Vorlesung Softwareentwicklung 2021

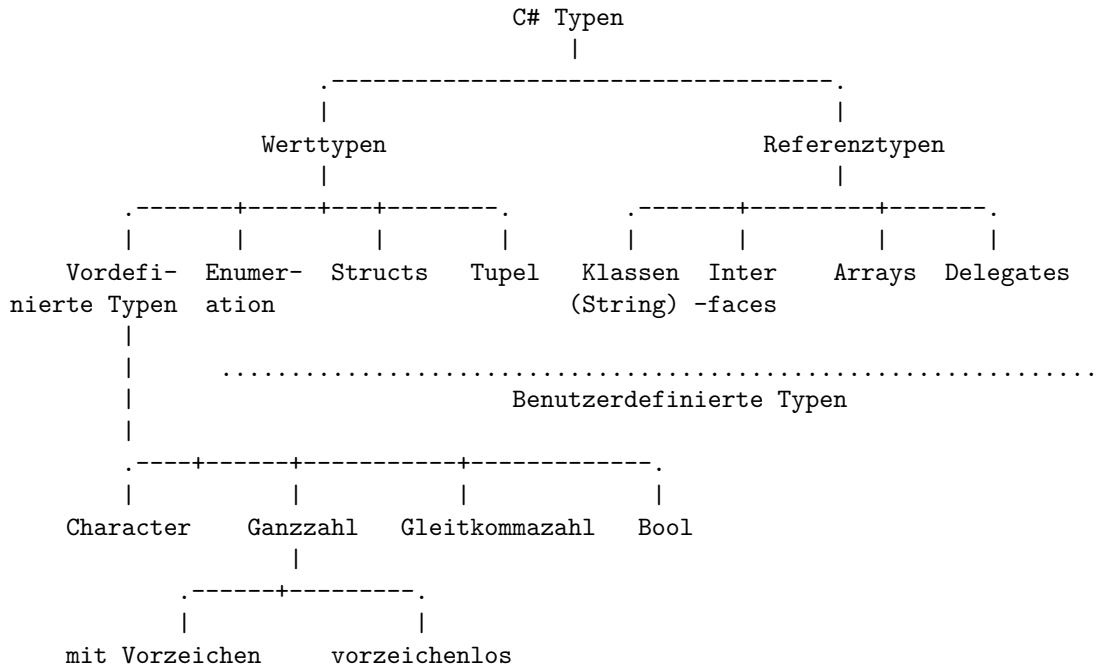
https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

Collections

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Containertypen und deren Implementierung in C#`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/20_Container.md
Autoren	@author

... zur Erinnerung und in Ergänzung



Die bisher behandelten Userdatentypen `struct` und `class` erfahren in C# 9.0 eine Erweiterung - `records`. Es wurden zwei Varianten integriert

- `record` ist nur eine Abkürzung für eine `record class` - ein Referenztyp.
- `record struct` ist ein Wertdatentyp.

```
using System;

public record PersonRecord(string FirstName, string LastName);

public class PersonClass
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Program
{
  public static void Main()
  {
      // Darstellung mit Records
      var record_1 = new PersonRecord("Calvin", "Allen");
      var record_2 = new PersonRecord("Calvin", "Allen");

      Console.WriteLine(record_1);
      Console.WriteLine(record_1 == record_2);
      //record_1.FirstName = "Tralla";

      // Darstellung mit Klasseninstanzen

      var class_1 = new PersonClass(){
            FirstName = "John",
            LastName = "Doe"
        };
      var class_2 = new PersonClass(){
            FirstName = "John",
            LastName = "Doe"
        };

      Console.WriteLine(class_1);
      Console.WriteLine(class_1 == class_2);
  }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

- die Klassen-Instanzen werden nicht als gleich angesehen, obwohl die Daten in den Objekten gleich sind. Dies liegt daran, dass die beiden Variablen auf unterschiedliche Objekte verweisen.
- die Record-Instanzen werden als gleich angesehen. Dies liegt daran, dass Datensätze bei der Überprüfung auf Gleichheit nur Daten vergleichen.
- die Records implementieren verschiedene Methoden automatisch `ToString()`
- Records sind per default immutable!

