C#

typsicher

einfach

modern

objektorientiert

Einführung

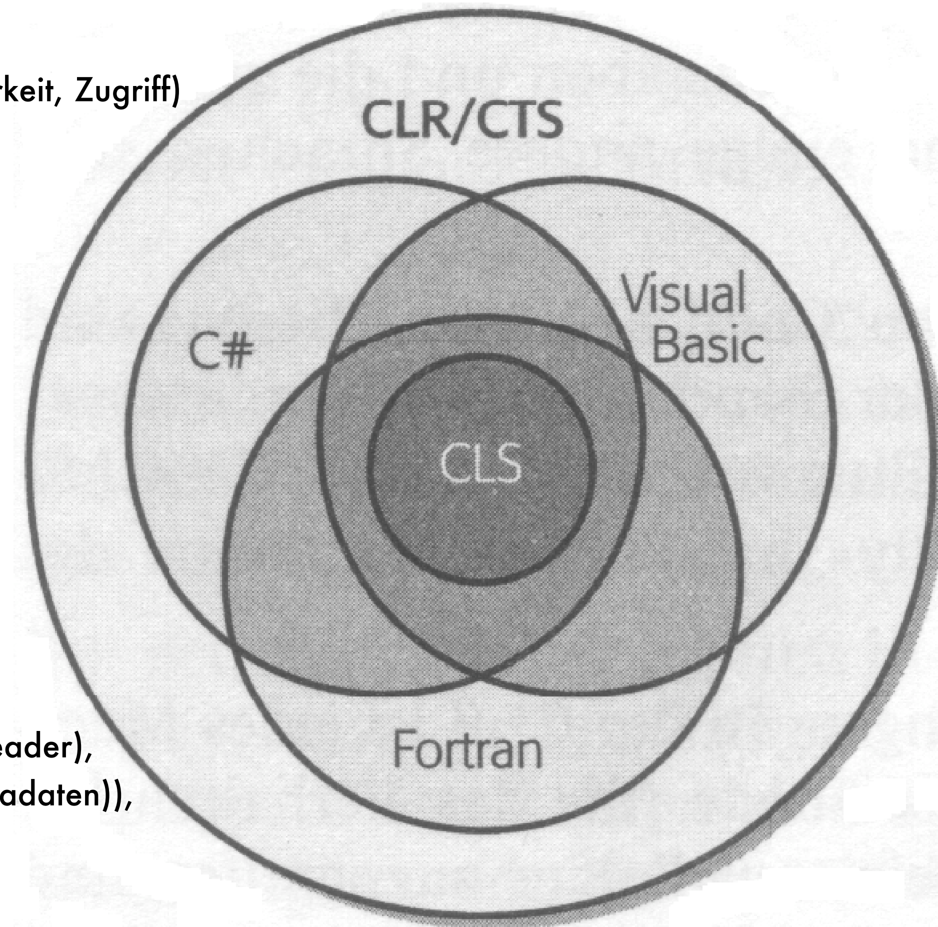
- **Veröffentlichung:** Juni 2000
- **C#-Creators:** Anders Hejlsberg, Scott Wiltamuth, Peter Golde
- **Standardisierung:**
  - Dezember 2001, ECMA-334 (European Computer Manufacturer's Association)
  - April 2003, ISO/IEC 23270
- **Schlüsselwörter:**

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unsafe
unchecked	ushort	using	virtual	void	while	

Einführung (2)

CommonTypeSystem:

- Spezifikation für Definition von Typen (Member, Sichtbarkeit, Zugriff)
- Regeln für Vererbung, virtuelle Funktionen, usw.



CommonLanguageSpecification:

- minimale Anforderung an Sprachen, um CLR-kompatibel zu sein
- kleinster gemeinsamer Nenner aller Sprachen, um Interoperabilität zu gewährleisten

Verwaltetes Modul:

- Programm, das durch die CLR ausgeführt wird
- Enthält: Header für übertragbare, ausführbare Windows-Dateien (PE-Header), CLR-Dateiheader (Informationen über Modul (Ort der IL u. Metadaten)), Metadaten, IL-Code

Metadaten:

- Beschreibung der gesamten inneren Struktur (Typen, Methoden, Felder) des Moduls u. seiner äusseren Abhängigkeiten (welche Typen werden referenziert)

Assemblies:

- grundlegende Einheit für Sicherheit, Versionsverwaltung und Wiederverwendbarkeit
- besteht Ressourcendateien u. aus verwalteten Modulen, ein Modul enthält das Manifest (Bezeichnung der Assembly, Versionsnummer, Liste der anderen Dateien, Liste der Datentypen der anderen Dateien)

Typen - Primitive Typen

Primitiver Typ in C#	FCL-Typ	CLS-kompatibel	Werttyp	Beschreibung
`sbyte`	`System.SByte`	Nein	Ja	8-Bit-Wert mit Vorzeichen
`byte`	`System.Byte`	Ja	Ja	8-Bit-Wert ohne Vorzeichen
`short`	`System.Int16`	Ja	Ja	16-Bit-Wert mit Vorzeichen
`ushort`	`System.UInt16`	Nein	Ja	16-Bit-Wert ohne Vorzeichen
`int`	`System.Int32`	Ja	Ja	32-Bit-Wert mit Vorzeichen
`uint`	`System.UInt32`	Nein	Ja	32-Bit-Wert ohne Vorzeichen
`long`	`System.Int64`	Ja	Ja	64-Bit-Wert mit Vorzeichen
`ulong`	`System.UInt64`	Nein	Ja	64-Bit-Wert ohne Vorzeichen
`char`	`System.Char`	Ja	Ja	16-Bit-Unicodezeichen
`float`	`System.Single`	Ja	Ja	IEEE-32-Bit-Gleitkommazahl
`double`	`System.Double`	Ja	Ja	IEEE-64-Bit-Gleitkommazahl
`decimal`	`System.Decimal`	Ja	Ja	128-Bit-Gleitkommazahl
`bool`	`System.Boolean`	Ja	Ja	Wert, der entweder True oder False ist
`object`	`System.Object`	Ja	Nein	Basistyp aller Typen
`string`	`System.String`	Ja	Nein	Array mit Zeichen

