Microsoft .net

I/S TU technische universität wien

Softwareentwicklung mit MS.NET und C#

Die Sprache C#

Robert Bruckner

13.11.2002

Agenda

- Syntax
- Typen
- OO-Sprachfeatures (Vererbung, Polymorphismus)
- Delegates/Events
- Attribute
- Reflection
- Kommentare
- Gemeinsamkeiten und Unterschiede Java vs. C#
- Zukunft von C# (Generics, etc.)

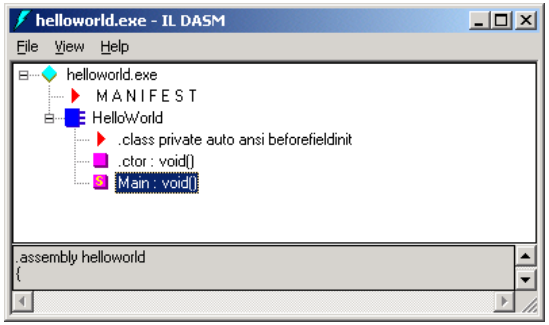
Einleitung

- Mitglied der „C-Familie“
- C# als Bestandteil der .NET Strategie
- Basierend auf CLR
- Konkurrenz zu Java, Evolution zu C++
  - Ähnlichkeiten mit Java
  - Unterschiede zu Java

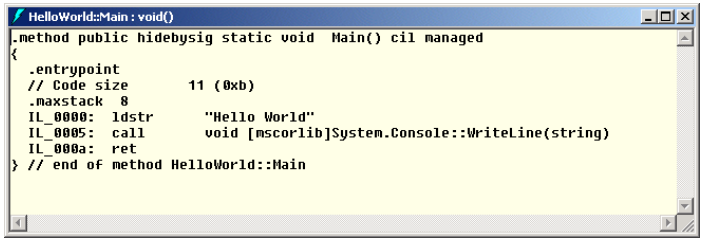
C# Designziele

- C/C++/Java ähnliche komponenten-orientierte Sprache
- Garbage Collection
- Exceptions
- Typsicherheit
- Interoperabilität via CLR

ILDASM



Intermediate Language



ILDASM Symbols

Symbol	Meaning
	More info
	Namespace
	Class
	Value type
	Interface
	Method
	Static method
	Field
	Static field
	Event
	Property

CSC.exe (C# Compiler)

- **/out:<file>** Output file name (derived from first source file if not specified)
- **/target:exe** Build a console executable (/t:exe)
- **/target:winexe** Build a windows executable
- **/target:library** Build a library
- **/target:module** Build a module that can be added to another assembly
- **/reference:<files>** Reference metadata from the specified assembly files (/r)
- **/doc:<file>** Generate XML documentation file

Zwei Arten von Typen

	Value (struct)	Reference (class)
Variable enthält	Wert	Referenz
Speicher	Stack	Heap
Initialisiert mit	Alles 0	Konstante: `null`
Zuweisung	kopiert Wert	kopiert Referenz

Klassen (`class`)

- Einfachvererbung (single inheritance)
- Implementierung von beliebig vielen Interfaces
- Elemente einer Klasse
  - Konstanten, Felder, Methoden, Operatoren, Konstruktoren, Destruktoren
  - Properties, Indexer, Events
  - Verschachtelte Typen
  - Statische Elemente (`static`)
- Zugriffsschutz
  - public, protected, internal, private

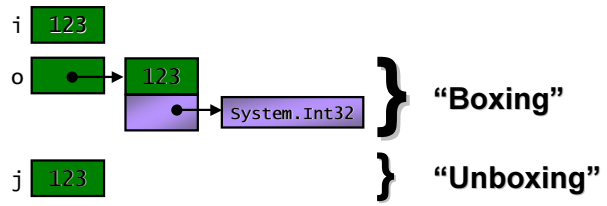
Strukturen (`struct`)

- Sind immer “value types“
- Ideal für “kleine” Objekte
  - Keine Allokierung auf dem Heap
    - Weniger Arbeit für den Garbage Collector
    - Effizientere Speicherbenutzung
  - Benutzer können “primitive” Typen selbst erzeugen
  - Operator Overloading, Conversion Operators
- .NET Framework nutzt diese auch
  - `int, float, double, ...` sind alles `structs`