```
public record Person
{
    public string FirstName { get; init; }
    //public string FirstName { get; set; }
    public string LastName { get; init; }
    // public string LastName { get; set; }
}
```

Collections

> **Merke:** Sogenannte Container sind ein zentrales Element jeder Klassenbibliothek. Sie erlauben die Abbildung verschiedener Entitäten in einem Objekt. Im Kontext von C# wird dabei von *Collections* gesprochen.

In der vergangen Vorlesung haben wir über die Vorteile von generischen Speicherstrukturen am Beispiel der Liste gesprochen. Allerdings ist die Möglichkeit durch die Struktur hindurchzuiterieren nicht immer die günstigste. In dieser Vorlesung wollen wir alternative Konzepte und deren Implementierung im C# Framework untersuchen.

Beginnen wir zunächst mit einem Vergleich einiger listenähnlichen Konstrukte. Diese sind in den Namespaces `System.Collections` und `System.Collections.Generic` enthalten. Um zu vermeiden, dass diese beständig mitgeführt werden, betten wir sie mit `using` in unseren Code ein.

```
using System;
using System.Collections;
using System.Collections.Generic;


public class Animal
{
    public string name;
    public Animal(string name){
        this.name = name;
    }
}

public class Program{
    public static void Main(string[] args){
        Animal[] arrayOfAnimals = new Animal[3]
        {
            new Animal("Beethoven"),
            new Animal("Kitty"),
            new Animal("Wally"),
        };
        ArrayList listOfAnimals = new ArrayList()
        {
            new Animal("Beethoven"),
            new Animal("Kitty"),
            new Animal("Wally"),
        };
        List<Animal> genericlistOfAnimals = new List<Animal>()
        {
            new Animal("Beethoven"),
            new Animal("Kitty"),
            new Animal("Wally"),
        };
        foreach (Animal pet in listOfAnimals){
            Console.WriteLine(pet.name);
        }
        listOfAnimals.RemoveAt(1);
        listOfAnimals.Add(new Animal("Flipper"));
        Console.WriteLine();
        foreach (Animal pet in listOfAnimals){
            Console.WriteLine(pet.name);
        }
        Console.WriteLine("\n");
    }
}
```

Dabei setzen die vielfältigen Methoden Anforderungen an die im Container gespeicherten Werte.

```
using System;
```

```
public class Point
{
    public int x;
    public int y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}

public class ArrayExamples  {

  // Return true if X times Y is greater than 100000.
  private static bool ProductGT10(Point p)
  {
      return p.x * p.y > 100000;
  }

  public static void Main()
  {
     // Example 1 - Setzen
     String[,] myArr2 = new String[5,5];
     myArr2.SetValue( "one-three", 1, 3 );
     Console.WriteLine( "[1,3]:   {0}", myArr2.GetValue( 1, 3 ) );

     // Example 2 - Sortieren
     String[] words = { "The", "QUICK", "BROWN", "FOX", "jumps",
                        "over", "the", "lazy", "dog" };
     Array.Sort(words, 1, 3);
     foreach (var word in words){
         Console.Write(word + " ");
     }
     Console.WriteLine("\n");

