Templates sehen ähnlich aus aber man verfolgt andere Ziele damit!!

.NET: Generics

Version 5.0

Komplexität von Generics

- Generics sind komplexer als vielfach angenommen.
- Können Sie diese Fragen zu (Java-)Generics beantworten?
  - Was bedeutet

    bounded wildcard

    ```
    Set<T>.addAll(Collection<? extends T> c) bzw.
    Set<T>.containsAll(Collection<?> c)?
    ```
  - Warum kann in Java auf Template-Parameter der new-Operator nicht angewandt werden?
  - Ist *ArrayList* dasselbe wie *ArrayList<Object>*?

    raw type
  - Ist *ArrayList<Person>* eine Oberklasse von *ArrayList<Student>*, ist also

    ```
    ArrayList<Person> pList = new ArrayList<Student>()
    ```

    möglich? Wenn nein, warum nicht?

    geht nicht
  - Was sind die Auswirkungen der Zuweisung

    ```
    ArrayList<? extends Person> pList =
      new ArrayList<Student>() ?
    ```

    funktioniert

    keine Möglichkeit mehr Methoden die Eingangsparameter haben aufzurufen

Probleme bei Objekt-basierten Behältern

- Beispiel Objekt-basierter Stack:

```
class Stack {
  private readonly int size;
  private object[] items;
  public        Stack(int size)   { … }
  public void   Push(object item) { … }
  public object Pop()             { … }
}
```

heterogener Behälter

- Problem 1: Laufzeitverlust durch Boxing und Unboxing.

```
Stack s = new Stack(10);
s.Push(1);
int i = (int)s.Pop();
```

1 wird geboxt - achtung hier Overhead

casten

- Problem 2: Typsicherheit zur Übersetzungszeit nicht überprüfbar.

```
s.Push(3.14);
string str = (string)s.Pop();
```

Laufzeitfehler

Generics

- Beispiel für generischen Stack:

```
class Stack<ElemType> {
  private readonly int size;
  private ElemType[]   items;
  public          Stack(int size)     { … }
  public void     Push(ElemType item) { … }
  public ElemType Pop()               { … }
}
```

- Erzeugung konkreter Stack-Klassen:

```
Stack<int> s1 = new Stack<int>(10);
s1.Push(1);
int i = s1.Pop();
```

type erasure bei Java (Integer statt int)

- Vorteile:
  - keine Typkonversionen,
  - kein Boxing/Unboxing bei Wertetypen,
  - gleiche Implementierung für verschiedene Elementtypen.

in Java schon Boxing/Unboxing

Derivation Constraint

- Für Parametertypen dürfen nur die Eigenschaften von *Object* angenommen werden:

nur Operationen verlangen die in Object vorhanden sind (ToString)

```
public class LinkedList<K,V> {
  public V Find(K key) {
    while (…) {
      if (key.CompareTo(current.key)) { // Syntaxfehler
…
```

C++ Commpiler würde hier keinen Fehler liefern -> muss mich selbst darum kümmern

IComparable Interface

- *where*-Bedingung schreibt vor, dass Parametertyp bestimmte Interfaces implementiert oder von einer bestimmten Klasse abgeleitet sein muss.

```
public class LinkedList<K,V> where K : IComparable<K> {
  V Find(K key) {
    while (…) {
      if (key.CompareTo(current.key)) {
…
```

Java: K extends Comparable

Unterschied zu Java kennen

```
LinkedList<string, int> strList;
~~LinkedList<object, int> objList;~~
```

object implementiert IComparable nicht

Constructor Constraint

- Für Parametertypen kann gefordert werden, dass sie den Default-Konstruktor implementieren.

where V : struct ist auch noch möglich

```
public class Node<K,V> where V : new() {
  private K key;
  private V value;

  public Node() {
    key = default(K);
    value = new V();
  }
}
```

gibt es in Java nicht

default ist Default Wert von dem Datentyp (null oder Default Wert)

```
Node<int, object> objNode = new Node<int, object>();
Node<int, string> objNode = new Node<int, string>();
  // Syntaxfehler: string hat keinen Default-Konstruktor.
```

- Die Verwendung von *new* für generische Typparameter ist in Java nicht möglich.

Generische Methoden

- Methoden dürfen generische Typparameter aufweisen.

```
public class Math {
  public static T Min<T>(T a, T b)
                  where T : IComparable<T> {
    if (a.CompareTo(b) < 0)
      return a;
    else
      return b;
  }
}
```

Java T extends Comparable

- Methode kann mit beliebigem Typ instanziert werden.

```
string minStr = Math.Min<string>("abc", "efg");
```

- Parametertyp kann meistens vom Compiler ermittelt werden.

```
string minStr = Math.Min("abc", "efg");
```

automatische Typermittlung

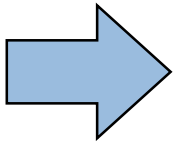
Implementierung von Generics in .NET

Laufzeitsystem von C# kennt generische Typen

Java Laufzeitsystem kennt nicht - Compiler weiß bescheid - erzeugt Code der ohne Generics auskommt (Auswirkungen z.B. auf Reflection) - deshalb auch Type erasure

C#

```
class Stack<T> {
  private readonly int size;
  private          T[] items;
  public Stack() {
    size = 10;
    items = new T[10];
  }
}
```



IL

```
.class Stack<T> {
  .field private initonly int32 size
  .field private !T[] items
  .method public … void .ctor() …
    …
    ldarg.0
    ldc.i4.s 10
    newarr !T
    stfld !0[] class Stack<!T>::items
    …
}
```

- CLR und IL wurden für generische Typen erweitert.
- Generische Typen können in Sprache A implementiert und in Sprache B instanziert werden.
- Generische Typen werden zur Laufzeit (aber nur bei Bedarf)
  - für jeden Wertetyp und
  - einmal für alle Referenztypen gemeinsam instanziert.