Boxing und Unboxing

- Jeder Datentyp kann als Objekt gespeichert oder übergeben werden

```
int    i = 123;
object o = i;
int    j = (int)o;
```



Deklaration einer Klasse

```
public class MyClassName : BaseImpl, Itf1, Itf2
{
  // member definitions go here
}
```

Name of base type

List of supported interfaces

Access Modifiers

	C#	VB.NET	Meaning	Java
Type	`public`	`Public`	Type is visible everywhere	`public`
	`internal`	`Private`	Type is only visible inside of assembly	**~ package**
Member	`public`	`Public*`	Member is visible everywhere	`public`
	`internal`	`Friend`	Member is only visible inside of assembly	-
	`protected`	`Protected`	Member is only visible inside of declaring type and its subtypes	`protected`
	`protected internal`	`Protected Friend`	Member is only visible inside its subtypes or other types inside of assembly	-
	`private`	`Private*`	Member is only visible inside of declaring type	`private`

Basistypen

- **Jeder Typ hat maximal einen Basistypen**
  - **System.Object und interfaces haben keinen Basistypen**
  - **System.ValueType für structs**
  - **System.Enum für enums**
  - **System.Array für arrays**
  - **Default-Basistyp für Klassen ist System.Object**
  - **`sealed` class modifier verhindert Vererbung**
  - **`abstract` class modifier verlangt Vererbung**

Interfaces

- **Enthalten Methoden, Properties, Indexer, und Events**
- **Explizite Implementierung möglich**
  - **Löst Interface Probleme bei Namenskollisionen**
  - **Ausblenden der Implementierung vor dem Benutzer**

```
interface IDataBound
{
    void Bind(IDataBinder binder);
}

class EditBox: Control, IDataBound
{
   void IDataBound.Bind(IDataBinder binder) {...}
}
```

Konstruktoren

- Methoden zur Initialisierung von Objekten
  - Werden durch CLR ausgeführt
  - Statische-Konstruktoren (`.cctor`) werden einmal pro Typ bei erstmaligen Zugriff auf Typ ausgeführt
  - Instanz-Konstruktoren (`.ctor`) werden einmal pro Instanz bei Erzeugung des Objekts ausgeführt
  - Instanz-Konstruktoren können Parameter haben
  - Instanz-Konstruktoren können sich gegenseitig aufrufen (`this(args)` syntax)
  - Code für Feld-Initialsierung wird hinzugefügt

Initialisierung

```
public class Patient {
  public double age = a();
  public int children = b();
  public static int patientCount = c();
  static Patient() {
    d();
  }
  public Patient() {
    e();
  }
}
void Main() {
  Patient p1 = new Patient();
  // call order: c(), d(), a(), b() and e()
}
```


Virtuelle Methoden

- Virtuelle Methoden können in abgeleiteten Klassen überschrieben werden
  - Überschriebene Methoden werden immer zur Laufzeit aufgrund des vorliegenden Objekttyps gewählt
  - Deklarierter Statischer Typ spielt keine Rolle
  - Methode der Basisklasse muß als `virtual` oder `abstract` deklariert werden
  - Abgeleitete Klasse muß explizit `override` Schlüsselwort verwenden

Vererbung

- Methoden sind standardmäßig NICHT virtual
- Eine Methode kann nur mit „override“ überschrieben werden

```
class B {
   public virtual void foo() {}
}

class D : B {
   public override void foo() {}
}
```

Vererbung

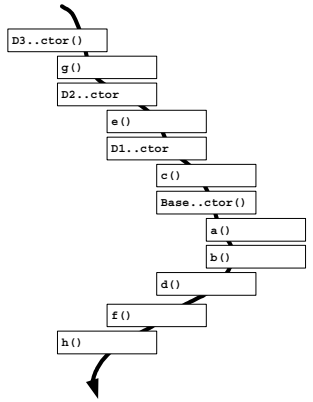
- Methoden sind standardmäßig NICHT virtual
- Überschreiben einer nicht-virtual Methode → Compilerfehler!
  Außer man verwendet „new“

```
class N : D {
   public new void foo() {}
}

N n = new N();
n.foo();        // call N's foo
((D)n).foo();   // call D's foo
((B)n).foo();   // call D's foo
```


