Generisches Programmieren/ Generics

Von der Objektorientierung unabhängiges Konzept!

Paradigma wird von C# parallel zur Objektorientierung angeboten.

Generisches Programmieren/ Generics

Variable von Klassen und Strukturen und Methodenparameter haben einen festen, statischen Datentyp

→ der Compiler kann für Typsicherheit sorgen, d.h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern

Aber:

- Oft werden völlig analog arbeitende Methoden für unterschiedliche Datentypen benötigt, man möchte den Algorithmus nicht für jeden Datentyp duplizieren:
    - z.B. ein Quicksort, das integers und chars sortieren kann
    - eine Liste oder ein Stack, die für Elemente verschiedenen (einheitlichen) Typs benutzbar sein sollen
    - CTrucks und CMopeds sollten sortierbar sein nach Preis, nach Reichweite, ...

→ *typgenerische (=typunabhängige)* Definition der (Sortier-)Methode mit Typplatzhaltern

Bei der *Verwendung* des Algorithmus ist der zu verarbeitende Elementtyp erst zur Laufzeit konkret festzulegen

Generisches Programmieren

Besonders erfolgreiches Anwendungsfeld für Typgenerizität:

Klassen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen und sonstige Collections/ Container, die Objekte enthalten können

Generics: Beispiel mit List<T>

Die Klasse *CFleet* zur Objektverwaltung und Ausleihe

mit den Methoden:

- Book()
- CancelBooking()
- AddRentalObject()
- RemoveRentalObject()

soll nicht nur für die CVehicle-Objekte eines Rent-A-Car funktionieren,

sondern (alternativ!) auch für die Appartments eines Rent-A-Home (AirBnB) vom Typ CHome.

Die Klasse *CFleet* könnte und sollte dann *CRentalsManager* heissen.

Generisches Programmieren: Motivation

Wir könnten die Klasse **ArrayList** (**System.Collections)** als Container für Objekte beliebigen Typs verwendet:

```
var rentals = new ArrayList();
            //... Zu verwaltende Objekte anlegen
             rentals.Add (moped24); // Typ ist „CMoped“
             rentals.Add (truck4);   // Typ ist „CTruck“
             rentals.Add (appartment12);      // Typ ist „CSingleRoom“
```

Die Nachteile der Typbeliebigkeit können allerdings stören:

Fehlende Typsicherheit

Wenn beliebige Objekte zugelassen sind (Typ **object**), kann der Compiler nicht sicherstellen, dass ausschließlich Objekte des gewünschten Typs in den Container eingefüllt werden.

Notwendigkeit von expliziten Typumwandlungen

Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihres Typs ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.

Generics: Beispiel mit List<T>

```
using System.Collections.Generic;
```

Bessere Lösung gesucht für alternative (nicht gemeinsame!) Verwaltung von:

Klasse CHome:

```
public class CHome
{
   public float  PricePerDay {get; set; }
   public int    SquareMeter {get; set; }

   public CHome (float price, int sqm)
   {  PricePerDay = price; SquareMeter = sqm;   }
}
```

Klasse CVehicle:

```
public class CVehicle
{
   public float PricePerDay {get; set; }
   public int    Seats {get; set; }

   public CHome (float price, int seats)
   {  PricePerDay = price; Seats = seats;   }
}
```

Klassen CHome und CVehicle sollen NICHT von der gleichen Basisklasse erben müssen!!!

Generics: Beispiel mit List<T>

Klasse CFleet bisher (für CVehicle):

```
public class CFleet
{
   public void Book (CVehicle v ) { … booked.Add(v); … }
   public bool CancelBooking (CVehicle v ) { … booked.Remove(v); … }
   public void AddRentalObject (CVehicle v ) { … rentals.Add(v); … };
   public void RemoveRentalObject (CVehicle v ) { … rentals.Remove(v); … }
   public float CalculateCost (CVehicle v, DateTime from, DateTime to) { …}

   CVehicle[] rentals = new CVehicle[]();
   CVehicle[] booked = new  CVehicle[] ();
}
```