Tabelle: Primitive Typen und ihre Äquivalente

- Abbildung auf Typen der FCL, dadurch einheitliche Behandlung aller Typen

Typen (2)

- alle Typen von `System.Object` abgeleitet

Methode	Beschreibung
`public virtual bool` **`Equals`**`(object)`	Vergleich von zwei Objekten über ihre Referenzen, d.h. Gleichheit besteht, wenn beide auf dasselbe Objekt zeigen.
`public virtual int` **`GetHashCode`**`()`	Hashwert für den Wert des Objekts (eindeutige Zahl innerhalb der AppDomain).
`public Type` **`GetType`**`()`	Instanz eines von `Type` abgeleiteten Objekts, das im Zusammenhang mit den Reflektionsklassen dazu eingesetzt werden kann, um Informationen über den Typ zu ermitteln.
`public virtual string` **`ToString`**`()`	Stringdarstellung des Objektwertes (bei `object` allerdings vollständiger Name des Typs)
`protected object` **`MemberwiseClone`**`()`	Anlegen einer neuen Instanz des Typs mit flacher Kopie.
`protected virtual void` **`Finalize`**`()`	Wird aufgerufen, wenn der Garbage Collector entschieden hat, dass dieses Objekt gelöscht werden kann. In C# werden Finalizer mit Hilfe eines Destruktors ausgedrückt: `~Object();`

Tabelle: Methoden von `System.Object`

- alle Typen implizit nach `object` konvertierbar

```
object o1 = new A(); // Upcast (in beliebige Basisklasse möglich)
A a = (A)o1; // Downcast
class B : A { ... }
B b = (B)o1; // Runtime-Fehler: InvalidCastException
```

Typen - Werttypen

- Erzeugung auf dem Stack
- Boxing ermöglicht Werttypen im Heap
- Deklaration durch Schlüsselwort `struct` oder `enum`
- implizite Ableitung von `System.ValueType` (oder `System.Enum`)
  - erbt von `object`
  - Methoden `Equals()` und `GetHashCode()` überschrieben
- Definition von Standardkonstruktor verboten

  Folge: Inline-Initialisierung nicht erlaubt (`int x` ~~`= 5`~~`;`)

```
struct WertTyp
{
    int a;
    double b;
}_ // kein Semikolon nach struct- und class-Deklarationen

WertTyp t1;
WertTyp t2 = t1; // Compile-Fehler: t1 noch nicht initialisiert
t1 = new WertTyp(); // implizite Initialisierung mit 0/null
WertTyp t2 = t1; // Kopie der Felder a,b
```

Typen - Werttypen (2)

- Initialisierungskonstruktoren erlaubt
  - Pflicht ist mindestens ein Parameter
  - alle Member müssen initialisiert werden
  - expliziter Aufruf erforderlich
- Automatische Initialisierung bei Einbettung im Verweistyp
- explizite Ableitung nur von Schnittstellen möglich

  Werttyp ohne Methodentabelle und Zeiger auf diese

```
struct Point : ICloneable
{
    int x, y; // beide Typfelder sind private
    public Point(int wert)
    {
        x = wert;
        y = x + 2; // ohne Initialisierung von y Compile-Fehler
    }
    public object Clone() { return MemberwiseClone(); }
}
Point t1 = new Point(10);
```

- Definition von Destruktoren verboten

Typen - Boxing/Unboxing von Werttypen

- Boxing:

  Allokation des Werttyps auf dem Heap mit Overheadmembern (Zeiger auf Methodentabelle, SyncBlockIndex)

- Compiler generiert automatisch Boxing/Unboxing-Code
- Auftreten von Boxing:

  - Upcast nach `object`

    z.B. bei Methodenparametern (benötigen meist `object`-Verweistyp)

  - Konvertierung in Zeiger einer implementierten Schnittstelle
  - Aufruf von geerbten Methoden

- Achtung: Stack-Werttyp `!=` Boxed Werttyp
- Unboxing:

  Rückgabe der Adresse von geboxtem Wert im Heap

- C#: Unboxing + Kopieren des Wertes aus Adresse
- Achtung: Konvertierung in Boxed Typ notwendig

Typen - Boxing/Unboxing von Werttypen (2)

- **Beispiele:**

```
ArrayList a = new ArrayList();
Point p1 = new Point(2);
a.Add(p1); // p1 boxen und Verweis zum Array hinzufügen

Point p2 = (Point)a[0]; // Unboxing + Kopieren der geboxten Felder

int x = 5;
object o = x; // Boxing von x, o verweist auf geboxtes Objekt im Heap
x = 20; // ändert nur Wert auf dem Stack
short y = (short)o; // InvalidCastException
y = (short)(int)o; // Okay, y=5

p1.ToString(); // Boxing erforderlich, da Zugriff auf geerbte Methode
x.ToString(); // Int32 überschreibt ToString(), daher kein Boxing

public override string ToString() // zusätzliche Point-Membermethode
{
   return x.ToString() + y.ToString(); // kein Boxing bei p1.ToString()
} // Compiler benötigt keinen Zeiger auf M.-Tabelle, sondern direkter Aufruf
```

Typen - Boxing/Unboxing von Werttypen (3)

- Verständnisfrage:

```
public object Clone()
{
    return MemberwiseClone();
}
```

„Tritt im folgenden Code Boxing auf?“

```
    p2 = (Point)p1.Clone();
```

Typen - Boxing/Unboxing von Werttypen (3)

- Verständnisfrage:

```
public object Clone()
{
    return MemberwiseClone();
}
```