Demo

Beispiel 9: Virtuelle Methoden

Vererbung und Initialisierung




```
public class Base {
  public int x = a();
  public Base() { b(); }
}
```

```
public class D1 : Base {
  public int y = c();
  public D1() { d(); }
}
```

```
public class D2 : D1 {
  public int z = e();
  public D2() { f(); }
}
```

```
public class D3 : D2 {
  public int w = g();
  public D3() { h(); }
}
```


Interfaces und Basistypen

```
public interface IVehicle {
  void Start(); void Stop(); void Turn();
}
public class Base : IVehicle {
  void IVehicle.Start() { a(); }
  public void Stop() { b(); }
  public virtual void Turn() { c(); }
}
public class Derived1 : Base {
// illegal - cannot override non-existant method
  public override void Start() { d(); }
// illegal - Base.Stop not virtual
  public override void Stop() { e(); }
// legal, replaces Base.Turn + IVehicle.Turn
  public override void Turn() { f(); }
}
public class Derived2 : Base, IVehicle {
// legal - we redeclared IVehicle support
  void IVehicle.Start() { g(); }
// legal - we redeclared IVehicle support
  public void Stop() { h(); }
// legal - replaces IVehicle.Turn (but not Base.Turn)
  public void Turn() { i(); }
}
```

C# Programmstruktur

- Namespaces
  - Enthalten Typdefinitionen und Namespaces
- Typdefinitionen
  - Klassen, Strukturen, Interfaces, ...
- Elemente von Typen
  - Konstanten, Felder, Methoden, Properties, Indexer, Events, Operatoren, Konstruktoren, Destruktoren
- Organisation der Dateien
  - Datei kann beliebig viele Klassen enthalten

C# Program Struktur (1)

```
using namespace;
namespace qualifier1.qualifier2
{
   public class Classname
   {
      VarType varname;
      public void Methodname(type paramname) {
         paramname = new Classname(params);
      }
   }
}
```


C# Program Struktur (2)

```
public returntype Methodname() {
   if (varname == null) throw new
      SomeException();
   Vartype resultvar = somethingElse;
   return resultvar;
}
```


Statements und Expr.

- If, while, do benötigen eine boolsche Bedingung
  - `if (value)` → Fehler
  - `if (value==true)` → ok
- Switch Statement
  - "break", "goto case" or "goto default" notwendig

```
switch (expression) {
   case constant-expression:
      statement jump-statement
   [default: statement jump-statement]
}
```

Demo

Beispiel 10: Switch in C#

Foreach Statement

- **Iteration von Arrays**

```
foreach (string s in args) Console.WriteLine(s);
```

- **Iteration von selbstdefinierten Collections**

```
foreach (Customer c in customers.OrderBy("name"))
{
    if (c.Orders.Count != 0)
    {
        ...
    }
}
```


Parameterübergabe: IN

- **in Parameter: Entspricht einer Übergabe „ByVal“. Es wird der Wert an die Methode übergeben**

```
static void Foo(int p) {++p;}
static void Main()
{
   int x = 8;
   Foo(x);  // Kopie von x wird übergeben
   Console.WriteLine(x);  // x = 8
}
```


Parameterübergabe: REF

- **ref Parameter: Man übergibt die Referenz an die Methode**

```
static void Foo(ref int p) {++p;}
static void Main()
{
   int x = 8;
   // Referenz von x wird übergeben
   Foo(ref x);
   Console.WriteLine(x);  // x = 9
}
```

Parameterübergabe: OUT

- **out Parameter: Erhält einen Wert von Methode zurück (Variable muss vorher nicht initialisiert werden)**

```
static void Foo(out int p) {p = 3;}
static void Main()
{
   int x;
// x ist bei Übergabe nicht initialisiert
   Foo(out x);
   Console.WriteLine(x);  // x = 3
}
```


Properties

- **Felder mit Zugriffs-Methoden**
- **Properties sind:**
  - **für Read-Only Felder**
  - **für Validierung**
  - **für berechnete oder zusammengesetzte Werte**
  - **Ein Ersatz für Felder in Interfaces**

Properties Beispiel

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get {
         return caption;
      }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```