Vorteile von Generics

in C++ Problem - mehrere Stacks von int erzeugt Code Bloat

- Gemeinsame Nutzung des Codes (kein Code-Bloat)
- Performance-Gewinn bei generischen Behälterklassen:
  - keine Typkonversionen bei Referenztypen:

```
List<string> list = new List<string>();
string item = list[0]; // keine Typkonversion notwendig.
```

  - Kein Boxing/Unboxing bei Wertetypen:

```
List<int> list = new List<int>();
list.add(100); int item = list[0];
```

- Geringerer Speicherplatzbedarf bei Behältern mit Wertetypen.

Gewrapptes Objekt von Boolean z.B. -> braucht viel mehr Speicherplatz

je kleiner der Datentyp desto größer der Unterschied

- Zugriff auf generische Typparameter zur Laufzeit

```
List<int> list = new List<int>();
Type collType = list.GetType();
Type[] paramType = collType.GetTypeParameters();
```

paramType[0] = int

Java: Class statt Type

in Java Info nicht enthalten (int wurde durch object ersetzt - Reflection passiert zur Laufzeit - type erasure)

Unterschiede zu Generics in Java (1)

- Standarddatentypen können in Java nicht als Typparameter verwendet werden.
  - Umwandlung in Referenztypen mithilfe von Wrapper-Klassen.
- In Java stehen für Typparameter keine Konstruktoren zur Verfügung (auch nicht der Standardkonstruktor).
- Metadaten zu Typparametern sind in Java nur eingeschränkt verfügbar.
  - z. B. für Objekte vom Typ *ArrayList<String>* geht die Information über den Elementtyp verloren *(type erasure)*.
  - Statische Metadaten werden hingegen im Bytecode abgelegt: Für die Klasse

    class X implements List<String> { … }

    kann der Typparameter der generischen Basisklasse bestimmt werden.

ArrayList<Integer> list =...

list.add();

Zur Laufzeit ist die Information verloren

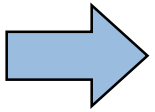
hier komme ich auf String von der Basisklasse List

Unterschiede zu Generics in Java (2)

Generische Klasse in Java

```
class Stack<T> {
  private T[] items;
  public void Push(T item) {…}
  public T    Pop()        {…}
}
```

```
Stack<Integer> s =
  new Stack<Integer>();
s.push(1);
int i = s.pop();
```



Übersetzung in Bytecode

```
class ObjectStack {
  private object[] items;
  public void Push(object item) {…}
  public object Pop()           {…}
}
```

```
ObjectStack s = new ObjectStack();
s.Push(new Integer(1));
int i =
  ((Integer)s1.Pop()).intValue();
```

Overhead

- Für Java-Generics musste die JVM nicht erweitert werden.
- Einbußen bei Laufzeit:
  - Zur Laufzeit müssen Typkonversionen durchgeführt werden.
- Erhöhter Speicherplatzbedarf bei Verwendung von Wrapper-Klassen.

Unterschiede zu Generics in Java (3)

- Kovarianz

```
Collection<? extends Person>
  coll = new ArrayList<Student>();
coll.add(new Student());
for (Person p : coll) …
```

```
interface IEnumerable<out T> { … }
```

```
IEnumerable<Person>
  coll = new List<Student>();
foreach (Person p in coll) …
```

  - Java: Für Eingangsparameter vom Typ *T* dürfen nur null-Werte übergeben werden.

- Kontravarianz

```
Comparator<Student> studCmp = …;
Comparator<Person>  persCmp = …;
Comparator<? super Student>
  cmp = persCmp;
cmp.compare(new Student(),
            new Student());
```

```
interface IComparer<in T> { … }
```

```
IComparer<Person> persCmp = …;
IComparer<Student> studCmp = …;
studCmp = persCmp;
studCmp.Compare(new Student(),
                new Student());
```

  - Java: Ausgangsparameter vom Typ *T* bekommen den Typ *Object*.

Unterschiede zu C++-Templates

- Generics sind typisierte Klassen, Templates sind „Macros“
  - Templates werden zur Compilezeit instanziert.
  - Generics werden zur Laufzeit instanziert.

weit auseinander - großer Unterschied

Templates und Generics haben unterschiedliche Designs

- Generics erhöhen Typsicherheit (bereits zur Compilezeit)
  - Viele C++-Compiler kompilieren Template-Code nicht.
  - Erst bei Template-Instanzierung wird Code erzeugt.
  - Erst bei Template-Instanzierung wird überprüft, ob Operationen auf Template-Parameter möglich sind.
- „Code Bloat“
  - Mehrfache Instanzierung von Templates für gleiche Elementtypen.
  - Längere Ladezeiten, erhöhter Speicherbedarf.
- Generics bieten weniger Funktionalität
  - Auf Parameter generischer Typen dürfen keine Operatoren angewandt werden.