„Tritt im folgenden Code Boxing auf?“

```
p2 = (Point)p1.Clone();

// MemberwiseClone() ist geerbte Methode, daher Boxing
// danach Unboxing + Kopieren des Wertes in p2
```

```
ICloneable ip = p1; // Boxing, wie Upcast nach object
object o = ip.Clone(); // Erzeugen eines neuen Objektes auf dem Heap
                       // Verweis in o speichern

p2 = (Point)o; // Unboxing + Kopieren der Werte nach p2
```

Typen - Verweistypen

- Anlegen des Speichers auf dem Heap
- Deklaration mit Schlüsselwort `class`

```
class VerweisTyp // Standardzugriffsmodifizierer bei Typdeklaration auf
{                // Namespaceebene: internal
    public int a;
    public int b;
}
```

- Erzeugung über Operator `new`
  - Speicherreservierung für Instanz
  - Initialisierung der Verwaltungsfelder
    (Zeiger auf Methodentabelle, SyncBlockIndex (Threadsynchronisation))
  - Initialisierung der Felder mit 0/null
    Felder sind also immer initialisiert, auch wenn dies im Konstruktor nicht geschieht
  - Aufruf des Instanzkonstruktors

```
VerweisTyp a = new VerweisTyp(); // Aufruf des vom Compiler erzeugten
                                 // Standardkonstruktors
```

- Deklaration mehrerer Konstruktoren möglich
- Inline-Initialisierung erlaubt

Typen - Verweistypen (2)

```
class Point
{
    string name = “P“; // Compiler fügt Initialisierung in jeden
    int x;             // Konstruktor, der nicht einen anderen aufruft, ein
    int y;
    public Point(int a, int b)
    {
        x = a;
        y = b;
    }
    public Point() : this(10, 10) {}
}

Point p1 = new Point(); // Initialisierung mit (“P“, 10, 10)
Point p2 = new Point(5, 5);
Point p3;
string s = p3.ToString(); // Compile-Fehler: p3 nicht zugewiesene Variable
Point p4 = null;
Console.WriteLine(p4.ToString()); // Runtime: NullReferenceException
```

- Konstruktoren auch `protected/private` deklarierbar
  - keine Möglichkeit ausserhalb der Klasse Instanzen zu erzeugen

Typen - Verweistypen (3)

- Aufruf von Basisklassenkonstruktor ist Pflicht

  (Basisklassenkonstruktor parameterlos, dann generiert der Compiler den Aufruf automatisch)

```
class Point3d : Point
{
    int z;
    public Point3d(int c)
    {
        z = c; // x,y = 10
    }

    public Point3d(int a, int b, int c) : base(a, b) // x=a, y=b
    {               // nur die direkt darüberliegende Basisklasse beeinflussbar
        z = c;
    }
}
```

- Destruktoren: `~Typname { ... }`

  - ausschließlich Instanzmethode
  - GC ruft Destruktor auf, wenn Referenzen auf Objekt = 0
  - Umsetzung in Deklaration einer `Finalize`-Methode

```
protected override void Finalize() { ... }
```

Typen - Verweistypen (4)

- Typkonstruktoren:
  - zur Initialisierung des Typs (statische Felder)
  - auch für `struct` anwendbar
  - nur ein Typkonstruktor möglich
  - muss parameterlos sein
  - implizit `private`
  - Deklaration mit Schlüsselwort `static`
  - Aufruf vor erstem Zugriff auf Typ (CLR-Entscheidung)
  - Aufruf ein einziges Mal in Lebensdauer des Prozesses
  - Generierung eines Typkonstruktors bei Inline-Initialisierung von statischen Typmembern

```
class Kreis
{
    static double PI;
    static string name = "K"; // Initialisierung wird Typkons. hinzugefügt
    _ static Kreis() { PI = 3.141573; } // Zugriffsmodifizierer verboten!
}
```

Typen - Zeigertypen

- Syntax wie in C
  - „&" Adressoperator
  - „*" Dereferenzierungsoperator
  - „->" Zeiger-auf-Member, syntaktische Kurzform
- Verwendung nur in `unsafe`-Block möglich
  - Compiler benötigt zusätzlich Optionsschalter `/unsafe`
  - CLR führt keine Sicherheitsprüfungen mehr aus
    (Arraybereichsgrenzenprüfung, Zugriff auf unerlaubten Speicherbereich)

```
Point pt = new Point(3, 3); // struct-Deklaration von Point verwendet
unsafe
{
    Point *p = &pt;  // Zeiger nur auf Werttypen (eigene Werttypen dürfen
    p->x = 5;        // nur Werttypen als Member enthalten) anwendbar!
    p[1].x = 5; // überschreiben fremden Speicherbereichs
    p[1].y = 5; // kann gut gehen, muss aber nicht
    Console.WriteLine(p->ToString()); // Ausgabe z.B. (5,3)
    Console.WriteLine(p[1].ToString());
} // Beispiel löste ExecutionEngineException aus
```

Typen - Zeigertypen (2)

- Zeiger auch auf Arrays von Werttypen möglich
  - Arrays von Werttypen sind Verweistypen (Verwaltung des GC)
  - müssen deshalb durch Schlüsselwort `fixed` fixiert werden
  - durch Fixierung darf Garbage Collector Speicher nicht umsetzen

```
Point[] pt = new Point[10]; // pt ist Verweistyp, Array enthält Werttypen
unsafe
{
    fixed(Point *p = pt) // Arrayname enthält Zeiger auf erstes Element
    {
        Point *pi = p; // fixed = “Point *const p = pt;“ (konstanter Zeiger)
        for (int i = 0; i < pt.Length; i++)
            Console.WriteLine(pi[i].ToString());
    }
}
```

- Möglichkeit Speicher auf dem Stack anzufordern

  Zeigerarithmetik + Schlüsselwort `stackalloc`

```
Point *p = stackalloc Point[10];
for (int i = 0; i < 10; i++)
    Console.WriteLine(“{0}, {1}“, p[i].x, p[i].y);
```