Klasse CRentalManager neu (für CVehicle oder CHome):

```
public class CRentalManager<T>
{
   public void Book (T v ) { … booked.Add(v); … }
   public bool CancelBooking (T v ) { … booked.Remove(v); … }
   public void AddRentalObject (T v ) { … rentals.Add(v); … };
   public void RemoveRentalObject (T v ) { … rentals.Remove(v); … }
   public float CalculateCost (T v, DateTime from, DateTime to) { …}

   List<T> rentals = new List<T>();
   List<T> booked = new List<T>();
}
```

Benutzung für Homes:

```
CRentalManager<CHome> myStore = new CRentalManager<CHome> ();

CHome aFlat = new CHome (price: 50, sqm: 25);

myStore.AddRentalObject(aHome);
```

Benutzung für Vehicles:

```
CRentalManager<CVehicle> myStore = new CRentalManager<CVehicle> ();

CVehicle aTruck = new CTruck (price: 100, seats:2);

myStore.AddRentalObject(aTruck);
```

Generisches Programmieren

Realisierte Ziele:

- Wiederverwendung von Code

  Code mit generischem Design kann für unterschiedliche Typkonkretisierungen verwendet werden.

- Typsicherheit

  Im Vergleich zur Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object, CItem**) für die Elemente in einer Kollektion spart man sich lästige und fehleranfällige Typanpassungen.

- Performanz

  Bei Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object**) für Kollektionselemente werden aufwändige (Un-)Boxing - Operationen werden in großer Zahl fällig für Elemente mit **Werttyp**

  Im Vergleich dazu bringt die generische Programmierung eine erhebliche Verbesserung der Performanz.

Generische Klassen

Wesentlicher Vorteil einer (typ-)generischen Klasse:

mit *einer* Definition werden beliebig viele konkrete Klassen für spezielle Datentypen geschaffen.

- bei den Collections-/ Containerklassen sehr verbreitet (FCL-Namensraum **System.Collections.Generic**), aber nicht darauf beschränkt.

Zusätzlich: generische Strukturen, Schnittstellen, Delegates und Ereignisse.

Eine generische Klassendefinition verwendet **Typformalparameter <T>** // oder jeden anderen Platzhalter

Beispiel: Klasse **CRentalManager<T>**

Typparameter generischer Klassen

Die Namen der Typformalparameter können völlig frei gewählt werden, häufig verwendet werden (zumindest als Präfix):

T Type

E Element — Keine Beschränkung auf Einzelbuchstaben! Bsp.: **Dictionary<TKey, TValue>**

R Return Type

K Key

V Value

Restriktionen für Typformalparameter

Häufig muss eine generische Verwendung der Typen, die einen Typparameter konkretisieren dürfen, gewisse Eigenschaften voraussetzen:

- Muss z. B. ein generischer Container (z.B. *List<T>*) seine Elemente sortieren,
  verlangt man in der Regel von jedem konkreten Elementtyp (*CVehicle, CHome*),
  dass er das *Interface* **IComparable<T>** erfüllt,
  d.h. eine Methode namens **CompareTo()** besitzt (beschrieben unter Verwendung des Typparameters **T**):
  Elemente dieses Typs müssen vergleichbar sein

  ↓

  public int CompareTo (T *element***)**

Restriktionen für Typformalparameter

public int CompareTo (T *element*)

int

		CompareTo() - Ergebnis
Das befragte Objekt ist im Vergleich zum Aktualparameter …	kleiner	-1
	gleich	0
	größer	1

```
public class AnySortedList<T,P> where T : System.IComparable<T>,... // weitere IF
                                where P : System.IComparable<P>,... // weitere IF
{   ...
```

