C# Fortgeschrittenenschulung

Generics und Constraints

Warum Generics?

- **Flexibilität**: Erlaubt die Verarbeitung von Datenstrukturen mit verschiedenen Typen.
- **Typsicherheit**: Compiler-Fehler anstelle von Laufzeitfehlern.
- **Wiederverwendbarkeit**: Einmal erstellte Methoden oder Klassen können mit verschiedenen Typen wiederverwendet werden.

Generische Klassen und Methoden

Generische Klasse

```
public class GenericList<T>
{
    private T[] elements;

    public void Add(T element) { /* ... */ }
}
```

Generische Methode

```
public T GetMax<T>(T x, T y) where T : IComparable
{
    return x.CompareTo(y) > 0 ? x : y;
}
```

- Platzhalter-Typ `T` wird durch spezifische Datentypen beim Aufruf ersetzt.

Mehrere generische Parameter

Verwendung in Klassen

- **Definition:**
  - Mehrere generische Typen können bei der Definition einer Klasse angegeben werden.

```
public class Pair<T1, T2>
{
    public T1 First { get; set; }
    public T2 Second { get; set; }

    public Pair(T1 first, T2 second)
    {
        First = first;
        Second = second;
    }
}
```

- **Anwendung:**
  - Kombinieren von Datentypen zu einem logischen Paar oder einer Einheit.

Verwendung in Methoden

- **Definition:**
  - Methoden können ebenfalls mehrere generische Typen haben.

```
public class Utilities
{
    public static TResult Combine<T1, T2, TResult>(T1 item1, T2 item2)
    {
        // Beispielimplementierung - kombiniert die repräsentativen Strings
        return (TResult)Convert.ChangeType(item1.ToString() + item2.ToString(), typeof(TResult));
    }
}
```

Generics in Interfaces

Definition eines generischen Interfaces

```
public interface IRepository<T>
{
    void Add(T item);
    T Get(int id);
}
```

- **Vorteile:**
  - Typsicherheit für alle Implementierungen.
  - Flexibilität bei der Wiederverwendung für verschiedene Typen.

Implementierung eines generischen Interfaces

Beispiel: Eine Sammlung

```
public class ListRepository<T> : IRepository<T>
{
    private List<T> items = new List<T>();

    public void Add(T item) => items.Add(item);
    public T Get(int id) => items[id];
}
```

- Durch Implementierung des Interfaces für einen bestimmten Typ können unterschiedliche Datenstrukturen für die gleiche Logik genutzt werden.

Generics bei abgeleiteten Klassen

Basisklasse mit Generics

```
public class BaseRepository<T>
{
    protected List<T> items = new List<T>();

    public virtual void Add(T item) => items.Add(item);
}
```

- Eine Basisklasse kann generische Typen beinhalten, die von abgeleiteten Klassen spezifiziert werden.

Abgeleitete Klasse spezifiziert Generics

Spezifischere Implementierung

```
public class ProductRepository : BaseRepository<Product>
{
    public Product FindByName(string name)
    {
        return items.FirstOrDefault(p => p.Name == name);
    }
}
```

- **Vorteil:**
  - Wiederverwendbare Logik wird in der Basisklasse gekapselt.
  - Erweiterung oder Spezialisierung in abgeleiteten Klassen.

Constraints in Generics

- **Warum Constraints?**
  - Einschränkung der Typen, die für generische Klassen oder Methoden verwendet werden können.
  - Zugriff auf bestimmte Schnittstellen oder Basisklassen.
- **Beispiel für Constraint**

```
where T : IComparable
```

- **Mögliche Constraints:**
  - `where T : class`
  - `where T : struct`
  - `where T : new()`
  - `where T : <base class>`

Demo: Generische Liste mit Constraint

```
public class GenericList<T> where T: IComparable
{
    private T[] elements;

    public void Add(T element) { /* ... */ }
    public T GetMax() { /* ... */ }
}
```

- Nur Typen, die `IComparable` implementieren, können verwendet werden.
- Unterstützt die `GetMax()` -Funktionalität.

Vorteile der Verwendung von Constraints

- **Erhöhte Flexibilität**: Ermöglicht die Verwendung spezifischer Methoden wie `CompareTo`.
- **Sicherheit**: Garantiert die Verwendbarkeit bestimmter Schnittstellen in generischen Klassen/Methoden.
- **Klarheit**: Besser lesbarer Code durch Angabe von Anforderungen für Typen.

Praktische Anwendung

- Generics und Constraints sind weit verbreitet in .NET:
  - **Listen und Sammlungen**
  - **Algorithmen und Dienstklassen**
  - **Interfaces und abstrakte Basisimplementierungen**

Zusammenfassung

- Generics bieten mächtige Möglichkeiten zur Erzeugung flexibler und sicherer APIs.
- Constraints ermöglichen die Spezifikation von Anforderungen an Typeigenschaften.
- Entwickeln Sie wiederverwendbaren Code, der über viele Anwendungsszenarien hinweg einsetzbar ist.

Anwendung von Generics in der Praxis

Anstehende Übungen

- **Übung 1: Generische Sammlung**
  - Entwickeln Sie eine generische Klasse, die als Container für verschiedene Datentypen dient.
  - Implementieren Sie grundlegende Methoden wie `Add`, `Remove` und `Get`.
- **Übung 2: Generische Methode mit Constraints**
  - Erstellen Sie eine Methode, die Objekte mit `IComparable`-Constraints vergleicht.
  - Nutzen Sie die Methode, um generische Vergleiche durchzuführen.