Neue Features in C# 2.0

- Generische Typen
- Iteratoren
- Vereinfachte Delegate-Erzeugung
- Anonyme Methoden
- Partielle Klassen
- Sonstiges

Generische Typen

Probleme ohne generische Typen

Klassen mit unterschiedlichen Elementtypen

```
class Buffer {
    private object[] data;
    public void Put(object x) {...}
    public object Get() {...}
}
```

Probleme

- Typumwandlungen nötig

```
buffer.Put(3);                // Boxing kostet Zeit
int x = (int)buffer.Get();    // Typumwandlung kostet Zeit
```

- Homogenität kann nicht erzwungen werden

```
buffer.Put(3); buffer.Put(new Rectangle());
Rectangle r = (Rectangle)buffer.Get();  // kann zu Laufzeitfehler führen!
```

- Spezielle Typen IntBuffer, RectangleBuffer, ... führen zu Redundanz

Generische Klasse Buffer

generischer Typ | Platzhaltertyp

```
class Buffer<Element> {
   private Element[] data;
   public Buffer(int size) {...}
   public void Put(Element x) {...}
   public Element Get() {...}
}
```

- geht auch für Structs und Interfaces
- Platzhaltertyp *Element* kann wie normaler Typ verwendet werden

Benutzung

```
Buffer<int> a = new Buffer<int>(100);
a.Put(3);                  // nur int-Parameter erlaubt; kein Boxing
int i = a.Get();           // keine Typumwandlung nötig
```

```
Buffer<Rectangle> b = new Buffer<Rectangle>(100);
b.Put(new Rectangle());    // nur Rectangle-Parameter erlaubt
Rectangle r = b.Get();     // keine Typumwandlung nötig
```

Vorteile

- Homogene Datenstruktur mit Compilezeit-Typprüfung
- Effizienz (kein Boxing, keine Typumwandlungen)

Generizität auch in Ada, Eiffel, C++ (Templates), Java 1.5

Mehrere Platzhaltertypen

Buffer mit Prioritäten

```
class Buffer <Element, Priority> {
   private Element[] data;
   private Priority[] prio;
   public void Put(Element x, Priority prio) {...}
   public void Get(out Element x, out Priority prio) {...}
}
```

Verwendung

```
Buffer<int, int> a = new Buffer<int, int>();
a.Put(100, 0);
int elem, prio;
a.Get(out elem, out prio);
```

```
Buffer<Rectangle, double> b = new Buffer<Rectangle, double>();
b.Put(new Rectangle(), 0.5);
Rectangle r; double prio;
b.Get(out r, out prio);
```

C++ erlaubt auch die Angabe von Konstanten als Platzhalter, C# nicht

Constraints

Annahmen über Platzhaltertypen werden als Basistypen ausgedrückt

Interface oder Basisklasse

```
class OrderedBuffer <Element, Priority> where Priority: IComparable {
   Element[] data;
   Priority[] prio;
   int lastElem;
   ...
   public void Put(Element x, Priority p) {
      int i = lastElement;
      while (i >= 0 && p.CompareTo(prio[i]) > 0) {data[i+1] = data[i]; prio[i+1] = prio[i]; i--;}
      data[i+1] = x; prio[i+1] = p;
   }
}
```

Erlaubt Operationen auf Elemente von Platzhaltertypen

Verwendung

```
OrderedBuffer<int, int> a = new OrderedBuffer<int, int>();
a.Put(100, 3);
```

Parameter muß IComparable unterstützen

Mehrere Constraints möglich

```
class OrderedBuffer <Element, Priority>
   where Element: MyClass
   where Priority: IComparable
   where Priority: ISerializable {
   ...
   public void Put(Element x, Priority p) {...}
   public void Get(out Element x, out Priority p) {...}
}
```

Verwendung

muß Unterklasse von MyClass sein

muß IComparable und ISerializable unterstützen

```
OrderedBuffer<MySubclass, MyPrio> a = new OrderedBuffer<MySubclass, MyPrio>();
...
a.Put(new MySubclass(), new MyPrio(100));
```

Konstruktor-Constraints

Zum Erzeugen neuer Objekte in einem generischen Typ

```
class Stack<T, E> where E: Exception, new() {
    T[] data = ...;
    int top = -1;
    public void Push(T x) {
        if (top >= data.Length)
            throw new E();
        else
            data[++top] = x;
    }
}
```

spezifiziert, daß der Platzhalter *E* einen parameterlosen Konstruktor haben muß.