Indexer

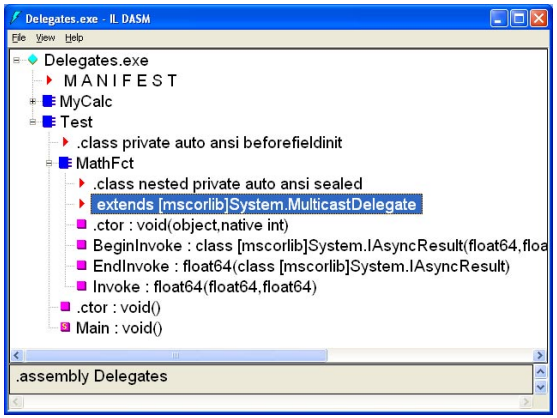
- **Praktische Möglichkeit, Container zu implementieren**
- **Erweitern die Idee der Properties**
- **Erlauben Indexierung von Daten innerhalb des Objekts**
- **Zugriff ist wie bei Arrays**
- **Der Index selbst kann von jedem Datentyp sein**

Indexer Beispiel

```
public class ListBox: Control
{
   private string[] items;

   public string this[int index] {
      get {
         return items[index];
      }
      set {
          items[index] = value;
          Repaint();
      }
   }
}
```

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine  (listBox[0]);
```


ILDASM Delegate / Beispiel 5

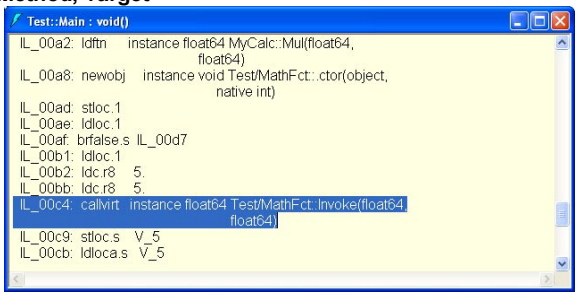


Delegates

- Ersatz für Funktionspointer
- Objektorientiert, type-safe und gesichert
- Delegates sind Objekte
- Eine Instanz eines Delegates kapselt
  - eine Methode und
  - eine Referenz auf eine Instanz (nur wenn Methode nicht statisch)
- Grundlage für Events

MulticastDelegate

- Konstruktor
  - Wird von Kompiler-generierter Klasse aufgerufen
  - protected MulticastDelegate(object target, string method);
- Instance Properties
  - Method, Target



Definition Delegates

```
public delegate void DeathCallback(IPatient p, String msg);
public class Morgue {
  public static void ReceivePatient(IPatient patient,
                                    DeathCallback proc) {
    patient.IsAlive = false;
    proc(patient, "They shipped us the body, so...");
  }
}
```


Delegates nutzen

```
public class FuneralDirector {
  Mortuary mortuary; Pastor pastor;
  public int ComeAndGetIt(IPatient p, string reason)
  {
    mortuary.Embalm(p);
    pastor.ConsoleFamily(reason);
  }
}
void Main() {
  IPatient p = new Patient();
  FuneralDirector fd = new FuneralDirector();
  DeathCallback proc = new DeathCallback(fd.ComeAndGetIt);
  Morgue.ReceivePatient(p, proc);
}
```


EventHandler/EventArgs

- Delegates für Events werden von `System.EventHandler` abgeleitet
  - `System.EventHandler` unterstützt zwei Parameter: `sender, data`
  - `Sender` ist typischerweise ein `System.Object`
  - `Data` ist ein `System.EventArgs` oder ein Subtyp dessen
  - Events ohne Daten sollen mit `System.EventArgs.Empty` aufgerufen werden
  - Anzahl der Event-Klassen wird damit drastisch reduziert

System.EventHandler

