

CLI 2.0 & C# 2.0

**Tomáš Matoušek
&
Ladislav Prošek**

tmd.havit.cz/teaching/csharp.htm

Improvements to Version 1.0

- CLR
 - generic types and methods
- BCL
 - generic collections
 - new functionality
- C#
 - generic types and methods
 - anonymous methods
 - iterators
 - type inference
 - nullable types
 - partial types
 - namespace hierarchies

Generics

- goals
 - reduce dynamic type casts
 - which cannot be checked at compile-time
 - reduce boxing/unboxing
 - which waste memory and time
- generic language elements
 - classes, interfaces, structures, methods, delegates
- non-generic language elements
 - properties, events, indexers, operators
- CLR knows generics
- compared to other languages
 - weaker than C++ templates
 - stronger than Java generics
 - JVM doesn't know generics
 - replaced with most generic type (usually object) by compiler
 - e.g. arrays with generic element type not supported by Java

Generic Types

```
class GenericClass<S,T>
{
    private T field;
    public GenericClass() { ... } // instance constructor
    public S M() { ... } // instance method
}
```

- type parameters
 - e.g. S, T
 - types can be overloaded on the number of parameters
- constructed types
 - created from generic types by specifying type parameter(s)
 - open
 - e.g. GenericClass<X,Y>, GenericClass<int,X>
 - closed
 - e.g. GenericClass<int,string>
- a nested type of a generic type is a generic type
 - can use type parameters of the outer type
 - can have additional type parameters
 - these hide outer type's type parameters having the same name

Implementation in CLR

- each closed constructed type has its own
 - method table
 - type object (since it is asked for)
 - static fields
 - .cctor executed for each used closed constructed type
 - .cctor is a convenient place for run-time type parameter checks
- generic code specialization
 - reference type specializations
 - shares one native code
 - value type specializations
 - a specialized native code copy is made for each closed constructed type
 - by JIT, not by C#
 - different from Java
 - Java compiler handles generics, JVM doesn't know about them
 - different from C++
 - templates handled as a kind of preprocessing

Constraints

- constrain a type parameter
 - to inherit from a specified type
 - struct or class keywords can also be used
 - to implement an interface
 - to have a default constructor
- where clause for each constrained type parameter
 - can contain a list of constraints on the type parameter
- constraints allows
 - to access implied instance members

```
class MyDictionary<K,V,S>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
    where S: struct
{
    ... new V(); ...
}
```

```
class SortedList<T> where T: IComparable<T>
{
    public void Add(T item)
    {
        ... if (item.CompareTo(...)) ...
    }
}
```

Default Values

- returns a default value of a specified type parameter
 - null for reference types
 - zero for numeric value, false for boolean, '\0' for character
 - a structure initialized by default values
- via default keyword applied on the type parameter

```
public class C<T>
{
    private T value;
    public T M()
    {
        return (condition) ? value : default(T);
    }
}
```

Notes on Generic Types

- implementation of generic interfaces
 - if there exist values of type parameters such that interface types are the same after substitution
 - ⇒ type declaration is invalid

```
interface I<T> { ... }  
class T<U,V> : I<U>, I<V> { ... } // error
```

- overloading
 - if there exist values of type parameters such that overloads have the same signature after substitution
 - ⇒ overload declaration is invalid

```
class T<U>  
{  
    void f(int a);  
    void f(U a); // error  
}
```

Some BCL Generic Types

- delegates
 - void Action<T>(T)
 - bool Predicate<T>(T)
 - U Converter<T, U>(T)
 - int Comparison<T>(T, T)
 - void EventHandler<T>(object sender, T args) where T : EventArgs
- collection interfaces
 - IEnumerable<T> ... enumerator
 - IEnumerable<T> ... something enumerable via an enumerator
 - ICollection<T> ... a collection of items
 - IDictionary<K,V> ... a collection of key-value pairs
- collection classes
 - Stack<T>
 - Queue<T>
 - List<T>
 - LinkedList<T> ... doubly linked list with a head
 - Dictionary<K,V> ... a hashtable
 - SortedDictionary<K,V>

Generic Methods

```
public static T Find<T>(T[] items, Predicate<T> test) where T : new()
{
    foreach (T item in items) if (test(item)) return item;
    return new T();
}
```