Typmember - Übersicht & Zugriffsmodifizierer

- Typ kann beliebig viele folgender Member definieren:
  - Konstanten, Felder, Enumerationsinstanzen, Properties
  - Konstruktoren, Methoden (Instanz, Typ)
  - Operatorüberladungen (nicht Teil der CLS)
  - Konvertierungsoperatoren (nicht Teil der CLS)
  - Events
  - Typen (Delegates, Enumerationstypen, verschachtelte Klassen, Schnittstellen)
- Compiler generiert für jede Art von Member Metadaten
- Zugriffsmodifizierer für alle Member:

Schlüsselwort in C#	Beschreibung
`public`	Alle Methoden aus verschiedenen Assemblies dürfen auf das Member zugreifen
`protected`	Nur Methoden innerhalb des Typs oder eines davon abgeleiteten Typs in einer beliebigen Assembly dürfen auf das Member zugreifen
`private`	Nur Methoden innerhalb des Typs dürfen auf das Member zugreifen
`internal`	Nur Methoden aus derselben Assembly dürfen auf das Member zugreifen
`protected internal`	Nur Methoden innerhalb des Typs oder eines davon abgeleiteten Typs oder Methoden aus derselben Assembly dürfen auf das Member zugreifen

Tabelle: Zugriffmodifizierer

Typmember - Konstanten

- Definition nur für primitive Typen möglich
- Teil des Typs, nicht der Instanz (implizit statisch)
- Compiler fügt Konstante direkt in den Code ein
- Speicherung in Metadaten des Moduls

```
class Kreis
{
    public const double PI = 3.1415729;
    public int radius;
    public Kreis(int r) { radius = r; }
}

Kreis k = new Kreis(5);
double umfang = 2 * Kreis.PI * k.radius;
```

Typmember - Felder

- Typfeld (statisch) o. Instanzfeld
  (Felder enthalten Instanzen von Werttypen oder Verweise von Verweistypen)
- dynamische Speicherzuweisung von Feldern:
  - Typfelder: Laden des Typs in Appdomain
    (geschieht wenn JIT-Compiler erstes Mal eine Methode übersetzt, die den Typ benutzt)
  - Instanzfelder: Anlegen einer Instanz des Typs
- Unterstützung von flüchtigen Feldern (`volatile`)
- Unterstützung von schreibgeschützten Feldern (`readonly`)
  Initialisierung im Konstruktor oder inline

```
class Kreis
{
    public static readonly double PI = 3.1415729;
    public readonly int radius;
    public Kreis(int r) { radius = r; }
}

Kreis k = new Kreis(10);
double umfang = 2 * Kreis.PI * k.radius;
k.radius = 5; // Compile-Fehler
```

Typmember - Enumerationen & Bitflags

```
enum Color { Rot, Gelb, Blau, Grün = 10}
```

- „richtiger" Typ:
  - direkt von `System.Enum` abgeleitet
  - Werttyp (darf aber nur Symbole, wie oben gezeigt, definieren)
  - zugrundeliegender Typ, der Werte speichert, änderbar (Standard: `Int32`)

```
enum Color : byte { Rot, Gelb, Blau, Grün}
```

- Instanzbildung:

```
Color c = Color.Blau; // auf dem Stack liegt Wert 2
Console.WriteLine(c.ToString()); // „Blau“ (aus Metadaten)
Console.WriteLine(c.ToString(“D“)); // Decimal: „2“
```

- `System.Enum` statische Methoden:

```
Type GetUnderlyingType(Type enumType)
string Format(Type enumType, object value, string format)
Array GetValues(Type enumType)
object Parse(Type enumType, string value, bool ignoreCase)
```

Typmember - Enumerationen & Bitflags (2)

- sinnvoll ausserhalb von Typen zu definieren (FCL i.d.R)
- Bitflags:

```
[Flags]
enum FileAttributes
{
    ReadOnly  = 0x0001,
    Hidden    = 0x0002,
    System    = 0x0004,
    Directory = 0x0010,
    ...
}

FileAttributes fa = FileAttributes.ReadOnly | FileAttributes.Hidden;
Console.WriteLine(fa.ToString()); // „ReadOnly, Hidden“
```

- Flagsattribut steuert Verhalten von
  - `ToString()`
  - `Format()`
  - `Parse()`

Typmember - Properties

- vereinfachen Kapselung von Daten
  (OO-Paradigma: Felder eines Typs niemals für öffentlichen Zugriff freigeben)

```
class Person
{
    private string _name = “Tobias“;
    private int _alter;

    public string Name
    {
        get { return _name; } // schreibgeschützte Eigenschaft
    }

    public int Alter
    {
        get { return _alter; }
        set
        {
            if (value < 0 || value > 100)
                throw new ArgumentOutOfRangeException(“Falsches Alter“);
            _alter = value; // Schlüsselwort value steht für den neuen Wert
        }
    }
}
```

- `value` hat den Typ vom Property-Rückgabewert!