```
namespace System {
  public class EventArgs {
    public static readonly Empty;
  }
  public delegate void EventHandler(Object sender,
                                    EventArgs data);
}
```

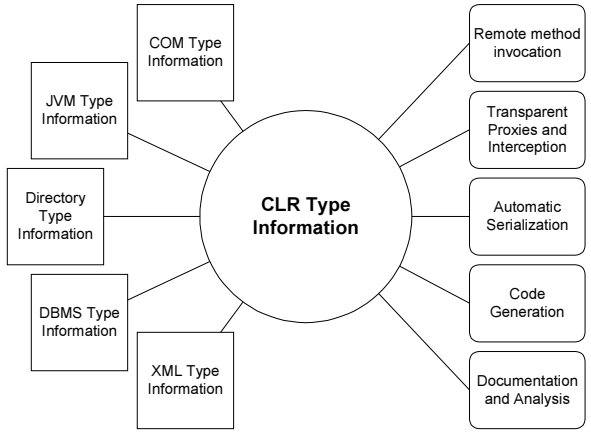
EventArgs

```
public class DeathEventArgs : System.EventArgs {
  IPatient p; string msg;
  public DeathEventArgs(IPatient p, string msg)
  { this.p = p; this.msg = msg; }
  public IPatient Patient { get { return p; } }
  public string   Message { get { return msg; } }
}
public delegate void DeathEventHandler(object sender,
                                       DeathEventArgs data);
public class Doctor : IDoctor {
  public event DeathEventHandler OnPatientDied;
  public void Operate(IPatient p) {
    p.Operate();
    if (p.Dead && OnPatientDied != null)
      OnPatientDied(this,
                    new DeathEventArgs(p, "last words..."));
  }
}
```

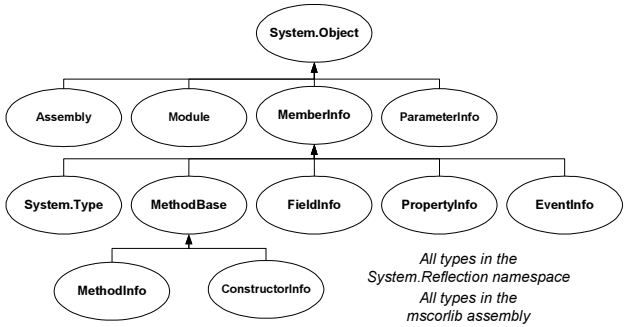

Reflection

- Alle Typen der CLR besitzen Information über sich selbst
  - Typinformationen können aus DLL-Kopf gelesen werden
  - CLR bietet Klassen für Lese- und Schreibzugriff auf Typinformationen
  - Programme können geschrieben werden, die Programme als Input akzeptieren
  - Typsystemen können leicht abgebildet werden
  - In laufendem Programm können Informationen über Typen und Objekte leicht abgefragt werden

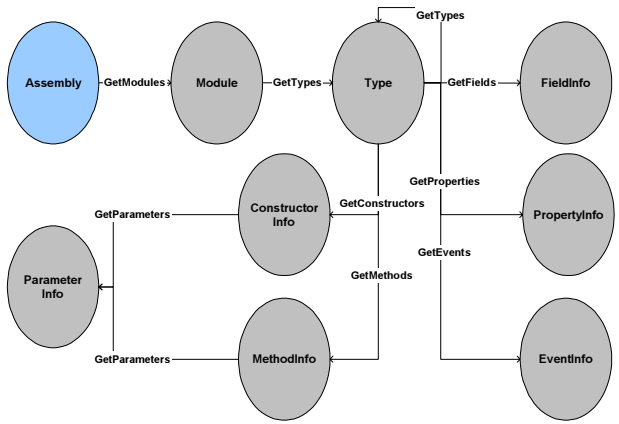
Reflection



Reflection-Klassen



Reflection Objekte und Methoden



MethodInfo

```
static double CallAdd(Object target, double x, double y) {
  Type type = target.GetType();
  MethodInfo method = type.GetMethod("Add");
  Object[] args = { x, y };
  Object result = method.Invoke(target, args);
  return (double)result;
}
```


Demo

Beispiel 7 (Wiederholung): Reflection

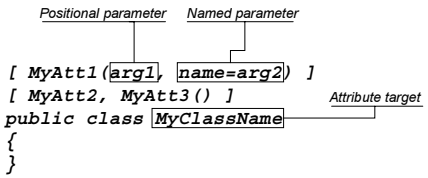
Attribute

- Runtime / Design-Time Informationen für Typen und deren Elemente
- Beispiele
  - URL für Dokumentation einer Klasse
  - Informationen für COM marshalling
  - Wie wird in XML persistiert
- Bekannte Methoden sind entkoppelt
  - Neue Schlüsselwörter oder `pragma`
  - Zusätzliche Dateien, z.B.: .IDL, .DEF
- C# Lösung: Attribute

Attribute

- Für Typen und deren Elemente
- Zugriff zur Laufzeit über “reflection“
- Vollständig erweiterbar
  - Ein Attribut ist eine Klasse, die von System.Attribute abgeleitet wurde
- Wird im .NET Framework oft benutzt
  - XML, Web Services, Security, Serialization, Component Model, COM und P/Invoke Interop …

Attribute definieren



Beispiel Attribute

