    SolidBrush brush = new SolidBrush(Color.Gray);
    for (int row=0; row<seats.GetLength(0); row++) {
      for (int col=0; col<seats.GetLength(1); col++) {
        Rectangle rect = new Rectangle(col*sw, row*sh, 15, 15);
        brush.Color = seats[row,col] ? Color.Red : Color.Green;
        g.FillEllipse(brush, rect);
      }
    }
  }
}
```

This could be improved in a zillion ways: automatically scale seats when window is resized etc.

Reacting to mouse clicks

The `OnMouseClick` method is called when a mouse click happens within the panel.

The `e` argument carries the (x, y) coordinates of the mouse click.

When a click happens within the rectangle containing a seat, we change the seat from free to sold, or back.

The call to `Invalidate` causes the panel to be redrawn, so `OnPaint` gets called.

```
protected override void OnMouseClick(MouseEventArgs e) {
  if (seats != null) {
    int col = e.X / sw, row = e.Y / sh;
    if (0 <= row && row < seats.GetLength(0) &&
        0 <= col && col < seats.GetLength(1)) {
      seats[row,col] = !seats[row,col];
      Invalidate();
    }
  }
}
```

Winforms example: Displaying a data grid (file `Sheet.cs`)

A DataGridView is a spreadsheet-style GUI component, but without any underlying functionality.

```
Form form = new Form();
form.Text = "SuperCalc 2005";
DataGridView dgv = new DataGridView();
dgv.ShowEditingIcon = false;
dgv.ColumnCount = 70;
dgv.RowCount = 40;
dgv.AllowUserToAddRows = false;
// Put labels on columns and rows:
for (int col=0; col<dgv.ColumnCount; col++)
  dgv.Columns[col].Name = ColumnName(col);
for (int row=0; row<dgv.RowCount; row++)
  dgv.Rows[row].HeaderCell.Value = (row+1).ToString();
// Set data grid size, add to form, and display:
dgv.Size = new System.Drawing.Size(800,500);
form.Controls.Add(dgv);
form.ClientSize = dgv.Size;
form.StartPosition = FormStartPosition.CenterScreen;
form.ShowDialog();
```

This creates and displays a 40-row, 70-column data grid with row and column headers, scrollbars etc.

The `ColumnName` method (not shown) converts 0 1 2 . . . to column names A B . . . Z AA AB . . . AZ BA BB . . .

Event handlers: reacting to cell entry and exit etc.

Add an event handler to show current cell's coordinates in top lefthand corner.

An event handler is a delegate.

The `CellEnter` event is raised when the used gives focus to a cell.

The effect of raising an event is to call the delegates associated with it.

```
dgv.CellEnter +=
  delegate(Object sender, DataGridViewCellEventArgs arg) {
    int row = arg.RowIndex, col = arg.ColumnIndex;
    dgv.TopLeftHeaderCell.Value = ColumnName(col) + (row+1);
  };
```

Class System.Windows.Forms.Control has events (MouseClick, Paint) and corresponding methods (OnMouseClick, OnPaint) as seen in TheatrePanel.

Creating forms with the Visual Studio designer

The *normal* way to create WinForms is to use the Visual Studio graphical GUI designer.

Choose `File | New | Project` and `Windows Application`.

When you switch from design view to code view, you get a partial class!

Your code (event handlers) go in file `Form.cs`; auto-generated code goes into file `Form1.Designer.cs`.