Typmember - Properties (2)

```
p.Name = “Tobias“; // Compile-Fehler: p.Name schreibgeschützt
p.Alter = 23;
p.Alter = “dreiundzwanzig Jahre“; // Compile-Fehler: Alter vom Typ int
Console.WriteLine(“Name: {0} Alter: {1}“, p.Name, p.Alter);
p.Alter = -1; // Runtime-Fehler: ArgumentOutOfRangeException
```

- Compiler generiert entsprechende `get`- u. `set`-Methoden
- einfache Zugriffsmethoden werden inline JIT-kompiliert
- Deklaration von parameterlosen Properties als:
  - statische Eigenschaften
  - Instanzeigenschaften
  - virtuelle Eigenschaften
- parameterbehaftete Properties (C#: Indexer):
  - entsprechen `[ ]`-Operatorüberladung
  - nur auf Instanz anwendbar
  - überladbar
  - Index kann beliebige Anzahl an Parametern umfassen

Typmember - Indexer

```
class Zeugnis
{
    private Hashtable zensuren = new Hashtable();

    public object this[string fach]
    {
        get { return zensuren[fach]; } // Nutzen Indexer vom Hashtable
        set { zensuren[fach] = value; }
    }

    public string this[object fach, object zensur]
    {
        get
        {
            if (zensuren[fach].Equals(zensur))
                return "Zensur stimmt mit Fach überein.";
            else
                return "Zensur stimmt nicht überein.";
        }
    }
}
z["Compilerbau"] = 2; // Werttypen werden geboxt
z["Analysis"] = 2.3;
z["BWL"] = "eine eins";
Console.WriteLine(z["Analysis"]); // Achtung: „2,3"
Console.WriteLine(z["BWL", "eine eins"]); // „Zensur stimmt mit Fach..."
```

Typmember - Methoden

- Definition immer inline (in der Klasse)
  - keine Headerdateien
  - vorherige Deklaration nicht möglich
- Parameter:
  - Standard: Werte oder Verweise „by value“ übergeben
  - 2 Schlüsselwörter: `ref`, `out`
    - Übergabe „by reference“
    - für CLR ist `ref` und `out` dasselbe, dienen in C# zur Überprüfung der Korrektheit
    - `ref`: Parameter muss vor Übergabe initialisiert sein
    - `out`: Parameter wird in Methode initialisiert; Fehler, wenn Methode zuerst liest
    - Benutzung der Schlüsselwörter bei Methodendeklaration **und** -aufruf

```
void SetValue(out int new) { new = 10; }; // autom. Dereferenzierung
int x;
SetValue(out x);

void SetValue(ref int xnew) { xnew = 10; };
int x = 5; // ohne Initialisierung: Compile-Fehler
SetValue(ref x);
```

Typmember - Methoden (2)

```
static void Tausch(ref object a, ref object b)
{
    object t = b;
    b = a;
    a = t;
}

string s1 = "Hallo";

string s2 = "Welt";

Tausch(ref s1, ref s2); // Compile-Fehler!
```

- Übergabeparameter müssen bei `ref`, `out` den erwarteten Typ haben

- variable Anzahl von Parametern:
    - Schlüsselwort `params`
    - nur als letzter Methodenparameter möglich

```
static int Add(params int[] values)
{
    int sum = 0;
    for (int i = 0; i < values.Length; i++)
        sum += values[i]);
    return sum;
}

int summe = Add(1, 2, 3, 4, 5, 6);
```

Typmember - Methoden (3)

- `extern`-Modifizierer:
  - Methode wird extern implementiert, Verwendung bei Dll-Importen

```
[DllImport("User32.dll")]
public static extern int MessageBox(int h, string m, string c, int type)
```

- Überladen von Operatoren:
  - unäre und binäre Operatoren überladbar
  - Operatorenmethoden überladbar
  - immer `public static`

```
public static TimeSpan operator -(DateTime t1, DateTime t2)
public static DateTime operator -(DateTime t1, TimeSpan t2)
```

- Konvertierungsoperatoren:
  - für primitive Typen generiert Compiler Code für Konvertierung
  - eigene Typen können implizite (ohne Cast) und explizite (mit Cast) Konvertierungsoperatoren definieren
  - Verwendung:
    - implizit: kein Genauigkeitsverlust bei Umwandlung
    - explizit: Genauigkeitsverlust; Zieltyp kann nicht alle Quellwerte darstellen

Typmember - Methoden (4)

```
class Rational
{
    public Rational(int zahl) { ... }
    public Rational(double zahl) { ... }
    public int ToInt() { ... }
    public double ToDouble() { ... }

    public static implicit operator Rational(int zahl)
        { return new Rational(zahl); }

    public static implicit operator Rational(double zahl)
        { return new Rational(zahl); }

    public static explicit operator int(Rational r)
        { return r.ToInt(); }

    public static explicit operator double(Rational r)
        { return r.ToDouble(); }
}

Rational r1 = 5; // implizite Konvertierung von int nach Rational
Rational r2 = 2.5; // implizite Konvertierung von double
int z1 = (int)r1; // explizite Konvertierung von Rational nach int
double z2 = (double)r2; // explizite Konvertierung nach double
float z3 = (float)r2; // explizite Konvertierung nach int
                      // dann Konvertierung nach float
```

- Mechanismus, um Callback-Funktionen aufzurufen
- Funktionsweise:

  - Schlüsselwort `delegate` + Name + Signatur der Callback-Funktion

  `delegate void Feedback(object value);` (ausserhalb von Klassen deklarierbar)

  - Compiler generiert intern Klasse (von `MulticastDelegate` abgeleitet)
    - `_target`-Feld (Verweis auf Objekt, Verwendung für Instanzmethodencallbacks)
    - `_methodPtr`-Feld (interner Integer zur Identifikation der aufzurufenden Methode)
    - `_prev`-Feld(Verweis auf ein anderes Delegatobjekt, Benutzung bei Delegatketten)
    - `Invoke`-Methode (ruft Callback-Funktionen auf, besitzt daher selbe Signatur)

  - Callback-Funktionen anmelden:

```
Feedback fb = new Feedback(App.ToConsole); // statische Cb-Fkt.
fb += new Feedback(appobj.ToMsgBox); // Instanz-Cb-Fkt. hinzufügen
Feedback(3); // Aufruf der beiden Callback-Funktionen mit Parameter 3
fb -= new Feedback(App.ToConsole); // Cb-Fkt. aus Delegate entfernen
Feedback(1); // Aufruf von appobj.ToMsgBox
```

