

C# Overview

Part III

Tomáš Matoušek
&
Ladislav Prošek

tmd.havit.cz/teaching/csharp.htm

Compiler Known Interface: IEnumerable

- interface used by foreach statement
 - each class implementing IEnumerable can be used in foreach stmt.

```
ArrayList a = new ArrayList();  
a.Add("first");  
a.Add("second");  
a.Add("third");  
foreach (string s in a)  
{  
    Console.WriteLine(s);  
}
```

- general pattern
 - foreach (*type identifier in expression*) *statement*
 - each enumerated element is casted to the specified *type*
 - *expression* implements IEnumerable interface
 - not a mandatory condition, several more patterns are possible

Compiler Known Interface: IDisposable

- disposable pattern
 - deterministic clean-up by IDisposable.Dispose() method
 - implemented by classes requiring clean-up after the job is done
 - usually classes holding unmanaged system resources (handles)
 - e.g.: file streams
- syntactic sugar in C# exploiting using keyword:

```
using (FileStream fs = File.Create(path))
{
    /* some useful job */
}
```

is equivalent to:

```
FileStream fs = File.Create(path);
try
{
    /* some useful job */
}
finally
{
    if (fs!=null) ((IDisposable)fs).Dispose();
}
```

Enumerated Types & Bit Flags

- based on structure System.Enum
- enums (one only)

```
enum Colors : int { Red, Green, Blue };  
enum Choice : byte { Yes, No };
```

```
Colors c = Colors.Red;
```

- bit flags (combinable using bitwise operators)

```
[Flags] enum Access : int  
{  
    Read = 1,  
    Write = 2,  
    Execute = 4  
};
```

```
Access a = Access.Read | Access.Write;  
if ((a & Access.Write) != 0) { ... }
```

Type Objects

- a type is not loaded until used for the first time
 - how types are resolved will be described in later lectures
 - class constructor (.cctor) called when type is loaded
- each type represented by an instance of System.Type
- got by
 - typeof operator
`typeof(type_identifier)`
 - or GetType() instance non-virtual method
`instance.GetType()`
- type object stored in vtable header
 - created on-demand
 - used by reflection

Fields

- data associated with an object or a type
- visibility
 - public – visible to everyone
 - protected – visible to the declaring type and its subtypes
 - private – visible to the declaring type only (default)
 - internal – visible to all types in current assembly
 - protected internal – union of protected and internal
- static fields
 - associated with a type, shared by all instances
 - initial value (if present) assigned to in the static constructor

```
public static int X = 1;
```

- instance fields
 - associated with particular instances
 - initial value (if present) assigned to in all instance constructors

```
public int Y = 2;
```

Fields (cont.)

- constant fields
 - const keyword, inherently static
 - type of the constant limited to
 - numbers
 - string
 - reference types initialized to null
 - constant use is replaced with its value by the compiler
 - no representation at run-time

```
private const string s = "Hello";
```

- readonly fields
 - readonly keyword, works for both static and instance fields
 - can be assigned to only in a constructor

```
public static readonly int Year;
```

Fields (cont.)

- volatile fields
 - volatile keyword
 - prohibits reordering of instructions related to the field
 - prohibits storing the value in register
 - stronger than volatile in Java
 - double checked-locking pattern correct in C#

```
internal static volatile object singleton;
```

- thread-static fields
 - static field decorated with the ThreadStatic attribute
 - each thread has its own value

```
[ThreadStatic]  
public static object context;
```

Fields (cont.)

- new keyword should be used when hiding an inherited field by a field of the same name

```
class A
{
    public int X;
}

class B : A
{
    new public string X;
}
```

- field name and the name of its type can be identical

```
public Color Color;
```

Methods

- code associated with an object or a type
- visibility
 - public, protected, private, internal, protected internal
 - the same meaning as for fields
- static methods
 - do not operate on a specific instance

```
public static void Foo()  
{ }
```

- instance methods
 - operate on a given instance that is accessible via this
 - either non-virtual

```
public void Foo()  
{ }
```

- or virtual (virtual keyword)

```
public virtual void Foo()  
{ }
```

Methods (cont.)

- hiding non-virtual methods
 - new keyword should be used

```
class A
{
    public void Foo()
    { }
}
class B : A
{
    new public void Foo()
    { base.Foo(); } // invokes the hidden method
}
```

- overriding virtual methods
 - override keyword has to be used (instead of virtual)

```
class A
{
    public virtual void Foo()
    { }
}
class B : A
{
    public override void Foo()
    { base.Foo(); } // invokes the overridden method
}
```

Methods (cont.)

