

C# Overview

Part IV

Tomáš Matoušek
&
Ladislav Prošek

tmd.havit.cz/teaching/csharp.htm

“Reserved” Method Names

- used by C# for properties, indexers, events and overloaded operators
 - get_* (property getter)
 - set_* (property setter)
 - get_Item (indexer getter)
 - set_Item (indexer setter)
 - add_* (event handler concatenation)
 - remove_* (event handler removal)
 - op_Addition (binary +)
 - op_Subtraction (binary -)
 - op_Implicit (implicit cast)
 - op_Explicit (explicit cast)
 - ...

Delegates

- type-safe pointers to methods
- delegate is a class (subclass of System.MulticastDelegate)
- declaration defines target method signature

```
delegate void MyDelegate(string x);
```

- delegate can “point” to both static and instance methods

```
class A
{
    static void f(string x) { }
    void g(string x) { }

    static void Foo(MyDelegate d)
    {
        if (d != null) d("bar"); // invokes the target method
    }

    static void Main(string[] args)
    {
        Foo(new MyDelegate(f));
        Foo(new MyDelegate(new A().g));
    }
}
```

Events

- aid for implementing the publisher/subscriber push pattern
- look like fields of delegate type marked with the event keyword

```
class A
{
    public event MyDelegate Event; // backed by a private MyDelegate field

    void Fire(string x)
    { if (Event != null) Event(x); } // "fires" the event
}
```

- explicit accessor declaration

```
class B
{
    public event MyDelegate Event
    {
        add { Console.WriteLine(value); } // called when subscribing
        remove { Console.WriteLine(value); } // called when unsubscribing
    }
}
```

- subscribing to the event with +=, unsubscribing with -=

```
A a = new A();
a.Event += new MyDelegate(f); a.Event -= new MyDelegate(g);
```

Exceptions

- uniform error-handling model
- exceptions change flow of control when something abnormal happens
- all exceptions inherit from System.Exception class
- no checked exceptions
 - programmer is not required to list thrown exceptions in method signatures

```
try
{
    // protected block of code
}
catch (ArgumentNullException e)
{
    // executes only if ArgumentNullException (or its subclass) is thrown
}
catch (ArgumentException e)
{
    // executes only if ArgumentException (but not ArgumentNullException) is thrown
}
finally
{
    // always executes regardless of exception occurrence
}
```

Exceptions (cont.)

- throwing exceptions

```
throw new ArgumentException();
```

- rethrowing caught exceptions

```
catch (ArgumentException e)
{
    // ...
    throw;
}
```

- common exceptions

System.NullReferenceException	Attempt to dereference a null object reference.
System.ArgumentException	Supplied argument is "invalid".
System.InvalidCastException	Attempt to perform an invalid type cast.
System.IndexOutOfRangeException	Attempt to access an out-of-range array element.
System.IO.IOException	Thrown when an I/O error occurs.

Unsafe C#

- types, methods and blocks of code marked with the unsafe keyword may work with pointers
 - have to be compiled with /unsafe compiler option
 - function pointers disallowed even in unsafe code
 - useful for dealing with legacy data structures, advanced COM Interop and P/Invoke, performance critical code, etc.

```
unsafe static void Copy(byte *dest, byte *src, uint length)
{
    while (length-- > 0) *dest++ = *src++;
}
```

- pointers can only point to blittable types
 - value types that do not embed object references
- unsafe code is not verifiable – unsafe programs require full trust

Unsafe C# (cont.)

references vs. pointers

- managed reference
 - 32-bit number containing address of an object on managed heap
 - the number changes as the object is moved by GC
 - cannot point “inside” an object
- managed pointer (&)
 - can point to fields and array elements (“interior” pointers)
 - can be stored only in locals and parameters
 - not directly available in C#
 - ref/out parameters implemented as managed pointers
- unmanaged pointer (*)
 - 32 bit-number containing address of something somewhere
 - the number is fully under programmer’s control – not tracked by GC
 - if it points to an objects on managed heap, the object has to be pinned – temporarily avoided from being moved by GC (fixed keyword)

XML Comments

- comments that begin with `///` are handled as XML by the C# parser
- have to immediately precede type or member declaration

```
/// <summary>  
/// The main entry point for the application.  
/// </summary>  
/// <param name="args">Command line arguments.</param>  
static void Main(string[] args)
```