```
using System;

public class FlakeyCodeAttribute : Attribute {
  public bool Flakey = true;
  public int FlakeFactor = 0;
  public FlakeyCodeAttribute() {}
  public FlakeyCodeAttribute(bool flakey) {
    this.Flakey = flakey;
  }
}
```


Attribut deklarieren

```
[ FlakeyCode ]
class MyClass {
  [ FlakeyCode(FlakeFactor = 4) ]
  void f() {
    Object obj = null;
    obj.ToString();
  }
}
```

Attribute lesen

```
namespace System.Reflection {

// supported by MemberInfo, ParameterInfo, Assembly and Module
  public interface ICustomAttributeProvider {

// test for presence/absence of attribute of type attType
    bool IsDefined(System.Type attType, bool inherited);

// return all attributes that are compatible with attType
    object[] GetCustomAttributes(System.Type attType,
                                 bool inherited);

// return all attributes irrespective of type
    object[] GetCustomAttributes(bool inherited);
  }
}
```


Attributinformation nutzen

```
static bool IsFlakey(Object target, out int factor) {
  factor = -1;
  Type targettype = target.GetType();
  Type attrtype = typeof(FlakeyCodeAttribute);
  if (!targettype.IsDefined(attrtype))
    return false;
  object[] atts = targettype.GetCustomAttributes(attrtype,
                                                 false);
  FlakeyCodeAttribute attr = (FlakeyCodeAttribute)atts[0];
  factor = attr.FlakeFactor;
  return attr.Flakey;
}
```


Beispiele Attribute

- Attribute sind Klassen
  - Abgeleitet von System.Attribute
  - Klassenfunktionalität = Attributfunktionalität

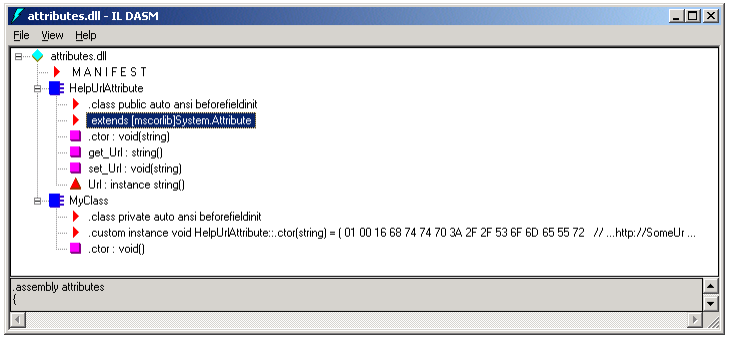
```
public class HelpUrlAttribute : System.Attribute
{
    public HelpUrlAttribute(string url) { … }

    public string Url   { get {…} }
    public string Tag  { get {…}  set {…}   }
}

[HelpUrl("http://SomeUrl/MyClass")]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass", Tag="ctor")]
class MyClass  {}
```


Attribut verwenden

Wenn der Compiler ein Attribut sieht

1. ruft er den Konstruktor auf und übergibt die Argumente
2. falls weitere Parameter existieren, setzt er das Property auf den entsprechenden Wert
3. speichert die Parameter in den Metadaten

ILDASM Attribute



Attribute abfragen

- **Mittels Reflection können Attribute abgefragt werden**

