C#

Die neue Sprache für Microsoft .net

Prof. Dr. Hanspeter Mössenböck
Johannes Kepler Universität Linz
Institut für Praktische Informatik

Merkmale von C#

Sehr ähnlich zu Java

70% Java, 10% C++, 5% Visual Basic, 15% neu

Wie in Java

- Objektorientierung (einf. Vererbung)
- Interfaces
- Exceptions
- Threads
- Namespaces (wie Packages)
- Strenge Typprüfung
- Garbage Collection
- Reflection
- Dyn. Laden von Code
- Spezielle String-Unterstützung
- ...

Wie in C++

- (Operator) Overloading
- Zeigerarithmetik in Unsafe Code
- Einige syntaktische Details

Neue Features in C#

Wirklich neu (vs. Java)

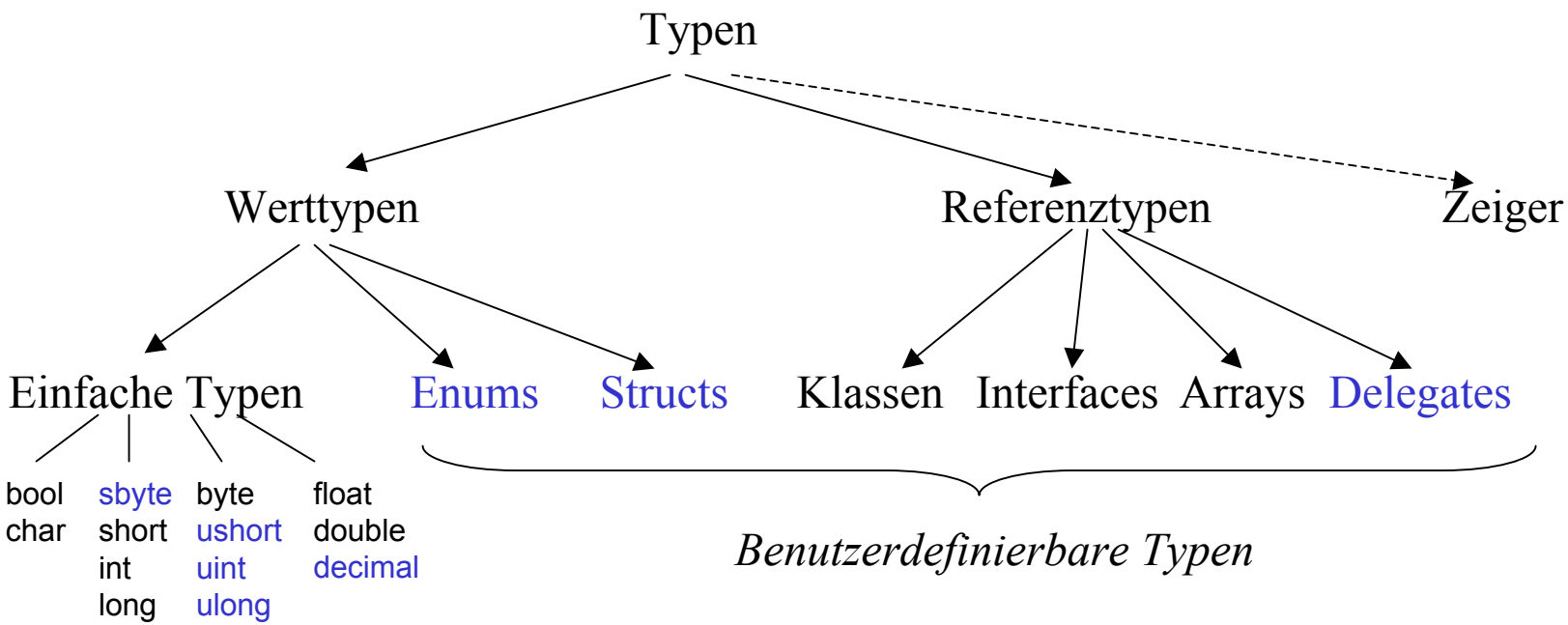
- Referenzparameter
- Objekte am Stack (Structs)
- Blockmatrizen
- Enumerationen
- Uniformes Typsystem
- goto
- Systemnahes Programmieren
- Versionierung
- Kompatibel mit anderen .NET-Sprachen

"Syntactic Sugar"

- Komponentenunterstützung
  - Properties
  - Events
- Delegates
- Indexers
- Operator Overloading
- foreach-Iterator
- Boxing/Unboxing
- Attribute
- ...