     // Example 3 - Suchen
     // Create an array of five Point structures.
     Point[] points = { new Point(100, 200),
         new Point(150, 250), new Point(250, 375),
         new Point(275, 395), new Point(295, 450) };
     // Find the first Point structure for which X times Y
     // is greater than 100000.
     Point first = Array.Find(points, ProductGT10);
     // Display the first structure found.
     Console.WriteLine("Found: X = {0}, Y = {1}", first.x, first.y);
  }
}
```

Worin liegt der Unterschied zu den bereits bekannten `Array` Implementierung?

Feature	Array	`ArrayList`	`Array<T>`
Generisch?	nein	nein	ja
Anzahl der Elemente	feste Größe	variabel	variabel
Datentyp	muss homogen sein (typsicher)	kann variieren (nicht streng typisiert)	muss homogen sein
null	nicht akzeptiert	wird akzeptiert	wird akzeptiert
Dimensionen	multidimensional `array[X][Y]`	-	-

Die Methoden von `ArrayList` sind zum Beispiel unter https://docs.microsoft.com/de-de/dotnet/api/system.collections.arrayl

3.1 zu finden.

Neben den genannten existieren weitere Typen, die spezifischere Aufgaben umsetzen. Diese können entweder als sequenzielle oder als assoziative Container klassifiziert werden.

Container (in der C#-Welt sprechen wir von Collections) können durch die folgenden drei Eigenschaften charakterisiert werden:

1. Zugriff, d.h. die Art und Weise, wie auf die Objekte des Containers zugegriffen wird. Im Falle von Arrays erfolgt der Zugriff über den Array-Index. Im Falle von Stapeln (*Stack*) erfolgt der Zugriff nach der LIFO-Reihenfolge (last in, first out) und im Falle von Warteschlangen (*Queue*) nach der FIFO-Reihenfolge (first in, first out);
2. Speicherung, d.h. die Art und Weise, wie die Objekte des Containers gelagert werden;
3. Durchlaufen, d.h. die Art und Weise, wie die Objekte des Containers iteriert werden.

Von den Containerklassen wird entsprechend erwartet, dass sie folgende Methoden implementieren:

- einen leeren Container erzeugen (Konstruktor);
- Einfügen von Objekten in den Container;
- Objekte aus dem Container löschen;
- alle Objekte im Container löschen;
- auf die Objekte im Container zugreifen;
- auf die Anzahl der Objekte im Container zugreifen.

Sequenzielle-Container speichern jedes Objekt unabhängig voneinander. Auf Objekte kann direkt oder mit einem Iterator zugegriffen werden.