Verwendung

```
class MyException: Exception {
    public MyException(): base("stack overflow or underflow") {}
}
```

```
Stack<int, MyException> stack = new Stack<int, MyException>();
...
stack.Push(3);
```

Generizität und Vererbung


```
class Buffer <Element>: List<Element> {
    ...
    public void Put(Element x) {
        this.Add(x);  // Add wurde von List geerbt
    }
}
```

kann auch generische Interfaces implementieren

Von welchen Klassen darf eine generische Klasse erben?

- von einer gewöhnlichen Klasse

```
class T<X>: B {...}
```

- von einer konkretisierten generischen Klasse

```
class T<X>: B<int> {...}
```

- von einer generischen Klasse mit gleichem Platzhalter

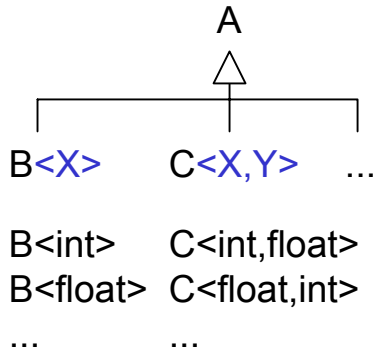
```
class T<X>: B<X> {...}
```

Kompatibilität in Zuweisungen

Zuweisung von T<x> an gewöhnliche Oberklasse

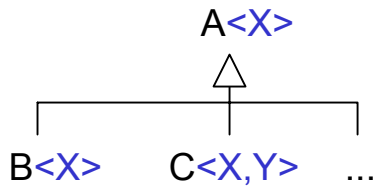
```
class A {...}
class B<X>:     A {...}
class C<X,Y>:   A {...}
```

```
A a1 = new B<int>();
A a2 = new C<int, float>();
```



Zuweisung von T<x> an generische Oberklasse

```
class A<X> {...}
class B<X>:     A<X> {...}
class C<X,Y>:   A<X> {...}
```




```
A<int> a1 = new B<int>();
A<int> a2 = new C<int, float>();
```

erlaubt, wenn korrespondierende Platzhalter durch denselben Typ ersetzt wurden

```
A<int> a3 = new B<short>();
```

verboten

Überschreiben von Methoden

Wenn von konkretisierter Klasse geerbt

```
class MyBuffer: Buffer<int> {
    ...
    public override void Put(int x) {...}
}
```

Put<Element> von Buffer<Element> geerbt

Wenn von generischer Klasse geerbt

```
class MyBuffer<Element>: Buffer<Element> {
    ...
    public override void Put(Element x) {...}
}
```

Folgendes geht nicht (man kann keinen Platzhalter erben)

```
class MyBuffer: Buffer<Element> {
    ...
    public override void Put(Element x) {...}
}
```

Laufzeittypprüfungen

Konkretisierter generischer Typ kann wie normaler Typ verwendet werden

```
Buffer<int> buf = new Buffer<int>(20);
object obj = buf;

if (obj is Buffer<int>)
    buf = (Buffer<int>) obj;

Type t = typeof(Buffer<int>);
Console.WriteLine(t.Name);  // => Buffer[System.Int32]
```

Reflection liefert auch die konkreten Platzhaltertypen!

Generische Methoden

Methoden, die mit verschiedenen Datentypen arbeiten können

```
static void Sort<T> (T[] a) where T: IComparable {
   for (int i = 0; i < a.Length-1; i++) {
      for (int j = i+1; j < a.Length; j++) {
         if (a[j].CompareTo(a[i]) < 0) {
            T x = a[i]; a[i] = a[j]; a[j] = x;
         }
      }
   }
}
```

kann beliebige Arrays sortieren, solange die Elemente *IComparable* implementieren

Benutzung

```
int[] a = {3, 7, 2, 5, 3};
...
Sort<int>(a);   // a == {2, 3, 3, 5, 7}
```

```
string[] s = {"one", "two", "three"};
...
Sort<string>(s);   // s == {"one", "three", "two"}
```

Meist weiß der Compiler aus den Parametern welchen Typ er für den Platzhalter einsetzen muß, so daß man einfach schreiben kann:

```
Sort(a);   // a == {2, 3, 3, 5, 7}
```

```
Sort(s);   // s == {"one", "three", "two"}
```

Generische Delegates

```
delegate bool Check<T>(T value);
class Payment {
    public DateTime date;
    public int amount;
}
class Account {
    ArrayList payments = new ArrayList();
    public void Add(Payment p) { payments.Add(p); }
    public int AmountPayed(Check<Payment> matches) {
        int val = 0;
        foreach (Payment p in payments)
            if (matches(p)) val += p.amount;
        return val;
    }
}
```