Restriktionen für Typformalparameter (Vergleichbarkeit)

```
using System;

int main ()
{
    fleet.AddToFleet(aTruck);  // CVehicle
    fleet. PrintItems();

    fleet.SortItems();
    fleet.PrintVehicles();
    //...

public class CFleet<T> where T : IComparable<T>,... // weitere IF
{  ...
     public void PrintVehicles()
     {
         foreach (var veh in fleet)
               Console.WriteLine
                     ($"Typ: {veh.GetType().Name}, Preis: {veh.ToString()}");
               Console.WriteLine();
}
```

```
public abstract class CVehicle: IComparable
{
   string color;
   private static int Count { get; set; }    // nur 1x pro Klasse
   public CVehicle Next { get; set;  }
   public int CompareTo(object? obj)
   {
       // compares to this.Price
       CVehicle vec = (CVehicle)obj;
       return (this.PricePerDay < vec.PricePerDay)  ?  -1  :
                                  ( this.PricePerDay == vec.PricePerDay ?  0 : 1 );
   }

   public override string ToString()
   {
      return PricePerDay.ToString("G");
   }
```

```
Typ: CDress, Preis: 140,5
Typ: CTruck,  Preis: 98
Typ:  CMoped, Preis: 275

Typ: CCar,    Preis: 98
Typ: CTruck,  Preis: 140,5
Typ: CMoped, Preis: 275
```

Schlüsselwort ***where*** (generischer Typconstraint)

- kann eine Basisklasse und/ oder Interfaces für einen Typ vorschreiben.
- Mit dem Schlüsselwort **class** wird vereinbart, dass nur Referenztypen erlaubt sind (where T: class)
- Mit dem Schlüsselwort **struct** wird vereinbart, dass nur Werttypen erlaubt sind (where T: struct)
- Man kann auch *mehrere* Restriktionen durch Kommata getrennt angeben, die von einem konkreten Typ allesamt zu erfüllen sind (**class** oder **struct** ggf. am Anfang einer Liste) (where T: class, IComparable, IEnumerable)
- Während nur *eine* Basisklasse vorgeschrieben werden darf bei der Vererbung, sind beliebig viele Schnittstellen erlaubt, die ein konkreter Typ *alle* erfüllen muss (von denen ein konkreter Typ erben muss).
- Mit dem Listeneintrag **new()** wird für die konkreten Typen ein parameterfreier Konstruktor vorgeschrieben.
  In einer *Liste* von Restriktionen muss der Eintrag **new()** ggf. am Ende stehen.
- Sind mehrere Typformalparameter mit Restriktionen vorhanden, ist für jeden Parameter eine eigene **where**-Klausel anzugeben, und die **where**-Klauseln sind durch Kommata zu trennen

```
(   public class Sample<T, U> where T: class, new(),
                              where U: struct
)
```

Generische Klassen und Vererbung

Bei der Definition einer generischen Klasse kann man als Basisklasse verwenden:

- eine nicht-generische Klasse, z. B.

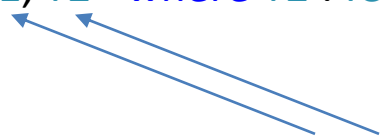
```
class GenDerived<T> : BaseClass
{   . . .
}
```

- eine geschlossene konstruierte Klasse, z.B.:

```
class GenDerived<T> : GenBaseClass<int, double>
{   . . .
}
```

- eine sogenannte *offene konstruierte Klasse* mit Typformalparametern, wobei Typrestriktionen der Basisklasse ggf. zu wiederholen sind, z. B.:

```
class GenBaseClass<T1, T2> where T2 : IComparable<T2> // Basisklasse
{   . . .
}
class GenDerived<T> : GenBaseClass<int, T> where T : IComparable<T>
{   . . .
}
```



Nullable Datentypen

Datentypen mit angehängtem Fragezeichen können == null sein (ohne Wert)

Bsp.: int?
bool?
double?

Wie geht das? Wie merkt sich die Runtime, dass der aktuelle Wert „kein Wert“ ist?

Nullable<T> als Beispiel für generische Strukturen

```
public struct Nullable<T> where T : struct
{
        private bool hasValue;    // Wert gültig?

        private T value;          // Wert

        public Nullable(T value) // c`tor mit Wert
        {
                this.value = value;
                this.hasValue = true;
        }
        public bool HasValue
        {
                get { return hasValue; }
        }

        public T Value
        {
                get { return value; }
        }
                . . .
}
```

Werte eines elementaren Datentyps lassen sich so verpacken, dass neben den normalen Werten auch die Aussage **null** zur Verfügung steht (*undefinierter* Zustand).