Typsystem

Einheitliches Typsystem



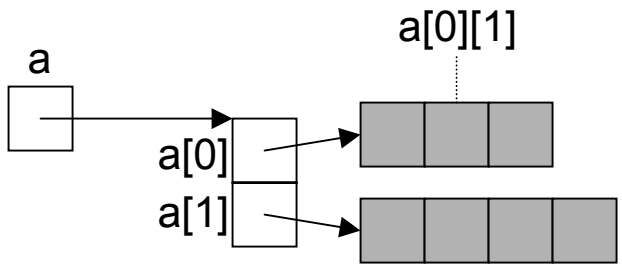
Alle Typen sind kompatibel mit object

- können *object*-Variablen zugewiesen werden
- verstehen *object*-Operationen

Mehrdimensionale Arrays

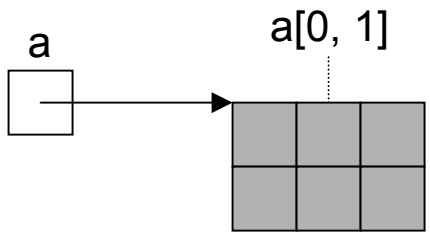
Ausgefranst (wie in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
```



Rechteckig (kompakter, effizienterer Zugriff)

```
int[,] a = new int[2, 3];

int x = a[0, 1];
```



Boxing und Unboxing

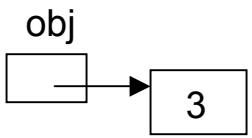
Auch Werttypen (int, struct, enum) sind zu *object* kompatibel!

Boxing

Bei der Zuweisung

```
object obj = 3;
```

wird der Wert 3 in ein Heap-Objekt eingepackt




```
class TempInt {
    int val;
    TempInt(int x) {val = x;}
}

obj = new TempInt(3);
```

Unboxing

Bei der Zuweisung

```
int x = (int) obj;
```

wird der eingepackte int-Wert wieder ausgepackt

Boxing/Unboxing

Erlaubt generische Container-Typen

```
class Queue {
        ...
        public void Enqueue(object x) {...}
        public object Dequeue() {...}
        ...
}
```

Diese Queue kann für Referenz- und Werttypen verwendet werden

```
Queue q = new Queue();

q.Enqueue(new Rectangle());
q.Enqueue(3);

Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```

Klassen, Interfaces und Vererbung

Klassen

C#

```
class A {
    private int x;
    public A(int x) { this.x = x; }
    public virtual void Foo() { ...}
}

class B : A, I1, I2 {
    const int c = 3;
    private int y;

    public B(int x, int y) : base(x) {
        this.y = y;
    }


    public override void Foo() {
        base.Foo();
        ...
    }

    public int Bar() { ...}
}
```

Java

```
class A {
    private int x;
    public A(int x) { this.x = x; }
    public void Foo() { ...}
}

class B extends A implements I1, I2 {
    static final int c = 3;
    private int y;

    public B(int x, int y) {
        super(x);
        this.y = y;
    }

    public void Foo() {
        super.Foo();
        ...
    }

    public int Bar() { ...}
}
```

Parameter von Methoden

Value-Parameter (Eingangsparameter)