Es wird eine Prüfmethode übergeben, die für jedes Payment prüft, ob es in Frage kommt

```
bool PaymentsAfter(Payment p) {
    return DateTime.Compare(p.date, myDate) >= 0;
}
...
myDate = new DateTime(2003, 11, 1);
int val = account.AmountPayed(new Check<Payment>(PaymentsAfter));
```

```
int val = account.AmountPayed(delegate(Payment p) {
    return DateTime.Compare(p.date, new DateTime(2003,11,1)) >= 0;
});
```

als anonyme Methode

Nullwerte

Nullsetzen eines Werts

```
void Foo<T>() {
    T x = null;             // Fehler
    T y = 0;                // Fehler
    T z = T.default;        // ok! 0, '\0', false, null
}
```

Abfragen auf null

```
void Foo<T>(T x) {
    if (x == null) {
        Console.WriteLine(x + " == null");
    } else {
        Console.WriteLine(x + " != null");
    }
}
```

```
Foo(3);                 // 3 != null
Foo(0);                 // 0 != null
Foo("Hello");           // Hello != null
Foo<string>(null);      //  == null
```

Namensraum System.Collections.Generic

Neue generische Typen

Klassen

List<T>	entspricht *ArrayList*	Es gibt kein *SortedList<T>*
Dictionary<T, U>	entspricht *HashTable*	*List<T>* kann aber mit
SortedDictionary<T, U>		*Sort*-Methode sortiert werden
Stack<T>		
Queue<T>		

Interfaces

ICollection<T>
IList<T>
IDictionary<T, U>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

Was geschieht hinter den Kulissen?


```
class Buffer<Element> {...}
```

Compiler erzeugt CIL-Code für Klasse Buffer mit Platzhalter für Element.

Konkretisierung mit Werttypen

```
Buffer<int> a = new Buffer<int>();
```

CLR erzeugt zur Laufzeit neue Klasse Buffer<int>, in der Element durch int ersetzt wird.

```
Buffer<int> b = new Buffer<int>();
```

Verwendet vorhandenes BufferInt.

```
Buffer<float> c = new Buffer<float>();
```

CLR erzeugt zur Laufzeit neue Klasse Buffer<float>, in der Element durch float ersetzt wird.

Konkretisierung mit Referenztypen

```
Buffer<string> a = new Buffer<string>();
```

CLR erzeugt zur Laufzeit neue Klasse Buffer<object>, die mit allen Referenztypen arbeiten kan.

```
Buffer<string> b = new Buffer<string>();
```

Verwendet vorhandenes BufferObject.

```
Buffer<Node> b = new Buffer<Node>();
```

Verwendet vorhandenes BufferObject.

Unterschiede zu anderen Sprachen

C++ ähnliche Syntax

```
template <class Element>
class Buffer {
   ...
   void Put(Element x);
}
Buffer<int> b1;
Buffer<int> b2;
```

- Compiler erzeugt für jede Konkretisierung eine neue Klasse
- keine Constraints, weniger typsicher
- dafür können Platzhalter auch Konstanten sein

Java

- für Version 1.5 angekündigt
- Platzhalter können nur durch Referenztypen ersetzt werden
- durch Type-Casts implementiert (kostet Laufzeit)
- Reflection liefert keine exakte Typinformation

Iteratoren

Iteratoren bisher

foreach-Schleife kann für Klassen verwendet werden, die IEnumerable implementieren

```
class MyClass: IEnumerable {
    ...
    public IEnumerator GetEnumerator() {
        return new MyEnumerator(...);
    }

    class MyEnumerator: IEnumerator {
        public object Current { get {...} }
        public bool MoveNext() {...}
        public void Reset() {...}
    }
}
```

```
interface IEnumerable {
    IEnumerator GetEnumerator();
}
```

```
MyClass x = new MyClass();
...
foreach (object obj in x) ...
```

aufwendig!!