Ein assoziativer Container verwendet ein assoziatives Array, eine Karte oder ein Wörterbuch, das aus Schlüssel-Wert-Paaren besteht, so dass jeder Schlüssel höchstens einmal im Container erscheint. Der Schlüssel wird verwendet, um den Wert, d.h. das Objekt, zu finden, falls es im Container gespeichert ist.

Welche Container-Typen sind programmiersprachenunabhängig gängig?

Typ	Unmittelbarer Zugriff	Beschreibung
Dictionary	via Key	Wert-Schlüssel Paar
Liste	via Index	Folge von Elementen mit einem Index als Schlüssel
Queue	nur jeweils erstes Objekt	FIFO (First-In-First-Out) Speicher
Stack	nur jeweils letztes Objekt	LIFO (Last-In-First-Out) Speicher
Set		Werte ohne Dublikate
...		

Und wie sieht es mit der Performance aus? Der Beitrag des Autors `Serj-Tm` auf Stackoverflow vergleicht in einem Codebeispiel unterschiedliche Operationen für verschiedene Container-Typen.

Array	`List<T>`	Penalties	Method
00:00:01.3932446	00:00:01.6677450	1 vs 1,2	Generate
00:00:00.1856069	00:00:01.0291365	1 vs 5,5	Sum
00:00:00.4350745	00:00:00.9422126	1 vs 2,2	BlockCopy
00:00:00.2029309	00:00:00.4272936	1 vs 2,1	Sort

Fragenkatalog für die Auswahl von Collections:

Frage	Mögliche Lösungen
Sollen Elemente nach dem Auslesen verworfen werden?	`Queue<T>`, `Stack<T>`
Benötigen Sie Zugriff auf die Elemente in einer bestimmten Reihenfolge?	`Queue<T>` vs. `LinkedList<T>`
Wird die Collection in einer nebenläufigen Anwendung eingesetzt?	
Benötigen Sie Zugriff auf jedes Element über den Index?	`ArrayList`, `StringCollection` und `List<T>` vs. assoziativer Container

Frage	Mögliche Lösungen
Sollen die Dateninhalte unveränderlich sein?	`ImmutableArray<T>`, `ImmutableList<T>`
Erfolgt die Indizierung anhand der Position oder anhand eines Schlüssels?	
Müssen Sie die Elemente abweichend von ihrer Eingabereihenfolge sortieren?	`SortedList<TKey,TValue>`
Soll der Container nur Zeichenfolgen annehmen?	`StringCollection`

Containerimplementierung in Csharp

Um die Konzepte der Implementierung der Container in C# zu verstehen, versuchen wir uns nochmal an einem eigenen Konstrukt. Wir systematisieren dazu die Idee der verlinkten Liste aus der vorangegangen Veranstaltung und fokussieren uns zunächst auf die Möglichkeit mit den C#-Bordmitteln über dieser Liste zu iterieren.

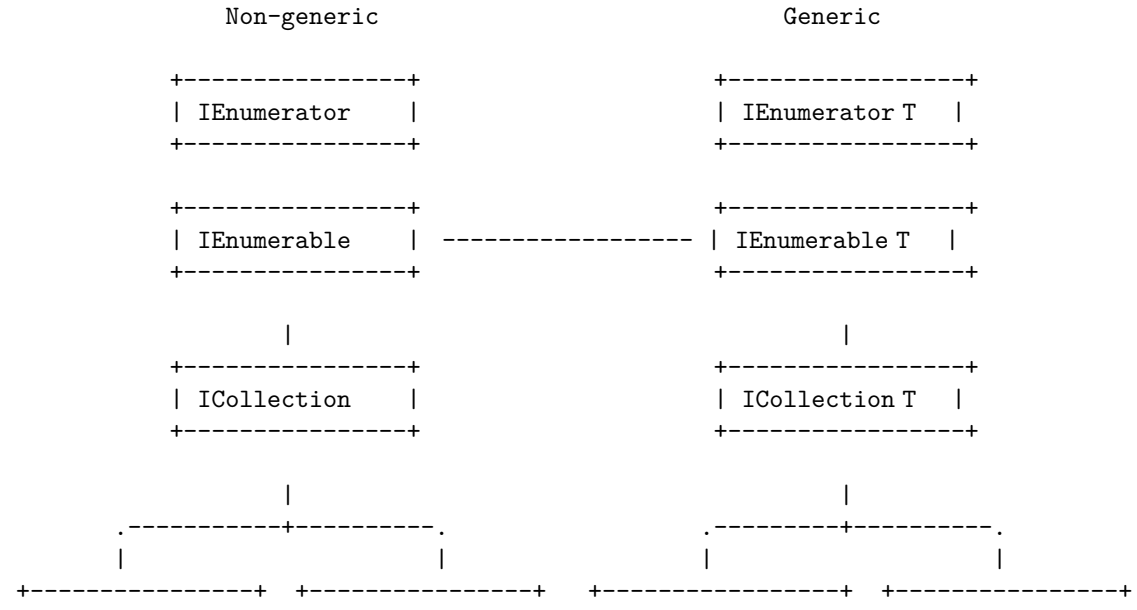
Zur Erinnerung, für die Möglichkeit der Iteration über einer Datenstruktur mittels `foreach` bedarf es der Implementierung der Interfaces `IEnumerable` und `IEnumerator`. Wir verbleiben dabei auf der generischen Seite.