- overloading
 - more than one method of the same name may be defined
 - as long as they differ in number or types of parameters
 - appropriate overload selected according to actual parameters
- abstract methods
 - abstract keyword, no body
 - allowed in abstract classes only
 - inherently virtual \Rightarrow implemented with override in a subclass

```
public abstract void Foo();
```

- sealed methods
 - sealed keyword
 - applies to override methods only
 - prevents the method from being further overridden

```
public sealed override void Foo()  
{ }
```

Method Parameters

- parameters passed by value
 - declared with no modifiers
 - passed “by-value”
 - creates a new storage location
 - assigning a new value has no effect on the actual parameter

```
public void Foo(int x)
{
    x = 1; // does not propagate back to caller
}

// invocation:
Foo(0);
```

Method Parameters (cont.)

- parameters passed by reference
 - declared with the ref modifier
 - passed “by-reference”
 - represents the same storage location as the actual parameter
 - address of the actual parameter (managed pointer) is passed to the method

```
public void Foo(ref int x)
{
    x = 1; // propagates back to caller
}
```

```
// invocation:
int x = 0;
Foo(ref x);
```

Method Parameters (cont.)

- output parameters
 - declared with the out modifier
 - passed “by-reference”
 - like reference parameters but
 - initially considered unassigned
 - must be definitely assigned before the method returns

```
public void Foo(out int x)
{
    x = 1; // propagates back to caller
}
```

```
// invocation:
int x;
Foo(out x);
```

Method Parameters (cont.)

- parameter arrays
 - parameters declared with the params modifier
 - must be the last one and of single-dimensional array type
 - if invocation specifies zero or more parameters implicitly convertible to the array element type, the array is automatically created and filled with the parameters
 - syntactic sugar

```
public void Bar(params string[] args)
{
    // args contains { "a", "b", "c" }
}

// invocation:
Bar("a", "b", "c");
```

Instance Constructors

- instance methods with name identical to the name of the enclosing type and without return value
- called during instantiation with parameters given to new

```
class A
{
    public A() { }
    public A(string x) { }
}

class B
{
    public B() : this("default") { }
    public B(string x) : base(x) { }
}

// instantiating the class:
A a1 = new A();
A a2 = new A("test");
```

- types may define more instance constructors that differ in signature
- subtypes do not inherit supertype's constructors
- if no user-defined constructor exists, the compiler creates default parameterless constructor

Static Constructors

- parameterless static method with name identical to the name of the enclosing type and without return value
- called by the runtime when the type is loaded, which happens when:
 - an instance of the type is created
 - any of the static members of the class are referenced
 - precise behavior depends on the beforefieldinit type attribute

```
class A
{
    static A()
    { }
}
```

- only one static constructor may be defined for a type
- executes at most one
 - static constructors are thread-safe

Properties

- pairs of get & set methods accessible with the field-like syntax

```
class A
{
    private string name;

    public string Name // public property
    {
        get { return name; }
        set { name = value; }
    }
}

// accessing the property:
A a = new A();
a.Name = "test";
Console.WriteLine(a.Name);
```

- syntactic sugar
- properties can be virtual and can be static
- one of get, set may be omitted
 - read-only or write-only property

Indexers

- properties that enable instances to be indexed in the same way as arrays

```
class Matrix
{
    object[,] data;

    public object this[int row, int col]
    {
        get { return data[row, col]; }
        set { data[row, col] = value; }
    }
}
```

```
// indexing instances:
Matrix m = new Matrix();
m[0, 1] = 3;
Console.WriteLine(m[2, -1]);
```

- syntactic sugar
- indexers cannot be static
- inherited indexer is accessed using the base[] syntax

Operators

Operator category	Operators
Arithmetic	+ - * / %
Logical (boolean and bitwise)	& ^ ! ~ && true false
String concatenation	+
Increment, decrement	++ --
Shift	<< >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &= = ^= <<= >>=
Member access	.
Indexing	[]
Cast	()
Conditional	?:
Delegate concatenation and removal	+ = - =
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &

Operator Overloading

- user-defined classes and structures can overload certain operators

– unary: + - ! ~ ++ -- true false

```
public static Complex operator -(Complex a)
{
    return new Complex(-a.re, -a.im);
}
```

– binary: + - * / % & | ^ << >> == != > < >= <=

```
public static Complex operator +(Complex a, Complex b)
{
    return new Complex(a.re + b.re, a.im + b.im);
}
```

– implicit and explicit cast

```
public static implicit operator Complex(double d)
{
    return new Complex(d, 0);
}
public static explicit operator double(Complex a)
{
    if (a.im != 0) throw new InvalidCastException();
    return a.re;
}
```