    - „+=“ u. „-=“ generiert zu `Delegate.Combine()` bzw. `Delegate.Remove()`
    - Möglichkeit mit `Delegate.GetInvocationList()` Cb-Fkt.-Aufruf zu steuern
    - Möglichkeit des dyn. Erzeugens mit `CreateDelegate()` u. `DynamicInvoke()`

Typmember - Events

- basieren auf Delegates
- Funktionsweise:

  - Schlüsselwort `event` + Delegatename + Eventname

```
public event TestEventHandler Test;
```

  - Namenskonvention des Delegates: `*EventHandler`
  - Prototyp des Delegates:

```
delegate void EventHandler(object sender, EventArgs e);
```

(Standard-Event-Delegate, deklariert im Namespace `System`)

- Delegates können eigene `*EventArgs` (Namenskonvention) definieren (von `System.EventArgs` neue Klasse ableiten)

  - Generierung des Compilers:

```
private TestEventHandler Test = null;

[MethodImpl(MethodImplOptions.Synchronized)]
public void add_Test(TestEventHandler h)
    { Test = (Test)Delegate.Combine(Test, h); }

[MethodImpl(MethodImplOptions.Synchronized)]
public void remove_Test(TestEventHandler h)
    { Test = (Test)Delegate.Remove(Test, h); }
```

- Methodenzugriffsmodifizierer = Modifizierer bei der Deklaration von `event`

Typmember - Events (2)

```
class TestEventArgs : EventArgs
{
    private string _Message;
    public string Message
        { get { return _Message; } }
    public TestEventArgs(string msg) { _Message = msg; }
}

delegate void TestEventHandler(object sender, TestEventArgs e);

class A
{
    public event TestEventHandler Test;
    public void Testen()
    {
        if (Test != null)
            Test(this, new TestEventArgs("Das ist der Test"));
    }
}

public static void TestIt(object sender, TestEventArgs e)
    { Console.WriteLine(e.Message); }

A obj = new A();
obj.Test += new TestEventHandler(TestIt);
obj.Testen(); // Funktion TestIt aufgerufen mit Ausgabe „Das ist ein Test“
```

- Hinzufügen/Löschen der Callback-Funktionen nur über „+=“ „-=“
- Möglichkeit eigene `add_*`, `remove_*` für `event` zu implementieren (ohne Synchronisation)

Schnittstellen

- entsprechen abstrakten Typen mit virtuellen Methoden
  enthalten keine Implementierung
- Member dürfen sein: Methoden, Properties, Events
- Deklaration mit Schlüsselwort `interface`
- Beispiele aus FCL:

```
public interace System.Collections.IEnumerable
    { IEnumerator GetEnumerator(); }

public interface System.Collections.IEnumerator
{
    _ bool MoveNext(); // Zugriffsmodifizierer verboten
    void Reset(); // implizit public virtual
    object Current { get; }
}
```

- Schnittstellen von anderen Schnittstellen ableitbar
- Implementierung von Schnittstellen:
  - Typ kann von einer oder mehreren Schnittstellen abgeleitet werden
  - Typ muss alle Schnittstellen(-member) implementieren, sonst abstrakt

Schnittstellen (2)

```
public sealed class String : IComparable, ICloneable,
                             IConvertible, IEnumerable { ... }
```

- Konvertierung des Objekts in Instanzen der Schnittstellen möglich

```
string s = "Test";
IComparable comparable = s; // Aufruf nur von IComparable-Methoden
ICloneable cloneable = s;
IEnumerable enumerable = (IEnumerable)cloneable;
```

  - Variable von einer Schnittstelle in andere konvertierbar, wenn Typ beide impl.

- explizite Implementierung von Schnittstellenmembern (Typsicherheit)

```
struct EinWertTyp : IComparable
{
   int x;
   public EinWertTyp(int x) { this.x = x; }
   public int CompareTo(EinWertTyp wert) { return x - wert.x; }
   _ int IComparable.CompareTo(object wert)
      { return CompareTo((EinWertTyp)wert); }
}
```

  - bei expliziter `CompareTo()` darf kein Zugriffsmodifizierer stehen (die Methode ist über Typobjekt `private` und über Schnittstellenvariable `public`)
  - auch Impl. von Schnittstellen mit gleichem Methodenamen und Signatur möglich
  - implementierte Schnittstellenmethoden ohne `virtual` implizit `sealed`

Geschachtelte Typen

- Zugriff auf alle Member des übergeordneten Typs
- Zugriff von ausserhalb über Typname des überg. Typs
- entsprechen den statischen inneren Klassen von Java

```
class A
{
    int x = 3;
    Nested n = new Nested();
    public void Foo()
        { Console.WriteLine("({0},{1})", x, n.y); } // kein Zugriff!

    public class Nested
    {
        int y = 2;
        public void Foo()
        {
            A a = new A();
            Console.WriteLine("({0},{1})", a.x, y); // Zugriff erlaubt
        }
    }
}

A.Nested n = new A.Nested();
n.Foo()
```

Vererbung

- statische Methoden u. Member werden vererbt
- Ableitung: eine Basisklasse, mehrere Schnittstellen
- virtuelle Funktionsmember (Methoden, Properties, Events):
  - Implementierung vorhanden
  - Methode über Schlüsselwort `virtual` deklarieren
  - Überschreiben dieser Methode benötigt Schlüsselwort `override` (Compiler setzt ein Flag, um Überschreiben anzudeuten, wichtig für Versionierung)
- abstrakte Klassen und Member:
  - Methode abstrakt, wenn sie keine Implementierung besitzt
  - Klasse abstrakt, sobald sie eine Methode ohne Implement. besitzt
  - Deklaration muss mit Schlüsselwort `abstract` erfolgen
  - abstrakte Methoden sind implizit virtuell
  - Instanzbildung nicht möglich
  - abgeleitete Klasse muss geerbte abstrakte Member implementieren (Methoden benötigen Schlüsselwort `override`) oder abstrakt deklariert sein

Vererbung (2)