Call by value

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val == 3
}
```

ref-Parameter (Übergangsparameter)

Call by reference

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

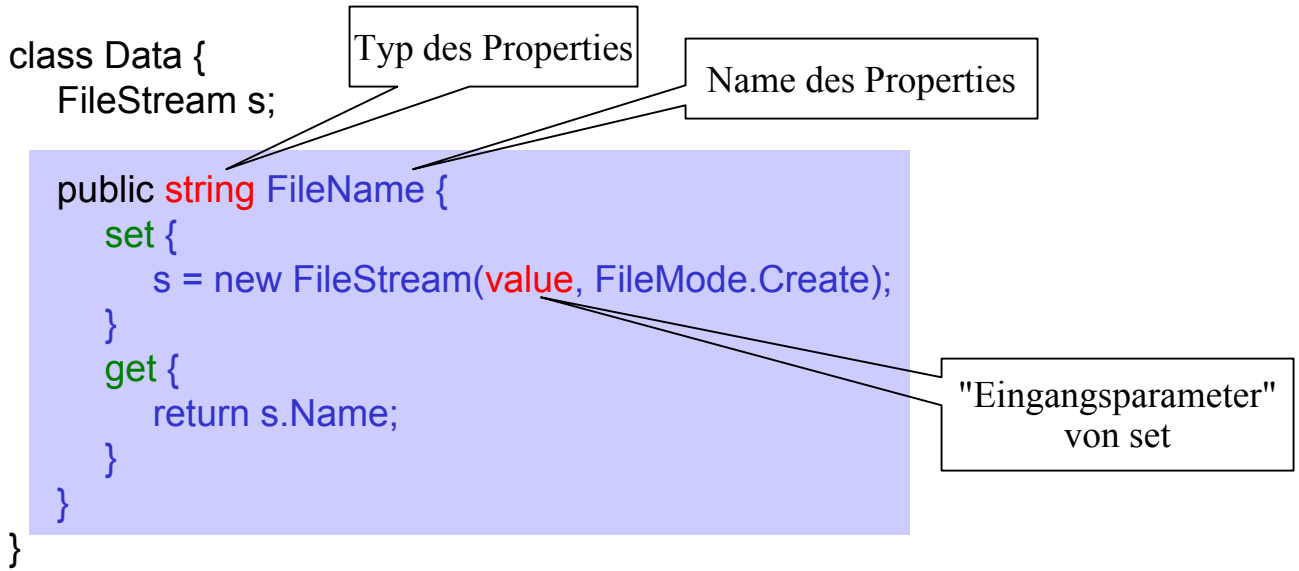
out-Parameter (Ausgangsparameter)

Wie ref-Parameter, aber wird zur Rückgabe von Werten verwendet.

```
void Read (out int first, out int next) {
    first = Console.Read(); next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

Properties

Syntaktische Kurzform für get/set-Methoden



Wird wie ein Feld benutzt ("smart fields")

```
Data d = new Data();

d.FileName = "myFile.txt";      // ruft d.set("myFile.txt") auf
string s = d.FileName;          // ruft d.get() auf
```

Durch Inlining der get/set-Aufrufe geht Zugriff gleich schnell wie normaler Feldzugriff.

Properties (Fortsetzung)

get oder set kann fehlen

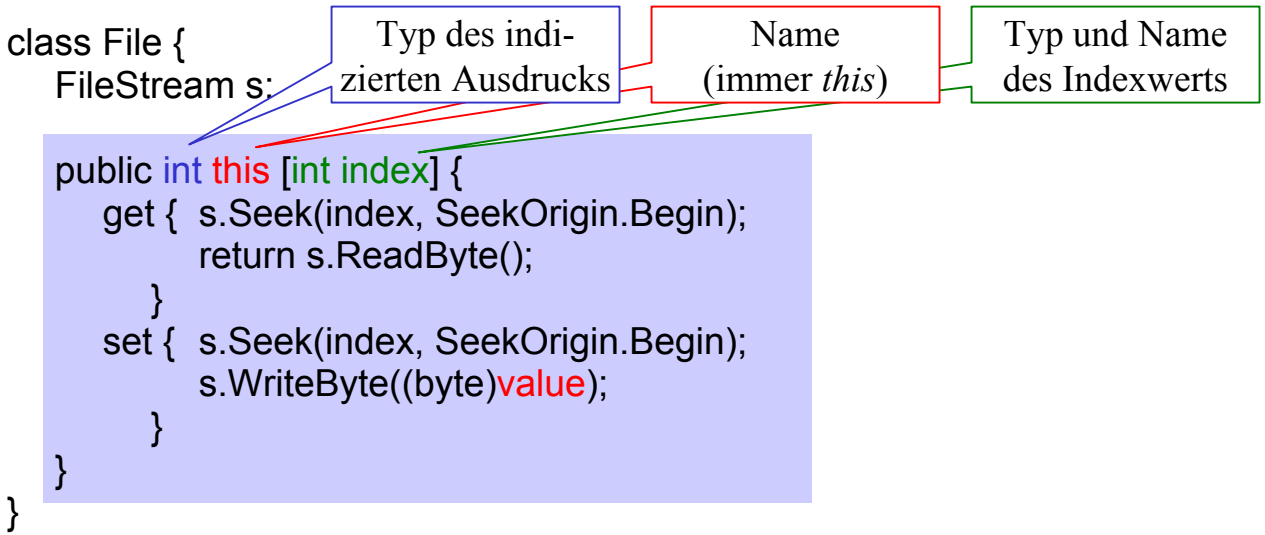
```
class Account {
    long balance;

    public long Balance {
        get { return balance; }
    }
}

x = account.Balance;        // ok
account.Balance = ...;      // verboten
```

