# Generics

Ingo Köster

Diplom-Informatiker (FH)

# Generics

- Variablen dienen als Platzhalter für Werte
- Generics sind Platzhalter für Datentypen
- Mittels Generics Klassen und Methoden definieren, für die bestimmte Datentypen zur Entwicklungszeit noch nicht bekannt sind
- In C++ als Templates bekannt

# Motivation

› Beispiel: Ein Stack der Objekte speichern kann

```
class Stack {
object[] elements;
```

…

› Verwendung:

```
Stack stack = new Stack(10);
stack.Push(2);
int wert = (int)stack.Pop();// Okay
string wert = (string)stack.Pop(); // Fehler
```

# Motivation

› Alternative

```
public class StackInt {
private int[] elemente; …
public void Push(int number) {...} ... }
```

› Lösung gewährleistet Typsicherheit

› Führt ggf. zu einer großen Anzahl ähnlicher Klassen für einen spezifischen Typ

› Klassen sind schlecht zu warten

› Ggf. wird in Zukunft ein weiterer Stack mit einem noch nicht berücksichtigten Typ nötig

# Generics

- Generics erlauben die Verwendung von Datentypen, die zum Zeitpunkt der Entwicklung noch nicht feststehen
- Anstelle eines konkreten Datentyps wird ein Platzhalter angegeben
- Platzhalter wird in spitzen Klammern angegeben
- Wird innerhalb der Klasse wie ein regulärer Datentyp verwendet

# Generics

› Beispiel mit Generics

```
class Stack<T> {
  T[] elements;
   …
```

› Beispiel ohne Generics

```
class Stack {
  object[] elements;
  …
```

# Generics

› Bei der Instanziierung der Klasse wird der generische Typparameter durch einen konkreten Datentyp ersetzt

› Beispiel:

```
Stack<int> stack = new Stack<int>(10);
```

› Alle Methoden der Klasse, die T als Parameter definieren oder zurückgeben, akzeptieren nur noch int-Werte

# Generics

- Klassen können zwei oder mehr generische Typparameter aufweisen

- Werden innerhalb der spitzen Klammern durch ein Komma voneinander getrennt
  - `class Demo<A, Z, H> { … }`

# Generische Methoden

› Methoden können ebenfalls generisch sein

```
class GenericClass<T>
{
 public void GenericMethod<K>(K obj)
 { ... }
}

GenericClass<string> obj = new GenericClass<string>();
obj.GenericMethod<int>(25);
```

# Generische Methoden

› Müssen nicht zu einer generischen Klasse gehören

```
static void Main(string[] args) {
  int a = 3, b = 5;
  int c = Add<int>(a,b);
}

static T Add<T>(T a, T b) {
  …
}
```

# Generische Methoden

› Generische Methode

```
static void Swap<T>(ref T a, ref T b) { … }
```

› Aufruf:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
```

› Wird das Typargument weglassen, wird der Typ vom Compiler ermittelt

```
int a = 1, b = 2;
Swap(ref a, ref b);
```

```
double x = 1, y = 2;
Swap(ref x, ref y);
```

# Schlüsselwort `default`

```
public T Pop() {
  pointer--;
  if (pointer >= 0) { return elements[pointer]; }
  else { // Methode Pop wird aufgerufen und der Stack ist leer
    pointer = 0;
    return null;
  } }
```

› Kein Problem, solange der parametrisierte Typ T ein Referenztyp ist

› Problem bei Wertetypen, da `null` einem Wertetyp nicht zugewiesen werden kann

  › die Rückgabe sollte z.B. 0 sein

# Schlüsselwort `default`

- Schlüsselwort `default` kann zwischen Referenz- und Wertetypen unterscheiden
  - liefert `null`, wenn T ein Referenztyp ist
  - liefert 0, wenn T ein Wertetyp wie int, double, etc. ist (sonst Standard des Datentyps)

```
public T Pop() {
  pointer--;
  if (pointer >= 0) { return elements[pointer]; }
  else {
    pointer = 0;
    return default(T); // oder nur return default;
  } }
```

# Vergleiche auf den Default-Wert

› Eine Variable vom Typ T kann nicht mittels == mit `default(T)` verglichen werden

› Lösung: Statische Methode `Equals` der Klasse `Object` verwenden

```
class MyClass<T> {
 T data;
 public void Test() {
   Console.WriteLine(object.Equals(data, default(T)) ? "leer" :
 "voll");
  }
}
```

# Generisches Interface