- versiegelte Klassen und Member durch `sealed`-Deklaration:
  - bei Klassendeklaration: Ableitung von Klasse verboten
  - bei `override`-Funktionsmember: Überschreiben in Subklasse verboten

```
abstract class Basis {
    public abstract void Display();
}

class A : Basis {
    public override void Display() { Console.WriteLine(“Klasse A“); }
}

class B : Basis {
    public override void Display() { Console.WriteLine(“Klasse B“); }
    public virtual string Zahl
    {
        get { return new Random().Next().ToString(); }
    }
}

sealed class C : B {
    public override string Zahl
    {
        get { return “Zufallszahl = “ + base.Zahl; }
    }
}

class D : C {}  // Compile-Fehler: Basisklasse versiegelt!
```

Vererbung (3)

```
Basis b = new Basis(); // Compile-Fehler: Instanz von abstrakter Klasse
Basis b1 = new A();
Basis b2 = new B();
B b3 = new C();
b1.Display(); // Ausgabe: Klasse A
b2.Display(); // Ausgabe: Klasse B
Console.WriteLine(b3.Zahl); // Ausgabe: Zufallszahl = 1245543523
```

- geerbte Member (alle Memberarten) mit `new` verbergen
  - Verwendung bei virtuellen Methoden:
    - Abbruch der virtuellen Aufrufkette

```
class A { public virtual void Test() {} }
class B : A { public override void Test() {} }
class C : B { public new void Test() {} }

B b = new C();
b.Test();      // Aufruf von B.Test()
A a = new C();
a.Test();      // Aufruf von B.Test()
```

  - Verwendung bei nicht virtuellen Methoden:
    - Implementierung in Ableitung änderbar (Zugriff auf `base` möglich)

Arrays

```
int[] intarray = new int[length]; // Array mit length int-Werten im Heap
Control[] ctrlarray = new Control[10]; // Array mit 10 Control-Verweisen
int[] arr = new int[] {2, 3, 4}; // Initialisierung (auch ohne new int[])
```

- implizit von `System.Array` abgeleitet (Verweistyp)
- 2 Arten:
  - rechteckig (n-Dimensionaler Block)

    ```
    int[,] = new int[5, 4]; // 2D-Array (5 Zeilen, 4 Spalten)
    string[,,] = new string[2, 2, 2]; // 3D-Array mit Stringverweisen
    ```

  - verzweigt (jagged, Arrays in Arrays)

    ```
    Point[][] polygone = new Point[2][];
    polygone[0] = new Point[10];
    polygone[1] = new Point[8];
    ```

- CLR überprüft Gültigkeit von Index

  ungültiger Index: `IndexOutOfRangeException`
- Referenztyp-Arrays konvertierbar (Werttyp-Arrays nicht)
  - `Array.Copy()` kann auch für Werttyp-Arrays zum konvertieren genutzt werden
- Array dynamisch anlegen mit `Array.CreateInstance()`

Benutzerdefinierte Attribute

- Verknüpfung zusätzlicher Informationen mit einem Ziel

  Ziele: Assembly, Modul, Typ, Feld, Methode, Parameter, Rückgabewert, Property, Event

- Deklaration im Quelltext:

  ```
  [Flags] // Zuweisung über Ziel
  enum Color { ... }
  ```

  - explizite Zuordnung durch Präfix: `[assembly:AssemblyVersion(“3.0.0.0“)]`

    (Präfixe: assembly, module, type, property, event, field, return, method, param)

- Funktionsweise:
  - Attribut = Instanz eines Typs, der von `System.Attribute` abgeleitet ist
  - Attributdeklaration = Konstruktoraufruf (bei Std-K. `()` auch weglassen, siehe `Flags`)
  - Initialisierung der Felder, Properties: `[DllImport(“Kernel32“, SetLastError=true)]`
  - mehrere Attribute möglich: `[Flags, Serializable]` = `[Flags][Serializable]`
  - Compiler serialisiert Attributinstanz und schreibt diese in die Metadaten oder wertet dieses sofort aus (`[Obsolete(“Veraltet“, IsError=true)]`, `[Conditional(“DEBUG“)]`)
  - einige Attribute sind pseudo-benutzerdef., d.h. Metadaten enthalten nur Bitflag
  - Attribute sollten im Code ausgewertet werden
    - Zugriff über Reflektionsklassen oder statische Methoden von `Attribute`
    - oft übernehmen das FCL-Klassen, beispielsweise beim Serialisieren

- Möglichkeit eigene Attribute zu definieren

Exceptions

```
try
{
    // Code, in dem Ausnahmen auftreten können
}
catch (Exception e) // Ausnahmefilter
{
    // Behandlung der aufgetretenen Ausnahmen
}
finally
{
    // Bereinigungscode
}
```

- nach `try`-Block muss `catch`- oder `finally`-Block folgen
- `catch`-Block:
  - Variablentyp:
    `System.Exception` oder abgeleitete Klasse (CLS-kompatibel)
  - Typ `Exception` hat Properties wie
    - `Message` (nützliche Erklärungen über Ursache)
    - `StackTrace` (Infos über Methoden, die vor Ausnahme aufgerufen wurden)
    - `TargetSite` (Methode, durch welche Ausnahme ausgelöst wurde)

Exceptions (2)

- Deklaration auch ohne Variablenname möglich
- Definition von keinem oder auch mehreren möglich

  (spezifischere Ausnahme vor der allgemeineren deklarieren)

```
...
catch (ArgumentException e) { ... }
catch (Exception) { ... }
catch { ... } // fängt auch nicht CLS-kompatible Ausnahmen
...
```

- Ausnahme erneut auslösen: `throw;`
  - teilweise ausgeführte Operation abbrechen und Aufrufer über Ausn. informieren