Bsp:
Typ **bool**: bekommt neben den Werten **true** und **false** noch den dritten Wert **null**: *unbekannt* (noch nicht initialisiert)

Variable vom Typ **Nullable<bool>**: // entspricht *bool?*

```
Nullable<bool> status;      // Instanz des Typs Nullable<T>
```

Man kann einer **Nullable**-Instanz jeden Wert des Grundtyps und außerdem den Wert **null** zuweisen, z. B.:

```
status = null;
```

Andere Notation für Nullable Datentypen

Notation mit angehängtem Fragezeichen:

Bsp.: Datentyp nullable bool

```
bool varb;    // bool, Wert undefiniert
varb= false;  // Wert ist == false

bool? varbn;  // nullable bool, Wert == null
varbn=false;  // Wert ist == false

if (null != varbn) { /* var hat einen Wert, kann verarbeitet werden  */ }

oder: if (varbn.HasValue) { /* ... */ }  // HasValue ist Property der Struktur nullable<T>
```

Vorteile:

- Compiler gibt Warnungen aus, wenn Nullable-Verweise dereferenziert werden, ohne dass ihr Wert zunächst auf null geprüft wird
- Der Compiler gibt auch Warnungen aus, wenn Non-Nullable-Verweisen ein Wert zugewiesen wird, der null sein kann

bool? ist nicht mehr nur ein Bit, sondern eine Struktur, die neben dem Datenwert noch die Information hasValue (bool Member) enthält

Operationen zwischen Nullables und nicht-Nullfähigen

Hat ein beteiligter Operand den Wert **null** (nicht initialisiert), so erhält auch der Ausdruck diesen Wert, z. B.:

```
double?  d1 = 1.0,   d2 = null;
double?  s = d1 + d2;
Console.WriteLine(s.HasValue);  // liefert false, da s null ist (keinen definierten Wert hat) !
```

Generische Methoden

- eine generische Methode ist oft besser als mehrere überladene mit unterschiedlichen Signaturen oder der Typ „object“ als Basisklasse

- Beispiel: Methode Max<T> liefert das Maximum von 2 Argumenten, Typ (Klasse) T erfüllt Interface **Comparable<T>**

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static T Max<T>(T x, T y) where T : IComparable<T> {`
`  return x.CompareTo(y) >= 0 ? x : y;`
`  }`
 	
`  public static void Main() {`
`  Console.WriteLine("int-max:\t" + Max(12, 13));`
`  Console.WriteLine("double-max:\t" + Max(2.16, 47.11));`
`  Console.WriteLine("String-max:\t" + Max("abc", "def"));`
`  }`
`}`
	`int-max:  13`
	`double-max:  47,11`
	`String-max:  def`

- vom Interface IComparable sind sehr viele Datentypen abgeleitet, auch System.Byte, System.char, System.int16/32/64 ...

default (T)

= typspezifischer Nullwert zu einem Typformalparameter

default(T) liefert ...

- den Wert **null**, wenn beim Konkretisieren des generischen Typs **T** ein Referenztyp oder ein **Nullable**-Strukturtyp angegeben wurde
- den typspezifischen Nullwert, wenn für **T** ein elementarer Datentyp angegeben wurde
- eine Strukturinstanz mit typspezifischen Nullwert-Initialisierungen für alle Felder, wenn für **T** ein Strukturtyp angegeben wurde:
  - Felder mit elementarem Datentyp erhalten den typspezifischen Nullwert.
  - Felder mit einem Referenztyp oder mit einem **Nullable**-Strukturtyp erhalten den Wert **null**.

Übung

Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. In C# können Typformalparameter auch durch Werttypen konkretisiert werden.

2. Wenn eine Methode einer generischen Klasse einen Typformalparameter als Rückgabetyp verwendet, liegt eine generische Methode vor.