Iterator-Methode


```
class MyClass {
    string first = "first";
    string second = "second";
    string third = "third";
    ...
    public IEnumerator GetEnumerator() {
        yield return first;
        yield return second;
        yield return third;
    }
}
```

Merkmale einer Iteratormethode

1. Hat die Signatur
   public IEnumerator GetEnumerator
2. Anweisungsteil enthält zumindest eine yield-Anweisung

```
MyClass x = new MyClass();
...
foreach (string s in x) Console.Write(s + " ");
// liefert "first second third"
```

Funktionsweise einer Iteratormethode

1. Liefert eine Folge von Werten
2. foreach-Anweisung durchläuft diese Werte

Achtung:

- *MyClass* muß *IEnumerable* gar nicht implementieren!
- Statt *IEnumerator* sollte besser *IEnumerator<int>* verwendet werden (kein Cast nötig)
- *IEnumerator<T>* ist in *System.Collections.Generic*

Was geschieht hinter den Kulissen?

```
public IEnumerator<int> GetEnumerator() {
    try {
        ...
    } finally {
        ...
    }
}
```

liefert ein Enumerator-Objekt

```
class _Enumerator : IEnumerator<int> {
    int Current { get {...} }
    bool MoveNext() {...}
    void Dispose() {...}
}
```

```
foreach (int x in list)
    Console.WriteLine(x);
```

wird übersetzt in

```
IEnumerator<int> _e = list.GetEnumerator();
try {
    while (_e.MoveNext())
        Console.WriteLine(_e.Current);
} finally {
    if (_e != null) _e.Dispose();
}
```

MoveNext läuft bis zum nächsten yield

Dispose ruft einen eventuell vorhandenen finally-Block in der Iterator-Methode auf

yield-Anweisung

2 Arten

`yield return expr;`

- liefert einen Wert an die foreach-Schleife
- darf nur in einer Iterator-Methode vorkommen
- Typ von expr muß kompatibel sein zu
  - wenn IEnumerator<T>, dann T
  - sonst object

`yield break;`

- bricht die Iteration ab
- darf nur in einer Iterator-Methode vorkommen

Spezifische Iteratoren

```
class MyList {
   int[] data = ...;
   public IEnumerator<int> GetEnumerator() {
      for (int i = 0; i < data.Length; i++)
         yield return data[i];
   }
   public IEnumerable<int> Range(int from, int to) {
      if (to > data.Length) to = data.Length;
      for (int i = from; i < to; i++)
         yield return data[i];
   }
   public IEnumerable<int> Downwards {
      get {
         for (int i = data.Length - 1; i >= 0; i--)
            yield return data[i];
      }
   }
}
```

Standard-Iterator

Spezifischer Iterator als Methode

- beliebiger Name und Parameterliste
- Rückgabetyp IEnumerable!

Spezifischer Iterator als Property

- beliebiger Name
- Rückgabetyp IEnumerable!

```
MyList list = new MyList();
foreach (int x in list) Console.WriteLine(x);
foreach (int x in list.Range(2, 7)) Console.WriteLine(x);
foreach (int x in list.Downwards) Console.WriteLine(x);
```

Übersetzung spezifischer Iteratoren


```
public IEnumerable<int> Range(int from, int to) {
    if (to > data.Length) to = data.Length;
    for (int i = from; i < to; i++)
        yield return data[i];
}
```

liefert ein Enumerable-Objekt

```
class _Enumerable : IEnumerable<int> {
    IEnumerator<int> GetEnumerator();
}
```

dieses liefert ein Enumerator-Objekt

```
class _Enumerator : IEnumerator<int> {
    int from, to;
    int Current { get {...} }
    bool MoveNext() {...}
    void Dispose() {..}
}
```

```
foreach (int x in list.Range(2, 7))
    Console.WriteLine(x);
```

wird übersetzt in

```
IEnumerator<int> _e =
    list.Range(2, 7).GetEnumerator();
try {
    while (_e.MoveNext())
        Console.WriteLine(_e.Current);
} finally {
    if (_e != null) _e.Dispose();
}
```

Iterieren über einen Baum


```
class Tree {
   Node root = null;

   public void Add(int val) {...}
   public bool Contains(int val) {...}

   public IEnumerator<int> GetEnumerator() {
      return root.GetEnumerator();
   }
}
```

```
class Node {
   public int val;
   public Node left, right;

   public Node(int x) { val = x; }

   public IEnumerator<int> GetEnumerator() {
      if (left != null)
         foreach (int x in left) yield return x;
      yield return val;
      if (right != null)
         foreach (int x in right) yield returnx;
   }
}
```

Benutzung

```
...
Tree tree = new Tree();
...
foreach (int x in tree)
   Console.WriteLine(x);
```

Es wird hier allerdings für jeden Knoten des Baums ein Enumerator-Objekt erzeugt!