```
using System;
using System.Collections;
using System.Collections.Generic;

public class GenericList<T> : IEnumerable<T>
{
    protected Node head;
    protected Node current = null;
    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;
        public Node(T t){
            next = null;
            data = t;
        }
        public Node Next {
            get { return next; }
            set { next = value; }
        }
        public T Data {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList(){
        head = null;
    }

    public void Add(T t)  {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public IEnumerator<T> GetEnumerator(){
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
```

```
            current = current.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator(){
        return GetEnumerator();
    }
}

public class Animal
{
    string name;
    int age;
    public Animal(string s, int i){
        name = s;
        age = i;
    }
    public override string ToString() => name + " : " + age;
}

class Program
{
    public static void Main(string[] args)
    {
        GenericList<Animal> animalList = new GenericList<Animal>();
    animalList.Add(new Animal("Beethoven", 8));
    animalList.Add(new Animal("Kitty", 4));
        foreach (Animal a in animalList)
        {
            System.Console.WriteLine(a.ToString());
        }
    }
}
```

Achtung: Das Beispiel implmentiert das Iteratorkonzept mittels `yield`. Damit lässt sich einige Tipparbeit sparen, die bei der konventionellen Umsetzung anfallen würde, vgl Link.

Die Methoden für das Handling der Daten beschränken sich aber auf ein `Add()` und die Iteration - hier braucht es noch deutlich mehr, um anwendbar zu sein. Um diese Funktionalität umzusetzen, greift die C#-Collections Implementierung auf eine ganze Reihe von Interfaces zurück, die den einzelnen Containern die notwendige Funktion geben.

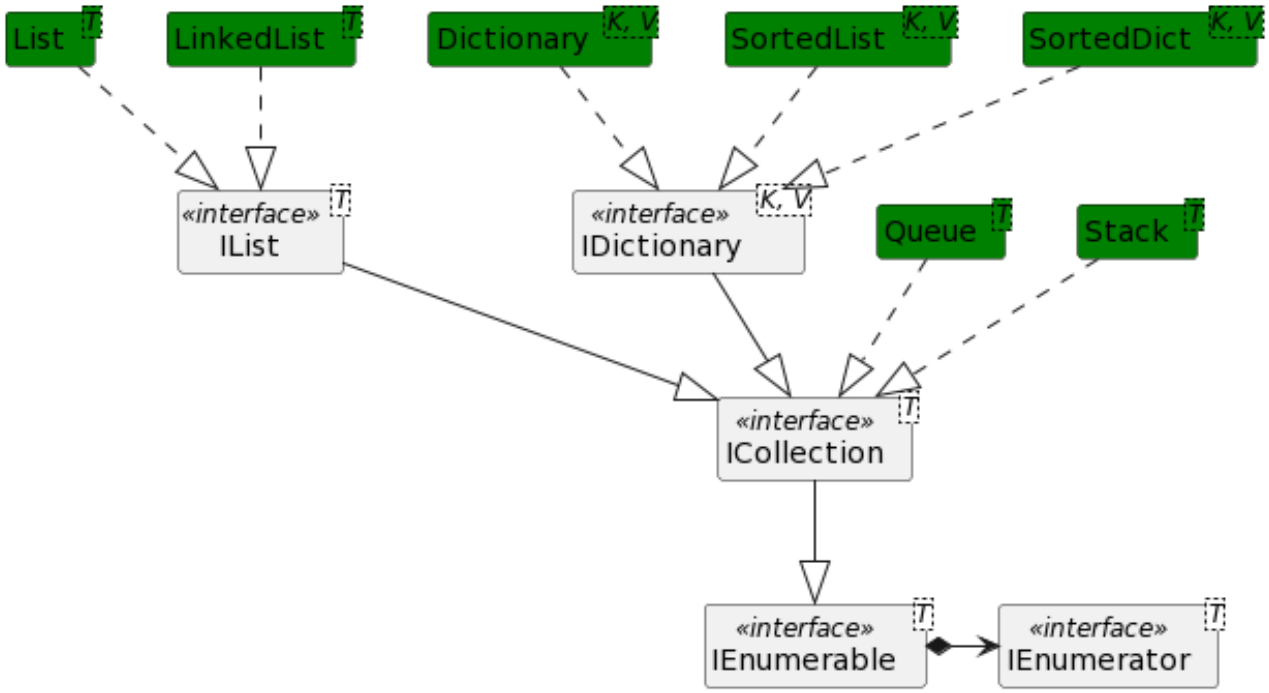


```
| IDictionary    |  | IList          |   | IDictionary T   |  | IList T        |
+----------------+  +----------------+   +-----------------+  +----------------+
```

An dieser Stelle greift das Interface `ICollection` und definiert die Methoden `Add`, `Clear`, `Contains`, `CopyTo` und `Remove`. Mit `Contains` kann geprüft werden, ob ein bestimmter Wert im Container enthalten ist. `CopyTo` extrahiert die Werte des Containers in ein Array. Dabei können bestimmte Ranges definiert werden. Die anderen Methoden sind selbsterklärend.

Schnittstelle	Spezifizierte Funktionen
IEnumerable	GetEnumerator()
ICollection	Count(), Add(), Remove()
IList	IndexOf(), Insert(), RemoveAt()
IDictionary	Keys(), Values(), TryGetValue()

Folgendes Klassendiagramm zeigt die Teile der in C# implementierten Collection-Types und deren Relationen zu den entsprechenden Interfaces.



Im Folgenden sollen Beispiele für die aufgeführten Datenstrukturen dargestellt werden.

C# Collection	Bezeichnung	Bedeutung	
List	unsortiertes Datenfeld indizierbarer Elemente	Im Unterschied zum Array “beliebig” erweiterbar	Link
SortedList	sortiertes Datenfeld	Abbildung der Reihenfolge über einen numerischen Schlüssel	Link
Stack	LIFO Datenstruktur		Link
Queue	FIFO Datenstruktur		Link
Dictionary	assoziatives Datenfeld	… Datenstruktur mit nicht-numerischen (fortlaufenden) Schlüsseln, um die enthaltenen Elemente zu adressieren.	Link

Anwendung der Generic Collections

List