- if `/doc:file` compiler option is used, XML comments are extracted from source files during compilation and placed into one XML file
 - XML transformed to documentation using tools (e. g. NDoc)
- predefined tags known to the compiler
 - `<summary>` `<remarks>` `<para>` `<see>` `<seealso>` `<param>` `<return>` `<exception>` `<paramref>` `<code>` `<list>` `<include>` `<example>` ...
- unknown tags written out verbatim along with their contents
 - useful for HTML

Numbers

- primitives
 - integer: sbyte, byte, ushort, short, int, uint, long, ulong
 - floating-point: single, double
 - fixed-point: decimal
- literal suffixes
 - integer: U (uint/ulong), L (int/long), UL (ulong)
 - default type is the first suitable among int, uint, long, ulong
 - floating-point: F (single), D (double), M (decimal)
 - default type is double
- implicit widening to nearest suitable, explicit narrowing
- binary integer operators defined on int, uint, long, ulong only
- unary integer operators defined on all integer primitives

```
byte b = 1;
b = (byte) (b + b);    // operator + takes and returns 32 bit integers
b++;                  // equivalent to b = (byte) (b + 1);
b *= 10;              // equivalent to b = (byte) (b * 10);
```

Checked & Unchecked Operations

- integer arithmetic operations are checked by default
 - exception `OverflowException` thrown on overflow/underflow
 - can be changed by `/checked` compiler option
- checked and unchecked keywords
 - applied on enclosed arithmetic operations
 - change default behavior set by `/checked` compiler option

```
byte b = 1, y = 2, x = 250;  
b = unchecked((byte)(b + 100));  
unchecked { y *= x; }  
b *= unchecked((byte)1000);
```

- floating-point operations never throw an exception
 - `NaN`, `PositiveInfinity`, `NegativeInfinity` values instead

```
double x = -1.0, y = 0.0;  
x /= y;  
if (x == Double.NegativeInfinity) { ... }
```

Characters & Strings

- all characters are Unicode, 2B wide

- char

```
char A = 'A';  
char B = '\u0041';
```

- immutable strings

```
string str = "Hello,/n \u0157!";  
string path = @"C:\Windows\System32";
```

- string builders (System.Text.StringBuilder)

```
StringBuilder sb = new StringBuilder("hello");  
sb[0] = 'H';  
sb.Append(" world!");  
string s = sb.ToString();
```

- builders performs modifications much faster than strings
- strings are immutable though a copies have to be made

Switching Over Strings

```
switch(str)
{
    case "AAA": ... break;
    case "BBB": ... break;
    default: ... break;
}
```

- syntactic sugar for
 - a sequence of "if-then-else" statements and string *interning* or (depending on the number of cases)
 - creating a hash table and switching over integers
- C# 1.0 differs from C# 2.0 in implementation

Array Types

- all array types derived from System.Array class

- vectors

```
int[] a = {1,2,3};
```

- multi-dimensional arrays

```
int[,] b = {{0,1},{2,3},{4,5}};
```

- jagged arrays (arrays of arrays)

```
int[][] c = {new int[] {0,1},new int[] {2,3,4}}
```

```
string[,] d = { new string[10,10], new string[1,2] };
```

- arrays with non-zero lower bounds (no special syntax in C#)

```
Array e = Array.CreateInstance(  
    typeof(int),           // element type  
    new int[] {1,2,3},     // lengths  
    new int[] {8,9,10});   // lower bounds
```

Custom Attributes

- a way how to assign a piece of information (an attribute) to an arbitrary language element (method, field, ...)
- useful for compilers and other tools
- used also by CLR and C#

- a structure of attribute's data defined by a class
 - derived from System.Attribute class
 - with name suffixed with "Attribute" by convention
 - e.g.: FlagsAttribute, ThreadStaticAttribute, ...

```
// class defining attribute data:
public class MyAttribute : Attribute
{
    // constructor:
    public MyAttribute(int i)
    {
        this.data = data;
    }

    private int data;
}
```

```
// attribute used on various lang. elements:
[My(1)]
public class MyClassWithAttribute
{
    [My(2), return: My(5)]
    public string MyMethod([My(8)] string arg)
    {
        ...
    }
}
```