Vereinfachte Delegate-Erzeugung

Vereinfachte Delegate-Erzeugung


```
delegate void Printer(string s);

void Foo(string s) {
   Console.WriteLine(s);
}
```

```
Printer print;
print = new Printer(this.Foo);
print = this.Foo;
print = Foo;
```

Vereinfachte Form:
Delegate-Typ wird aus dem Typ der linken Seite abgeleitet

```
delegate double Function(double x);

double Foo(double x) {
   return x * x;
}
```

```
Printer print = Foo;
Function square = Foo;
```

weist *Foo(string s)* zu
weist *Foo(double x)* zu

Überladung kann durch Typ der linken Seite aufgelöst werden

Anonyme Methoden

Normale Delegates

```
delegate void Visitor(Node p);

class C {
   int sum = 0;

   void SupUp(Node p) { sum += p.value; }
   void Print(Node p) { Console.WriteLine(p.value); }

   void Foo() {
      List list = new List();
      list.ForAll(SumUp);
      list.ForAll(Print);
   }
}
```

```
class List {
   Node[] data = ...;
   ...
   public void ForAll(Visitor visit) {
      for (int i = 0; i < data.Length; i++)
         visit(data[i]);
   }
}
```

- erfordern Deklaration einer benannten Methode (*SumUp*, *Print*, ...)
- *SumUp* und *Print* können nicht auf lokale Variablen von *Foo* zugreifen
  => *sum* muß global deklariert werden

Anonyme Methoden

```
delegate void Visitor(Node p);

class C {

   void Foo() {
      List list = new List();
      int sum = 0;
      list.ForAll(delegate (Node p) { Console.WriteLine(p.value); });
      list.ForAll(delegate (Node p) { sum += p.value; });
   }
}
```

formale Parameter — Code

```
class List {
   ...
   public void ForAll(Visitor visit) {
      ...
   }
}
```

- Methodencode wird in-place angegeben
- keine Deklaration einer benannten Methode nötig
- aufgerufene Methode kann auf lokale Variable *sum* zugreifen
- return beendet anonyme Methode, nicht die äußere Methode

Restriktionen

- anonyme Methoden dürfen keine formalen Parameter der Art *params T[]* enthalten
- anonyme Methoden dürfen nicht an *object* zugewiesen werden
- anonyme Methoden dürfen keine ref- und out-Parameter äußerer Methoden ansprechen

Weitere Vereinfachung


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

```
Button button = new Button();
Button.Click += delegate (object sender, EventArgs arg) { Console.WriteLine("clicked"); };
```

Kann vereinfacht werden zu

```
Button.Click += delegate { Console.WriteLine("clicked"); };
```

Formale Parameter können weggelassen werden,
wenn sie im Methodenrumpf nicht benutzt werden

Restriktion

- Bei anonymen Methoden ohne formale Parameter darf der Delegate-Typ keine out-Parameter haben

Äußere Variablen

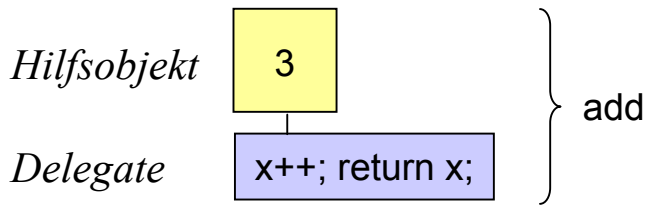
Wenn anonyme Methoden auf Variablen der umgebenden Methode zugreifen, werden diese in ein Hilfsobjekt ausgelagert (capturing).

```
delegate int Adder();

class Test {
    static Adder CreateAdder() {
        int x = 0;
        return delegate { x++; return x; };
    }
    static void Main() {
        Adder add = CreateAdder();
        Console.WriteLine(add());
        Console.WriteLine(add());
        Console.WriteLine(add());
    }
}
```

Ausgabe:

```
1
2
3
```



Das Hilfsobjekt lebt so lange wie das Delegate-Objekt

Partielle Typen

Klasse aus mehreren Teilen


```
public partial class C {
   int x;
   public void M1(...) {...}
   public int M2(...) {...}
}
```

Datei Part1.cs

```
public partial class C {
   string y;
   public void M3(...) {...}
   public void M4(...) {...}
   public void M5(...) {...}
}
```

Datei Part2.cs

Zweck

- Teile können nach Funktionalitäten gruppiert werden
- verschiedene Entwickler können gleichzeitig an derselben Klasse arbeiten
- erster Teil kann maschinengeneriert sein, zweiter Teil handgeschrieben

Sollte nur in Ausnahmefällen verwendet werden

Sonstiges

Statische Klassen

Dürfen nur statische Felder und Methoden enthalten

```
static class Math {
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    ...
}
```

Zweck

- bessere Dokumentation, daß die Klasse nur statische Methoden enthält
- Compiler kann sicherstellen, daß man in keiner der Methoden *static* vergessen hat

Math darf nicht als Typ verwendet werden