3. Weil **Object** eine Basisklasse von **String** ist, kann ein Objekt vom Typ **List<String>** durch eine Variable vom Typ **List<Object>** verwaltet werden.

4. Eine generische Klasse darf von einer nicht-generischen Klasse abstammen.

Übung

Als dynamisch wachsender Container für Elemente mit einem festen *Werttyp* (z. B. **int**) ist die Klasse **ArrayList** nicht gut geeignet, weil der Elementtyp **Object** zeitaufwändige (Un)boxing-Operationen erfordert.

Dieser Aufwand entfällt bei einer passenden Konkretisierung der generischen Klasse **List<T>**, welche dieselbe Größendynamik bietet.

Vergleichen Sie mit einem Programm den Zeitaufwand beim Einfügen von 1 Million **int**-Werten in einen **ArrayList**- bzw. **List<int>** - Container.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Prog
{
    static void Main()
    {
        const int kap = 1_000_000;
        long vorher , diff; // Zeitmessung vorbereiten


        vorher = DateTime.Now.Ticks;

        // ....  hier Aktion mit ArrayList einfuegen

        diff = DateTime.Now.Ticks - vorher;
        Console.WriteLine("Zeit in Millisek. für ArrayList :" + diff / 1.0e4);



        vorher = DateTime.Now.Ticks;

        // ...  hier Aktion mit List<T> einfügen

        diff = DateTime.Now.Ticks - vorher;
        Console.WriteLine("Zeit in Millisek. für List<int>: " + diff / 1.0e4);
    }
}
```

Lösung

Vergleichen Sie mit einem Programm den Zeitaufwand beim Einfügen von 1 Million **int**-Werten in einen **ArrayList**- bzw. **List<int>** - Container.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Prog
{
    static void Main()
    {
            const int kap = 1_000_000;
             long vorher, diff; // Zeitmessung

            vorher = DateTime.Now.Ticks;




            diff = DateTime.Now.Ticks - vorher;
            Console.WriteLine("Zeit in Millisek. für ArrayList: " + diff / 1.0e4);

            vorher = DateTime.Now.Ticks;




            diff = DateTime.Now.Ticks - vorher;
            Console.WriteLine("Zeit in Millisek. für List<int>: " + diff / 1.0e4);
    }
}
```

Übung 3

Erstellen Sie eine verbesserte Version der generischen Methode Max<T>(), die statt zweier generischer Werte einen generischen Serienparameter (Modifier: params) akzeptiert und das maximale Element einer beliebigen Anzahl von Werten liefert.

```
using System;
class Prog
{
    static T Max<T>( T x, T y ) where T : IComparable<T>
    {
            return x.CompareTo(y) >= 0 ? x : y;
    }
    public static void Main()
    {
            Console.WriteLine("int-max:\t" + Max(13, 12));
            Console.WriteLine("double-max:\t" + Max(2.16, 47.11));
            Console.WriteLine("String-max:\t" + Max("abc", "def"));
    }
}
```

Ausgabe:

```
int-max:        13
double-max:     47,11
String-max:     def
```

Lösung 3

Ausgabe:

```
int-max:        56
double-max:     79,71
String-max:     zeta
```

```
    public static void Main()
    {
            Console.WriteLine("int-max:\t" + Max(13, 4, 12,  56));
            Console.WriteLine("double-max:\t" + Max(2.16,  79.71, 47.11, 34.2));
            Console.WriteLine("String-max:\t" + Max("def", "zeta " , "abc"));
    }
}
```

Übung 4

Nutzen Sie die Klasse Dictionary<char, int> aus dem Namensraum System.Collections.Generic, um die Häufigkeit der einzelnen Zeichen im Satz „Otto spielt Lotto“ zu zählen.

Geben Sie anschließend alle Key-Value-Paare aus.

Geben Sie dann alle Keys aus, und dann alle Values.

Lösung 4

```
0:1 t:5 o:3  :2 s:1 p:1 i:1 e:1 l:1 L:1
0 t o   s p i e l L
1 5 3 2 1 1 1 1 1 1
```