```
using System;
using System.Reflection;
using System.Collections.Generic;

public class Program{
  public static void Main(string[] args){
    // Initialisieren mit Basiswerten, Ergänzungen der Liste
    var animals = new List<string>() { "bird", "dog" };
    animals.Add("cat");
    animals.Add("lion");
    // Fügt mehrere Objekte in die Liste ein
    animals.InsertRange(1, new string[] { "frog", "snake" });
    foreach (string value in animals)
    {
        Console.WriteLine("RESULT: " + value);
    }
    Console.WriteLine("In der Liste finden sich " + animals.Count + " Elemente");
    Console.WriteLine("Für die Liste reservierter Speicher (Einträge) " + animals.Capacity);
    Console.WriteLine("lion findet sich an " + animals.IndexOf("lion") + " Stelle");
    animals.Remove("lion");
    Console.WriteLine("In der Liste finden sich nun " + animals.Count + " Elemente");
  }
}
```

Dictionary<T, U>

```
using System;
using System.Reflection;
using System.Collections.Generic;


public class Program{
  public static void Main(string[] args){
     Dictionary<string, int> Telefonbuch = new Dictionary<string, int>();
     Telefonbuch.Add("Peter", 1234);
     Telefonbuch.Add("Paula", 5234);
     foreach( string s in Telefonbuch.Keys   )
     {
        Console.Write("Key = {0}\n", s);
     }
     // Enthält das Dictionary bestimmte Einträge?
     if (Telefonbuch.ContainsKey("Paula")){
         Console.WriteLine(Telefonbuch["Paula"]);
     }
     // Effektiver Zugriff
     int value;
     string key = "Peter";
     if (Telefonbuch.TryGetValue(key, out value))
     {
         Telefonbuch[key] = value + 1;
         Console.WriteLine("Wert von " + key + " " + Telefonbuch[key]);
     }
     // Mehrfache Nennung eines Eintrages
  }
}
```

HashSet

```
using System;
using System.Reflection;
```

```
using System.Collections.Generic;

public class Program{
  public static void Main(string[] args){
     HashSet<string> Telefonbuch1 = new HashSet<string>();
     Telefonbuch1.Add("Peter");
     Telefonbuch1.Add("Paula");
     Telefonbuch1.Add("Nadja");
     Telefonbuch1.Add("Paula");
     Console.Write("Telefonbuch 1: ");
     foreach(string s in Telefonbuch1){
        Console.Write(s + " ");
     }

     HashSet<string> Telefonbuch2 = new HashSet<string>();
     Telefonbuch2.Add("Klaus");
     Telefonbuch2.Add("Paula");
     Telefonbuch2.Add("Nadja");
     Console.Write("\nTelefonbuch 2: ");
     foreach(string s in Telefonbuch2){
        Console.Write(s + " ");
     }

     //Telefonbuch1.ExceptWith(Telefonbuch2);
     Telefonbuch1.UnionWith(Telefonbuch2);
     Console.Write("\nMerge       2: ");
     foreach(string s in Telefonbuch1){
        Console.Write(s + " ");
     }
  }
}
```

Achtung!

Die heute besprochenen Inhalte finden sich in verschiedenen Formen in allen höheren Programmiersprachen wieder.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
my_set.update([2, 3, 4])
print(my_set)

your_set = {4, 5, 6, 7, 8}
print(your_set)

print(my_set | your_set)
```

@Pyodide.eval

Aufgaben der Woche

- [ ] Erklären Sie, warum `Array` keine `Add`-Methode umfasst, obwohl es das Interface `IList` implementiert, dass wiederum diese einschließt. Tipp: Rufen Sie Ihr wissen um die explizite Methodenimplementierung noch mal auf.
- [ ] Die Erläuterung zu den Beschränkungen beim Einsatz von Generics im Dokumemt 19 basiert auf der nicht generischen Implementierung des Interfaces `IComparable`. Ersetzen Sie diese im Codebeispiel durch die generische Variante.
- [ ] Evaluieren Sie verschiedene Container in Bezug auf Methoden zum Einfügen, Löschen, etc. Generieren Sie dazu entsprechende künstliche Objekte, die Sie manipulieren *"Füge 100.000 int Werte in eine Liste ein."*. Messen Sie die dafür benötigten Zeiten.