

A decorative graphic on the left side of the slide, consisting of a 5x5 grid of squares in various shades of gray and blue, creating a subtle checkerboard effect.

# .NET: Generics

© J. Heinzlreiter  
Version 5.0

# Komplexität von Generics

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

- Was bedeutet

`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>`?

- Ist `ArrayList<Person>` eine Oberklasse von `ArrayList<Student>`, ist also

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

möglich? Wenn nein, warum nicht? *Nein, geht nicht*

- Was sind die Auswirkungen der Zuweisung

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

*→ nur als readonly verwenden*  
*des geht, bounded wildcard*

# Probleme bei Objekt-basierten Behältern

- Beispiel Objekt-basierter Stack:

heterogener Behälter

```
class Stack {  
    private readonly int size;  
    private object[] items; → array von Objekten  
    public Stack(int size) { ... }  
    public void Push(object item) { ... }  
    public object Pop() { ... }  
}
```

- Problem 1: Laufzeitverlust durch Boxing und Unboxing.

```
Stack s = new Stack(10);
```

s.Push(1); → Autoboxing, Hinweis: Overhead von Objekt 16 Byte da am Heap gespeichert

```
int i = (int)s.Pop(); → cast da statischer Typ Objekt ist
```

- Problem 2: Typsicherheit zur Übersetzungszeit nicht überprüfbar.

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

# Generics

- Beispiel für **generischen Stack**:

homogener Behälter → Typsicherheit

```
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();
```

bei Java müssen wir die Wrapper  
klasse von int nehmen

- Vorteile: *Typsicherheit!*

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

Generics auch im Common Type System vorhanden

# Derivation Constraint

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

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

→ nur Funktionen voraussetzen die in Object vorhanden

→ nur für Datentypen, die Comparable implementieren

- 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)) {  
...  
}
```

wenn Constraint gegeben kann ich Funktion verwenden

```
LinkedList<string, int> strList;
```

```
LinkedList<object, int> objList; implementiert Comparable nicht
```

# Constructor Constraint

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

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

*V: struct geht auch*  
*→ liefert defaultwert 0 oder null je nach Typ*

```
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

Generische Methoden sind auch in nicht generischen Klassen möglich

- 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;  
        }  
    }
```

- Methode kann mit beliebigem Typ instanziiert werden.

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

- Parametertyp kann meistens vom Compiler ermittelt werden.

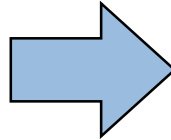
```
string minStr = Math.Min("abc", "efg");
```

# Implementierung von Generics in .NET

Off Laufzeitsystem kennt Generics  
Java kennt es nur der Compiler

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  
    ...  
}
```

→ hier gibt es generics, erst zur Laufzeit instanziiert

→ erster generischer Typ parameter

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



# Vorteile von Generics

- 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.
- Zugriff auf **generische Typparameter** zur Laufzeit

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

*in Java getClass() - kein Type Erasure*

*paramType[0] = int*

*in Java = Object*

*→ Type Erasure*

# Unterschiede zu Generics in Java (1)

- Standarddatentypen können in Java **nicht als Typparameter** verwendet werden. *int, double, boolean*
  - 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**). *zur Laufzeit ist die Typinfo weg, statische Info bleibt*
  - **Statische Metadaten** werden hingegen im **Bytecode** abgelegt: Für die Klasse  
`class X implements List<String> { ... }`  
kann der **Typparameter der generischen Basisklasse** bestimmt werden.

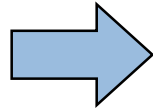
# Unterschiede zu Generics in Java (2)

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();
```

*↗ eher Objektarray  
und dann  
Größen  
nicht T*



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();
```

- 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 instanziiert.
  - Generics werden zur Laufzeit instanziiert. *von Compiler*
- 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.