Nutzen von Properties

- Benutzersicht und Implementierung der Daten können verschieden sein.
- read-only und write-only-Daten möglich.
- Validierung beim Zugriff möglich.
- Ersatz für Felder in Interfaces.
- Daten sind über Reflection deutlich als Property erkennbar (wichtig für Komponentenorientierte Programmierung).

Indexer

Programmierbarer Operator zum Indizieren einer Folge (Collection)




```
class File {
    FileStream s;

    public int this [int index] {
        get {  s.Seek(index, SeekOrigin.Begin);
               return s.ReadByte();
            }
        set {  s.Seek(index, SeekOrigin.Begin);
               s.WriteByte((byte)value);
            }
    }
}
```

Benutzung

```
File f = ...;
int x = f[10];          // ruft f.get(10)
f[10] = 'A';            // ruft f.set(10, 'A')
```

- get oder set-Operation kann fehlen (write-only bzw. read-only)
- Überladene Indexer mit unterschiedlichem Indextyp möglich
- .NET-Bibliothek enthält Indexer für *string* (s[i]), *ArrayList* (a[i]), usw.

Indexer (anderes Beispiel)


```
class MonthlySales {
      int[] product1 = new int[12];
      int[] product2 = new int[12];
      ...
      public int this[int i] {                    // set-Methode fehlt => read-only
          get { return product1[i-1] + product2[i-1]; }
      }

      public int this[string month] {        // überladener read-only-Indexer
          get {
              switch (month) {
                  case "Jan": return product1[0] + product2[0];
                  case "Feb": return product1[1] + product2[1];
                  ...
              }
          }
      }
}


MonthlySales sales = new MonthlySales();
...
Console.WriteLine(sales[1] + sales["Feb"]);
```

Dynamische Bindung (vereinfacht)


```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}

class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}
```

Es wird die Methode des dynamischen Typs des Empfängers aufgerufen
(stimmt nicht ganz, siehe später)

```
Animal animal = new Dog();
animal.WhoAreYou();                    // "I am a dog"
```

Jede Methode, die mit *Animal* arbeiten kann, kann auch mit *Dog* arbeiten

```
void Ask (Animal a) {
    a.WhoAreYou();
}

Ask(new Animal());          // "I am an animal"
Ask(new Dog());             // "I am a dog"
```

Dynamische Bindung (mit Verdecken)


```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal"); }
}

class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}

class Beagle : Dog {
    public new virtual void WhoAreYou() { Console.WriteLine("I am a beagle"); }
}

class AmericanBeagle : Beagle {
    public override void WhoAreYou() { Console.WriteLine("I am an American beagle"); }
}


Beagle beagle = new AmericanBeagle();
beagle.WhoAreYou();                    // "I am an American beagle"

Animal animal = new AmericanBeagle();
animal.WhoAreYou();                    // "I am a dog" !!
```