- pattern
 - ... Name<type-parameters>(...) [constraints] { ... }
- signature
 - (Name, number of type parameters, argument types and modifiers)
 - contains neither constraints nor names of the type parameters
- overloading
 - overloads must have different signatures
 - the analogous overloading rule mentioned before must hold
- overriding
 - names of type parameters can differ
 - overriding method must have the same constraints as the overridden one

Generics in IL

```
.class public 'AnotherGenericClass`2' <(object) S, (class IComparable) T>
{
    .field private !T field; // type is specified by T type-parameter
    .field private !0 field; // 0th type parameter

    .method public !!0 F<(object) T> (!!T a) { ... }
}
```

- type parameters (TPs) specified in angle brackets
 - a list of types constraining the parameter specified in parentheses
 - type TP referred via single exclamation mark (!T, !0, ...)
 - method TP referred via double exclamation mark (!!T, !!0, ...)
- C# mangles type names
 - suffixed with a back quote and the number of type arguments
 - is not necessary in IL
- two additional metadata tables
 - GenericParam, GenericParamConstraint

C# Anonymous Methods

```
int[] a = new int[] { 1,2,3 };  
Action<int> f = delegate(int item) { Console.WriteLine(item); }  
Array.ForEach<int>(a, f);
```

- delegate keyword reused
 - creates a new anonymous method yet not a new delegate
 - a trick
 - the resulting anonymous method must be assigned only into a storage which static type is a delegate matching the anonymous signature
 - hence, the delegate class always exists
- arguments of an anonymous method
 - can be omitted if not used
- a return value of an anonymous method
 - should be implicitly convertible to the delegate's one

Pseudo-local Variables

- anonymous method declarations hierarchy
 - the top-most method is always regular (non-anonymous)
 - an anonymous method can be declared in another one
- local variable scope
 - starts within the declaring method
 - ends within the inner-most anonymous method in the hierarchy
 - variables used on multiple levels of hierarchy are not actual locals
 - are said to be **captured** by the method which access them
 - a variable is an **outer variable** of the anonymous method
- display classes
 - stores variables accessible on multiple hierarchy levels
 - stores anonymous methods accessing the variables
 - an instance is created in the method declaring the variables

Implementation Example

```
delegate void D();
class C
{
    static void OuterMethod()
    {
        // pseudo-local variable outer with respect to the following anonymous method:
        int i = 0;
        D d = delegate { Console.WriteLine(i++); };
        d();
    }
}
```

```
delegate void D();
class C
{
    private sealed __Display
    {
        public int i;
        public void lambda_0 { Console.WriteLine(this.i++); }
    }

    static void OuterMethod()
    {
        __Display display = new __Display();
        display.i = 0;
        D d = new D(display, __Display.lambda_0);
        d();
    }
}
```

C# Syntactic Sugar

- delegates can be created implicitly

- constructor can be omitted

```
Array.ForEach<int>(a, new Action<int>(F));  
Array.ForEach<int>(a, F);
```

- type arguments can be omitted

- if the compiler can infer proper types

```
Array.ForEach(a, F);
```

Enumeration in C# 1.0

- IEnumerable
 - express the ability of an object to be enumerated
 - IEnumerator GetEnumerator()
 - creates an enumerator, i.e. an instance of IEnumerable
- IEnumerator
 - provides means for enumeration of a specified object
 - usually an inner class of the enumerated object's class
 - bool MoveNext()
 - moves an internal "pointer" to the next item of the enumeration
 - returns true if the enumeration has not finished, false otherwise
 - object Current()
 - returns the current item of the enumeration
 - void Reset()
 - resets the enumeration, invalidates the internal "pointer"
 - initially, the internal "pointer" is invalid
 - has to be advanced by MoveNext before Current is called

C# Iterators

- makes the writing of enumerators easier and more transparent
- converted to C# 1.0 like enumeration by the compiler
 - 4-state automaton implementing `IEnumerator<T>` is generated
- iterator block
 - a block containing so called yields statement(s)
 - yield return {expression}
 - produces the next item of the iteration
 - yield break
 - ends up the iteration
 - used as a method body, operator body or getter body
 - restrictions
 - return type should be `IEnumerable` or `IEnumerator` (+ generic versions)
 - no return statements (yield return only)
 - implicitly ends with yield break
 - method signature cannot contain ref/out arguments
 - cannot be used in try-catch blocks nor in a finally block

