New in C# 2.0: Anonymous methods: delegate expressions (C# Precisely section 12.20)

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

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
public delegate int DComparer<T>(Tv1, Tv2);
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
public delegate RFun<A,R>(Ax);
public delegate RFun<A1,A2,R>(A1x1,A2x2);
```

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

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

We can use it like this:

```csharp
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:

```csharp
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 (C# Precisely section 13.12)

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, ..., n`:

```csharp
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

```csharp
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 FromToEnumerable emeble;
        private int i;
        public FromToEnumerator(FromToEnumerable emeble) { this.emeble = emeble; i = emeble.m-1; }
        public int Current {
            get {
                if (emeble.m<=i&&i<=emeble.n)
                    return i;
                else
                    throw new InvalidOperationException();
            }
        }
        public bool MoveNext() {
            if (i < emeble.n)
                i++;
            return i <= emeble.n;
        }
        public void Dispose() { emeble = null; }
    }
}```
C# 2.0: Writing an enumerable using the yield statement

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

```csharp
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 (C# Precisely section 26)

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:

```csharp
partial interface I {
    void M2(C.S n);
}

sealed partial class C : I {
    public void M1(C.Sn) {
        if (n.x > 0)
            M2(n.Decr());
    }
    public partial struct S {
        public partial void M1(S n) {
            public partial struct S {
                public partial S Decr() {
                    x--; return this;
                }
                public void M2(S n) {
                    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 (C# Precisely section 18)

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

Calculations preserve nulls, so 17 + null gives null.

To simplify interaction with databases, C# 2.0 supports null also for value types.

If `t` is a non-nullable value type, then `t?` is a nullable type over `t`.

The type `t?`, which is shorthand for Nullable<T>, has the values of `t` and the additional value `null`.

There is an implicit conversion from `t` to `t?`, and an explicit conversion from `t?` to `t`.

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

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

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-coalescing operator ?? is a way to provide a fallback value

```csharp
el e2 el ?? e2
null v2 v2
v1 v2 v1
```

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

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

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

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

```csharp
for (int i = 0; i < arr.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):

```csharp
for (int i = 0; i < arr.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:

<table>
<thead>
<tr>
<th>x &amp; y \null</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>null</code></td>
<td><code>null</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>null</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>null</code></td>
<td><code>true</code></td>
<td><code>null</code></td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td></td>
</tr>
</tbody>
</table>

Boxing and nullable types (a subtlety)

At the last moment (summer 2005), Microsoft changed the design of boxing of nullable value types:

- Boxing of a `null` value of type `int?` produces a `null` reference.
- Boxing of a non-`null` value of type `int?` produces a boxed `int` – not a boxed `int?`.

Example:

```csharp
int? b1 = null, b2 = 17;
Object o1 = b1, o2 = b2;
```

Now `o1` would be a `null` reference, and `o2` would be a boxed 17.

As an interesting consequence, a boxed `int` can be turned into an `int?`:

```csharp
Object o1 = null, o2 = 17;
int? b1 = (int?)o1, b2 = (int?)o2;
```

Now `b1.HasValue` would be `false`, and `b2.HasValue` would be `true`.

This design makes `null` comparison work as expected, when a generic type parameter is instantiated with a nullable type:

```csharp
class C<T> {
    public bool M(T x) { return x == null; }
}

new C<int?>.M(null) returns true as expected ...
```

On the horizon: C# 3.0

Version 3.0 of C# has several new interesting features:

- Type inference (\texttt{var} declarations)
- Lambda expressions – an even neater notation for anonymous methods
- Extension methods – add an instance method to another class or interface
- Anonymous class types
- Expression trees – turn a lambda expression into an object
- Linq = Language Integrated Query, to access object structures, XML documents, and relational databases

This requires no changes to the .NET runtime system and therefore can be released fairly soon – 2007 or 2008?


See also last lectures in the ITU F2006 courses Advanced Object Oriented Programming and Komponent-baseret Design og J2EE.

Basicsof graphical user interfaces (GUI) in .Net

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


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.


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

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 window that can contain other components; it corresponds to a JFrame in Java Swing:

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

class MyTest {
    public static void Main(String[] args) {
        Form form = new Form();
        form.Text = "InferialBio";
        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.

```csharp
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) {
        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++) {
                    if (seats[row, col] == true) {
                        brush.Color = Color.Red;
                    } else {
                        brush.Color = Color.Green;
                    }
                    g.FillEllipse(brush, rect);
                }
            }
        }
    }

    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();
            }
        }
    }
}
```

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.

```csharp
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++) {
                if (seats[row, col] == true) {
                    brush.Color = Color.Red;
                } else {
                    brush.Color = 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.

```csharp
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.

```csharp
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.

```csharp
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 (MouseMove, 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.