Delegates

Delegate = Methodentyp

Deklaration eines Delegate-Typs

```
delegate void Notifier (string sender);      // normale Methodensignatur
                                             // mit Schlüsselwort delegate
```

Deklaration einer Delegate-Variablen

```
Notifier notify;
```

Zuweisung einer Methode an eine Delegate-Variable

```
void SayHello(string sender) {
    Console.WriteLine("Hello from " + sender);
}

notify = new Notifier(SayHello);
```

Aufruf der Delegate-Variablen

```
notify("Max");                               // Aufruf von SayHello("Max") => "Hello from Max"
```

Zuweisung unterschiedlicher Methoden

Jede passende Methode kann einer Delegate-Variablen zugewiesen werden

```
void SayGoodBye(string sender) {
    Console.WriteLine("Good bye from " + sender);
}

notify = new Notifier(SayGoodBye);

notify("Max");          // SayGoodBye("Max") => "Good bye from Max"
```

Erzeugen von Delegate-Werten

new DelegateType (obj.Method)

- *obj* kann *this* sein (und somit fehlen)
- *Method* kann *static* sein. In diesem Fall steht statt *obj* der Klassenname.
- *Method* darf nicht *abstract*, wohl aber *virtual*, *override*, *new* sein.
- *Method*-Signatur muß mit *DelegateType* übereinstimmen
  - gleiche Anzahl von Parametern
  - gleiche Parametertypen (inklusive Return-Typ)
  - gleiche Parameterarten (ref, out, value)

Delegate-Variable speichert Methode *und* Empfängerobjekt !

Multicast-Delegates

Delegate-Variable kann mehrere Werte gleichzeitig aufnehmen

```
Notifier notify;
notify = new Notifier(SayHello);
notify += new Notifier(SayGoodBye);

notify("Max");                    // "Hello from Max"
                                  // "Good bye from Max"

notify -= new Notifier(SayHello);

notify("Max");                    // "Good bye from Max"
```

Ähnliches in Java


```
interface Notifyer {                          // entspricht Delegate-Typ
    void notify(String sender);
}
```

```
class HelloSayer implements Notifier {       // entspricht Delegate-Wert
    public void notify(String sender) {
        System.out.println("Hello from" + sender);
    }
}
```

```
Notifier n = new HelloSayer();                // entspricht Initialisierung der Delegate-Variablen
```

```
n.notify("Max");
```

- Interface übernimmt Rolle des Delegates.
- Rund um die aufzurufende Methode braucht man eine Klasse, die das Interface implementiert.
- Keine Aufrufe statischer Methoden möglich.
- Es wird nur die Methode, aber nicht der Empfänger gespeichert.
- Multicasts erfordern zusätzlich Registrierungsmechanismus und Aufrufschleife.

Threads

Threads

C#

```
void P() {
    ... thread actions ...
}

Thread t = new Thread(new ThreadStart(P));
t.Start();
```

- Erfordert keine Unterklasse von *Thread*
- Beliebige Methode kann als Thread gestartet werden.

Java

```
class MyThread extends Thread {
    public void run() {
        ... thread actions ...
    }
}

Thread t = new MyThread();
t.start();
```

- Eigene Threadklasse nötig, die Unterklasse von *Thread* sein muß.
- Thread-Aktionen müssen in Methode *run* stecken.

Synchronisation von Threads

C#

```
class Buffer {
   int[] buf = new int[10];
   int head = 0, tail = 0, n = 0;

   void put(int data) {
      lock(this) {
         while (n == 10) Monitor.Wait(this);
         buf[tail] = data; tail = (tail+1)%10; n++;
         Monitor.Pulse(this);
      }
   }

   int get() {
      lock(this) {
         while (n == 0) Monitor.Wait(this);
         int data = buf[head];
         head = (head+1)%10; n--;
         Monitor.Pulse(this);
         return data;
      }
   }
}
```

Java

```
class Buffer {
   int[] buf = new int[10];
   int head = 0, tail = 0; n = 0;

   synchronized void put(int data) {
      while (n == 10) wait();
      buf[tail] = data; tail = (tail+1)%10; n++;
      notify();
   }


   synchronized int get() {
      while (n == 0) wait();
      int data = buf[head];
      head = (head+1)%10; n--;
      notify();
      return data;
   }
}
```