Iterators Example

```
public static IEnumerable<T> Greater<T>(IEnumerable<T> e, T threshold)
    where T : IComparable<T>
{
    foreach (T item in e)
        if (item.CompareTo(threshold) > 0)
            yield return item;
}

int[] a = new int[] {1,2,3,4,5};
foreach (int i in Greater<int>(a,3))
    Console.WriteLine(i);
```

C# Nullable Value Types

- constructed using question mark after the type (T?)
 - e.g. int?, long?, ...
 - useful when working with databases, XML files, ...
- translated by C# to System.Nullable<T> type
 - contains two fields
 - T value
 - bool hasValue
- implicit conversions
 - from a non-nullable type to its nullable form
 - null propagating conversions
 - for every conversion from non-nullable S to non-nullable T
 - exists implicit conversion from
 - S? to T? (null to null, original non-nullable conversion otherwise)
 - S to T? (identity + hasValue set to true)
 - exists explicit conversion from
 - S? to T (identity + exception thrown if !S.hasValue)

Lifted Conversions and Operators

- goal
 - to write conversions and operators only for non-null types
 - lifted implicitly to nullable types
- user defined conversion has a lifted form
 - if source and target types are both non-nullable
 - lifted form propagates null (as implicit conversions do)
- non-comparison operator has a lifted form
 - if operand types and result type are non-nullable
 - a lifted operator is applied if at least one operand is nullable

```
int? x = ..., y = ...;
```

```
int? z = x + y;
```

```
int? u = x + 1;
```

is equivalent to

```
int? x = ..., y = ...;
```

```
int? z = x.HasValue && y.HasValue ? x.Value + y.Value : (int?)null;
```

```
int? u = x.HasValue ? x.Value + 1 : (int?)null;
```

Lifted Conversions and Operators (cont.)

- comparison operator has a lifted form
 - if operand types are non-nullable and return type is boolean
 - comparison with null leads to false except for `null == null`
- null coalescing operator (`a ?? b`)
 - returns **b** if **a** is null, otherwise returns **a**
 - **a** ... nullable or reference type
 - **b** ... type convertible to the type of **a**
 - result ... is of **b**'s type (nullable, non-nullable, or reference)

```
int? a = ..., b = ...;
```

```
string s = ...;
```

```
int? c = a ?? b;
```

```
int d = a ?? -1;
```

```
string t = s ?? "empty";
```

C# Namespace Hierarchies

- global namespace hierarchy
 - types are implicitly placed here
- namespace alias qualifier (::)
 - refers to a particular hierarchy or namespace alias
 - global hierarchy qualifier
`global::System.IO.Stream`
 - alias qualifier
`using SIO = System.IO;`
`... SIO::Stream ...`
- additional hierarchies introduced by extern aliases
 - enables to reference types with the same fully qualified name stored in different assemblies
`extern alias X;`
`extern alias Y;`
`... X::N.T ...`
`... Y::N.T ...`
 - aliases bound to particular assemblies on CSC command line (or in VS.NET)
`csc /r:X=assembly1.dll /r:Y=assembly2.dll`

C# Partial Types

- type declarations can be stored in multiple files
 - recall, IL assembler already has this feature
- useful when
 - a type contains a lot of methods and is too large
 - parts of the type are machine-generated
- a new C# type modifier partial

```
public partial A { private int i; }  
public partial A { public void F() { Console.WriteLine(this.i); } }
```

is equivalent to:

```
public A  
{  
    private int i;  
    public void F() { Console.WriteLine(this.i); }  
}
```

Other C# Features

- accessor modifiers for property getters and setters

```
public int X { get { return x; } internal set { x = value; } }
```

- custom attributes on accessors and type parameters
- static classes
 - static modifier used on the class
 - “sealed class with private constructor” pattern
 - cannot be used as a type of any variable, parameter or field
 - cannot contain instance members
 - have no default constructor

- pragmas

```
#pragma warning disable {a list of warning numbers}
```

```
#pragma warning restore {a list of warning numbers}
```