```
interface IMyInterface<T> {
}
```

› Klasse muss bei Verwendung einer generischen Schnittstelle ebenfalls generisch sein

```
class MyClass<T> : IMyInterface<T> {
}
```

# Generisches Interface - Alternative

› Typ des Interface auf die Klasse festlegen -> kein Typ-Cast mehr nötig

```
class Person : IComparable<Person> {
  public string Name { get; set; }
  public int Alter { get; set; }
  public int CompareTo(Person other) {
    return this.Alter.CompareTo(other.Alter);
  }
}
```

# Generics und Vererbung

- Die abgeleitete Klasse kann den generischen Typparameter übernehmen und selbst generisch sein
  - `class BasisKlasse<T> {…}`
  - `class KindKlasse<T> : BasisKlasse<T> {…}`
- Soll die Kind-Klasse nicht generisch sein, wird ein Typ verwendet
  - `class KindKlasse : BasisKlasse<Person> {...}`

- Eine Kind-Klasse kann generisch sein, auch wenn die Oberklasse nicht generisch ist
  - `class BasisKlasse {…}`
  - `class KindKlasse<T> : BasisKlasse {...}`

# Problem bei Umwandlungen

› Soll innerhalb des Codes einer generischen Klasse ein bestimmtes Klassenmitglied des verwendeten Typs aufgerufen werden (z.B. eine Methode), ist eine explizite Umwandlung notwendig

› Fehler, die ggf. auftreten könnten, werden erst zur Laufzeit erkannt

# Problem bei Umwandlungen

```
public T Max<T>(T a, T b) {
 return ((IComparable)a).CompareTo(b) > 0 ? a : b;
}
```

› Implementiert T den Datentyp <T> `IComparable` nicht, wird eine Ausnahme ausgelöst

› Platzhalter können mit Bedingungen (Constraints) versehen werden
  › Ähnlich zu SQL werden diese mit dem Schlüsselwort `where` notiert

# Constraints

```
public class SortedList<T> where T : IComparable
{ ... }
```

› Der spätere konkrete Typ muss die Schnittstelle `IComparable` implementieren

› Bedingung ist nicht nur auf Schnittstellen beschränkt

› Auch eine Klasse kann angegeben werden, um die Basisklasse des Typparameter T festzulegen

# Constraints

| Constraint | Beschreibung |
|---|---|
| `where T : [Basisklasse]` | Basisklasse bzw. Elternklasse |
| `where T : [Interface]` | Schnittstelle |
| `where T : class` | Klasse (Referenz-Typ) |
| `where T : struct` | Wertetyp (int, double, etc.) bzw. Struktur (Kein Nullable erlaubt) |
| `where T : new ()` | Öffentlicher parameterloser Konstruktor (muss als letztes angegeben werden) |
| `where T : U` | T muss U sein oder davon abgeleitet |

# Constraints

- Es können mehrere Bedingungen angegeben werden
  - durch Kommata voneinander getrennt

```
public class SortedList<T> where T : IComparable, ICloneable
{ ... }

class ClassA<T, U> where T : class where U : struct
{ … }

class ClassB<T> where T : class where T : new ()
{ … }
```

# Constraints

› Constraints auch für einzelne Methoden verwendbar

```
public T Test<T>(T obj) where T : class
{ … }
```

# Constraints und Vererbung

- Verwendet die Basisklasse einen Constraint muss auch die abgeleitete Klasse diesen hinter der Basisklasse angeben
  - `class BasisKlasse<T> where T : IComparable {…}`
  - `class KindKlasse<T> : BasisKlasse<T> where T : IComparable {…}`

# Constraints

- Generische Typparameter ohne Constraint werden als ungebundene Typparameter bezeichnet

- Mit einem Constraint, nennt man sie gebundene Typparameter

# Generische Klassen in der UML

- Wird durch einen Kasten mit dem generischen Parameter und Typ in der rechten oberen Ecke festgehalten
  - Z.B. `T:class`
    - Wird der Typ weggelassen, ist der Standardwert class

- Im Software Ideas Modeler:
- Rechtsklick -> Properties -> Template Parameters -> Add

- Innerhalb der Klasse wird der Platzhalter verwendet

# Generische Klassen in der UML

```
class Stack<T>
{
 T[] elements;
 int size;
 int pointer = 0;
 public Stack(int length) {…}
 public void Push(T element) {…}
 public T Pop() {…}
 public int Length {…}
}
```