- `finally`-Block:
  - wird immer aufgerufen, egal ob Ausnahme ausgelöst wurde
  - auch dann aufgerufen, wenn in höherer Ebene Ausnahme behandelt wird
- Möglichkeit eigene Ausnahmeklassen zu definieren
- Ausnahmen auslösen (nur CLS-kompatible)

```
throw new NullReferenceException("Zugriff auf Null-Verweis");
```

  - spezifische Ausnahmen auslösen (d.h. Klassen, die keine Ableitung besitzen)

Namespaces

- Schlüsselwort `namespace` zur Deklaration
- Typen ohne Namensraumzuweisung gehören zum globalen Namensraum (oberste Ebene, in der alle Typen u. NS deklariert)
- Zugriff auf Typen über Namensraumangabe

```
System.Web.UI.WebControls.Label l = new System.Web.UI.WebControls.Label()
```

- Abhilfe gegen Schreibarbeit schafft `using`-Direktive
  - angegebener Namespace wird eingebunden und damit direkter Zugriff

```
using System.Web.UI.WebControls;
Label = new Label();
```

(Zugriff auf alle Typen in diesem Namensraum, nicht auf Subnamensräume `using System;` heißt nicht ich kann auf `Label` zugreifen)

  - Aliasdeklaration bei Namenskonflikten

```
using WebLabel = System.Web.UI.WebControls.Label;
using WinLabel = System.Windows.Forms.Label;

WebLabel webl = new WebLabel();
WinLabel winl = new WinLabel();
```

Kontrollstrukturen

- Syntax und Semantik identisch mit C++
  - Ausnahme: `switch`-Anweisung
    - jeder Zweig (auch `default`) muss mit Sprunganweisung (`break`, `goto`) enden (ausser der Zweig ist leer, dann funktioniert auch das „Durchfallen“)
    - switch-Anweisung kann auch `string` auswerten
- `foreach`-Schleife:
  - zum eleganteren Durchlaufen einer Collection
  - Anwendung auf Typen, die `IEnumerable` implementieren (alle Collections, Arrays)

```
Hashtable table = new Hashtable();
table[“1. Tag“] = “Montag“;
table[“2. Tag“] = “Dienstag“;
foreach (DictionaryEntry entry in table)
   Console.WriteLine(“table[{0}]={1}“, entry.Key, entry.Value);
```

entspricht

```
IEnumerator e = table.GetEnumerator();
while (e.MoveNext())
   Console.WriteLine(“{0}={1}“,e.Current.Key, e.Current.Value);
   // Ausgabe:    1. Tag = Montag
   //             2. Tag = Dienstag
```

Verschiedenes

- Operator `as`:
  - führt Downcast durch, bei Scheitern Verweis = `null`
  - nur auf Verweistypen anwendbar

```
int a = 7;
object o = a;
string s = o as string; // s = null, weil o auf int-Wert zeigt
s = (string)o; // InvalidCastException
```

- Operator `is`:
  - Test, ob Objekt Typ einer spezifischen Klasse oder Interface

```
Point pt = new Point3D();
if (pt is Point3D) // true
if (pt is Point) // true
if (pt is object) // true
if (pt is int) // false
```

- Operator `typeof`:
  - Abrufen des `System.Type`-Objekts für einen angegeben Typ

    Nutzen für Reflektion

```
Type t = typeof(Point); // äquivalent zu pt.GetType()
MethodInfo[] mi = t.GetMethods();
```

Verschiedenes (2)

- Operator `sizeof`:
  - Syntax u. Semantik wie C++
  - nur im `unsafe`-Modus u. nur auf Werttypen anwendbar
- `checked`, `unchecked`-Operatoren/Anweisungen:
  - arithmetische Überläufe in Anweisungen beachten oder nicht
  - bei Beachtung: Auslösen einer `OverflowException`
  - Standardeinstellung: keine Überlaufprüfung (schneller)
  - Compiler durch Schalter anweisen: `/checked`

    (Überlaufprüfung überall da ausführen, wo es nicht explizit anders steht)

```
byte b = 100;
b = checked((byte)(b + 200)); // OverflowException
b = (byte) checked(b + 200); // keine Exception,
checked
{
    b = 100;
    b = (byte)(b + 200); // oder: b += 200; OverflowException
}
```

Verschiedenes (3)

- `lock`-Anweisung:
  - verkürzte Syntax für Aufruf von `Enter()`, `Exit()` der Klasse `Monitor` (Synchronisierung von Threads)
  - Parameter ist Objekt vom Typ `object`

```
lock(this) { ... }
```

- `using`-Anweisung:

```
using (FileStream fs = new FileStream("Datei.txt", FileMode.Open)
{
    ...
}
```

  - verkürzte Syntax für

```
FileStream fs = new FileStream("Datei.txt", FileMode.Open)
try { ... }
finally
{
    if (fs != null)
        ((IDisposable)fs).Dispose();
}
```

  - nur anwendbar auf Typen, die `IDisposable` implementieren

Ausblick

- C# 2.0 Spezifikation:
  - Generics

    generische Typen (ähnelt stark den C++-Templates)
  - Anonymous Methods

    Definition von Inline-Eventhandlern

    ```
    Button.Click += delegate(object sender, EventArgs e)
                    { ... };
    ```
  - Iterators

    variableres Iterieren über Collections, Arrays
  - Partial Types

    Typ über verschiedene Dateien splitten
  - Nullable Types

    Werttypen einen unbestimmten Wert zuweisen
- Integriert im .NET Framework 2.0 (2005)
- Download Spezifikation:

  http://msdn.microsoft.com/vcsharp/team/language/

Quellen

- Jeffrey Richter: „Microsoft .NET Framework Programmierung“

  Microsoft Press, 2002

- Peter Drayton, Ben Albahari, Ted Neward: „C# in a nutshell“

  O‘Reilly, 2003

- Jeff Prosise: „.NET Entwicklerbuch“

  Microsoft Press, 2002

- MSDN-Library

- ILDasm.exe

  Intermediate Language Disassembler