

C# Overview

**Tomáš Matoušek
&
Ladislav Prošek**

tmd.havit.cz/teaching/csharp.htm

C# Program Structure

- types
 - code and data descriptors
 - visibility: public, internal (default)
- code defined in methods
 - no global functions (C# limitation, not the CLR one)
 - visibility: public, private (default), protected, internal, internal protected
- data defined by fields
 - called “attributes” in Java
 - no global variables (C# limitation)
 - visibility: public, private (default), protected, internal, internal protected
- metadata
 - generated by the C# compiler
 - custom metadata (attributes) can be defined and used in C#

Preprocessor Directives

- `#define`, `#undef`
 - define/remove a symbol
 - no value possible (no macros)
 - at the beginning of a source file
 - compiler option `/define`
- `#if`, `#else`, `#elif`, `#endif`
 - conditional compilation depending on defined symbols
- `#region`, `#endregion`
 - ignored by preprocessor, useful for tools like VS.NET

Namespaces

- types declared in namespaces
 - dot notation, e.g. System.Collections.Hashtable
 - subnamespaces
 - not related to directory structure nor source files (unlike Java packages)

```
namespace A.B.C
{
    namespace D
    {
        class MyClass1 { ... } // class's namespace is A.B.C.D
        class MyClass2 { ... }
    }
    class MyClass3 { ... } // class's namespace is A.B.C
}
```

- using keyword
 - allows using short names of types declared in the specified namespace
 - allows create type name aliases and namespace aliases

```
using MyCls = A.B.C.D.MyClass1;
using MyNamespace = A.B;
using A.B.C;
```

BCL Namespaces

- Base Class Library namespaces (implemented in mscorlib)
 - System
 - System.Text
 - System.Reflection
 - System.Threading
 - System.IO
 - System.Collections
 - System.Globalization
 - System.Diagnostics
 - System.Security
 - ...

Types

- type of each managed data must be known
- common supertype: `System.Object`
 - `ToString`, `GetType`, `Equals`, `GetHashCode`, `Finalize`, `MemberwiseClone`
- fundamental division
 - reference types
 - allocated on managed heap
 - accessed via a reference
 - value types
 - allocated on the stack or embedded to an object or array
 - simple data or compound structures
 - boxing converts a value to an object
 - sealed subtypes of `System.ValueType`
- primitive types
 - types known to a particular language compiler
 - C#
 - `System.Boolean` (bool), `System.Int32` (int), `System.Double` (double), `System.Char` (char), `System.String` (string), `System.Object` (object), ...

Boxing & Unboxing

- value storages
 - values allocated on stack
 - not managed by GC (however, GC knows about them)
 - lifetime during a method execution
 - freed when a stack frame is released
 - finalizer is not called
 - values embedded into objects or arrays
 - value lifetime corresponds to the object/array lifetime
 - not managed by GC, finalizer not called on release
 - boxed values on the managed heap
 - value is wrapped to an object allocated on heap
 - finalizer called if defined (which is not the case in C#)
 - implicit boxing when value is used as reference in C#

```
object o = 1;
```

- explicit unboxing

```
int a = (int)o;
```

Type Casting

- explicit cast
 - may lead to a compile-time or a run-time error
 - throws an `InvalidCastException` on run-time failure

```
string str = (string) obj;
```

- is operator
 - returns whether an object is of a specified type

```
bool b = obj is string;
```

- as operator
 - casts an object to a specified reference type
 - cannot be used on value types
 - returns a null reference if on failure

```
string str = obj as string;  
if (str != null) Console.WriteLine(str.ToLower());
```

Nested Types

- declared inside another type declaration
 - influences visibility only
 - no other relationship between enclosing and nested classes

```
public class A
{
    private static int x;
    private int m() { return 1; }

    private class B // nested class (a "friend" of A)
    {
        public f(A a) { return a.m() + x; }
    }
}
```

- Java inner classes emulation
 - inner class = nested class + field *outer* initialized in ctor

```
public class Outer
{
    private class Inner
    {
        private readonly Outer outer;
        public Inner(Outer outer) { this.outer = outer; }
    }
}
```

Interfaces

- contracts
- inheritance in CLR
 - single implementation inheritance
 - method implementations are inherited
 - multiple interface inheritance
 - methods have to be implemented unless type is abstract
- declaring an interface
 - interface keyword
 - contains
 - public virtual instance methods
 - CLR, not C#: static fields, methods, ctors, constants
 - can implement other interfaces
 - inherits a contract

Implementing Interfaces

- interface method implementation
 - same name & signature (param. names may differ)
 - has to be public (keyword required)
 - can be virtual or sealed (if virtual keyword missing)
- explicit interface method implementation
 - inherent problem of multiple inheritance
 - more methods having the same name and signature
 - implementations prefixed with interfaces
 - no access modifiers
 - not accessible via implementing class references (\Rightarrow private)
 - accessible only using reference typed to the interface (\Rightarrow public)
 - only interfaces directly implemented by the class are allowed here
 - another use:
 - hide implementation of an internal interface

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	+ (at least one operand has to be a string, no matter which one)
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;
}
```

“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)  
    {  
        ...  
    }  
}
```