```
Type type = typeof(MyClass);

foreach(object attr in
  type.GetCustomAttributes() )
{
    if ( attr is HelpUrlAttribute )
    {
        HelpUrlAttribute ha =
  (HelpUrlAttribute) attr;

        myBrowser.Navigate( ha.Url );
    }
}
```


C# und Pointer

- **C# unterstützt**
  - **Eingebauter Typ:** `String`
  - **Benutzerdefinierte Referenztypen**
  - **Große Auswahl an Collection-Klassen**
  - **Referenz- und Ausgabeparameter (**`out` , `ref`**)**
- **99% der Pointer werden nicht mehr benötigt**
- **Dennoch sind Pointer verfügbar, wenn Programmcode mit** `unsafe` **markiert ist**

Beispiel unsafe Code

```
class FileStream: Stream
{
   int handle;

   public unsafe int Read(byte[] buffer, int index, int
   count)
   {
      int n = 0;
      fixed (byte* p = buffer)
      {
         ReadFile(handle, p + index, count, &n, null);
      }
      return n;
   }

   [dllimport("kernel32", SetLastError=true)]
   static extern unsafe bool ReadFile(int hFile,
      void* lpBuffer, int nBytesToRead,
      int* nBytesRead, Overlapped* lpOverlapped);
}
```

Kommentare

- Spezialkommentare (ähnlich javadoc)
- Beispiel

```
/// ... comment ...
class C {
/// ... comment ...
public int f;
/// ... comment ...
public void foo() {...}
}
```

- Übersetzung

```
csc /doc:MyFile.xml MyFile.cs
```


Beispiel für Quellprogramm

```
/// <summary> A counter for accumulating values and
   computing the mean value.</ summary>
class Counter {
/// <summary> The accumulated values</ summary>
private int value;
/// <summary> The number of added values</ summary>
public int n;
/// <summary> Adds a value to the counter</ summary>
/// <param name=" x"> The value to be added</ param>
public void Add( int x) {
value += x; n++;
}
```


Daraus erzeugte XML- Datei

```
<? xml version=" 1.0"?>
<doc>
   <assembly>
        <name> MyFile</ name>
   </ assembly>
   <members>
        <member name=" T: Counter">
        <summary> A counter for accumulating values and computing the mean
   value.</ summary>
        </ member>
        <member name=" F: Counter. value">
        <summary> The accumulated values</ summary>
        </ member>
        <member name=" F: Counter. n">
        <summary> The number of added values</ summary>
        </ member>
        <member name=" M: Counter. Add( System. Int32)">
        <summary> Adds a value to the counter</ summary>
        <param name=" x"> The value to be added</ param>
        </ member>
   </ members>
</ doc>
```


XML- Tags

- <summary> </ summary>
- <remarks> Ausführliche Beschreibung </ remarks>
- <param name=" ParamName"> Beschreibung </ param>
- <returns> Beschreibung </ returns>

- <exception [cref=" ExceptionType"]> Beschreibung der Exception </ exception>
- <example> z. B. Aufrufbeispiel </ example>
- <code> Beliebiger Code </ code>
- <see cref=" ProgramElement"> Querverweis </ see>
- <paramref name=" ParamName"> Parameter </ paramref>

- Beliebige eigene XML- Tags, z. B. <author>, <version>, ...

C# vs. Java Gemeinsamkeiten

- Zwischencode-Generierung
- Garbage Collection
- Mächtige Reflection
- Klassenhierarchie mit einer Root-Klasse (object)
- Interfaces, Mehrfachvererbung von Interfaces, Einfachvererbung von Klassen
- Inner Classes
- Keine globalen Variablen, alles gehört zu Klassen
- jagged Arrays
- Alle Variablen werden initialisiert
- try – catch - finally

C# vs. Java Unterschiede

- Java: interpretiert, C#: JIT-Compiler
- Field modifiers, Parameter modifers, Properties, Indexers
- Delegates, Events, Pointer
- Operatoren-Overloading
- Typen: Primitives, Structs, Enums, Unsigned Types, Decimal
- Alle Methoden virtuell vs. Virtuelle Methoden müssen explizit deklariert werden
- Attribute
- Goto
- Destruktoren

Zukunft von C#

- Generics
- Anonyme Methoden
- Iteratoren

Infos:

- http://www.devx.com/cplus/Article/9937
- http://gotdotnet.com/team/csharp/learn/Future/default.aspx
- http://www.hardwaregeeks.com/board/printthread.php?threadid=2736
- http://www.microsoft.com/presspass/press/2002/Nov02/11-08OOPSLAPR.asp

Fragen?