Attribute

Attribute

Benutzerdefinierbare Informationen über Programmelemente

- Kann man an Typen, Members, Assemblies, etc. anhängen.
- Erweitern vordefinierte Attribute wie *public*, *sealed* oder *abstract*.
- Werden als Klassen implementiert, die von *System.Attribute* abgeleitet sind.
- Werden in Metadaten gespeichert.
- Werden teilw. von CLR-Services benutzt (Serialisierung, Remoting, COM-Interoperabilität)
- Können zur Laufzeit abgefragt werden.

Beispiel

```
[Serializable]
class C {...}                          // macht die Klasse serialisierbar
```

Auch mehrere Attribute zuordenbar

```
[Serializable] [Obsolete]
class C {...}

[Serializable, Obsolete]
class C {...}
```

Beispiel: Conditional-Attribut

Bedingter Aufruf von Methoden

```
#define debug                                   // Präprozessor-Kommando

class C {

    [Conditional("debug")]                      // geht nur bei void-Methoden
    static void Assert (bool ok, string errorMsg) {
        if (!ok) {
            Console.WriteString(errorMsg);
            System.Environment.Exit(0);         // geordneter Programmabbruch
        }
    }

    static void Main (string[] arg) {
        Assert(arg.Length > 0, "no arguments specified");
        Assert(arg[0] == "...", "invalid argument");
        ...
    }
}
```

Assert wird nur aufgerufen, wenn *debug* definiert ist.
Auch gut verwendbar für Hilfsdrucke.

Beispiel: Serialisierung


```
[Serializable]
class List {
    int val;
    [NonSerialized] string name;
    List next;

    public List(int x, string s) {val = x; name = s; next = null;}
}
```

[Serializable] Anwendbar auf Klassen.
Daten von Objekten dieser Klassen werden automatisch serialisiert.

[NonSerialized] Anwendbar auf Felder.
Diese Felder werden von der Serialisierung ausgenommen.

Zusammenfassung

C# versus Java

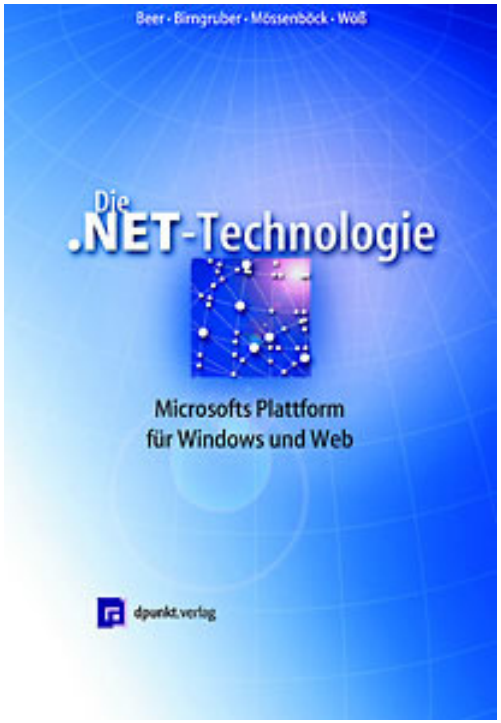
Gleiche Eigenschaften

- Objektorientierung
- Exceptions
- Threads
- Reflection

Zusätzlich

- Kompatibel zu anderen .NET-Sprachen (aber vorwiegend unter Windows)
- Uniformes Typsystem (Boxing, Unboxing)
- Blockmatrizen
- Referenzparameter
- Properties, Indexers
- Delegates
- Attribute
- Versionierung

Neues Buch



Beer, Birngruber, Mössenböck, Wöß:

Die .NET-Technologie

Microsofts Plattform für Windows und Web

dpunkt-Verlag, 2002 (erscheint im August)

- C#
- Common Language Runtime
- .NET-Klassenbibliothek
- ADO.NET
- ASP.NET
- Web Services

siehe auch

http://dotnet.jku.at