# Generische Typen in C#

Generische Typen sind ein leistungsfähiges Feature in C#, das es Entwicklern ermöglicht, Klassen, Schnittstellen und Methoden zu definieren, die mit einem oder mehreren Platzhaltern für Datentypen arbeiten können. Dieses Konzept erhöht die Flexibilität und Wiederverwendbarkeit von Code und verbessert die Typensicherheit zur Compile-Zeit.

# Vorteile von Generischen Typen

- **Typensicherheit**: Generische Typen bieten zur Compile-Zeit eine starke Typprüfung. Dadurch werden Laufzeitfehler, die durch falsche Typumwandlungen entstehen können, minimiert.
- **Wiederverwendbarkeit**: Mit generischen Typen können Entwickler eine einzige Implementierung erstellen, die mit verschiedenen Datentypen funktioniert. Zum Beispiel kann eine generische Klasse Buffer<T> für verschiedene Typen wie int, string oder benutzerdefinierte Typen verwendet werden.
- **Leistung**: Generische Typen vermeiden die Kosten für Boxen und Unboxing, die auftreten, wenn Werttypen in Referenztypen umgewandelt werden. Dies führt zu einer besseren Performance, insbesondere bei der Arbeit mit Sammlungen von Werttypen.

# 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

```
class Buffer<Element> {          // Element ist der generische Typ (Platzhalter)
  private Element[] data;
  public Buffer(int size) {...}
  public void Put(Element x) {...}
  public Element Get() {...}
}
```

- geht auch für structs und interface
- 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
- Effizient (kein Boxing, keine Typumwandlungen)

Generizität auch in Ada, Eiffel, C++ (Templates), Java 5.0

# 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

```
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

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

Platzhalter `MySubclass` muss Unterklasse von MyClass sein

Platzhalter `MyPrio` muss IComparable und ISerializable unterstützen

# 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;
  }
}
```

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

# Generizität und Vererbung

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

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>();        // erlaubt, wenn korrespondierende Platzhalter
A<int> a2 = new C<int, float>();// durch denselben Typ ersetzt wurden
```

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

# Überschreiben von Methoden

```
class Buffer<Element> {
  ...
  public virtual void Put(Element x) {...}
}
```

Wenn von konkretisierter Klasse geerbt

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

...Element wird durch konkreten Typ int ersetzt

Wenn von generischer Klasse geerbt

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

...Element bleibt als Platzhalter

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;
  }
}
```

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

# Generische Delegates

```
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");
  }
}
```

# Namensraum `System.Collections.Generic`

Neu generische Typen

## Klassen

```
List<T>           // entspricht ArrayList
Dictionary<T, U>  // entspricht Hashtable
SortedDictionary<T, U>
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 Buffer<int>.

Buffer<float> c = new Buffer<float>();  // CLR erzeugt zur Laufzeit neue Klasse Buffer<float>,
                                        // in der Element durch float erstezt wird.
```

## Konkretisierung mit Referenztypen

```
Buffer<string> a = new Buffer<string>();  // CLR erzeugt zur Laufzeit neue Klasse Buffer<object>,
                                          // die mit allen Referenztypen arbeiten kann.

Buffer<string> b = new Buffer<string>();  // Verwendet vorhandenes Buffer<object>.

Buffer<Node> b = new Buffer<Node>();      // Verwendet vorhandenes Buffer<object>.
```

## 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** (ab Version 1.5)

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