Generische Collections = Container

Typen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen, Stapeln, Warteschlangen und anderen Collections im Namensraum **System.Collections.Generic**

Bei mehreren nebenläufigen Threads wird eventuell eine Thread-sichere Kollektion aus dem Namensraum **System.Collections.Concurrent** benötigt.
Klassen: **BlockingCollection<T>**, **ConcurrentDictionary<K, V>**, **ConcurrentQueue<T>** und **ConcurrentStack<T>**

Nichtgenerische Klassen aus **System.Collections** (ArrayList, HashTable, Queue, Stack)**:**
nicht mehr verwenden, nicht typsicher

Existierende Collectionklassen nutzen, statt neue zu erfinden!

Arrays versus Collections

Schwächen von Arrays:

- Größe muss beim Erstellen festgelegt werden und kann nicht mehr geändert werden (wenn ohne IList-Interface). Beim Neuanlegen und Kopieren der Elemente des alten Arrays entsteht lästige Routinearbeit.

- Das Einfügen und Entfernen von *inneren* Elementen ist mit einem hohen Aufwand verbunden.

- Kovariantes Verhalten hinsichtlich des Elementtyps (permanent drohender Programmierfehler, den der Compiler aus Kompatibilitätsgründen *nicht* verhindern kann).
  Die bei generischen Klassen verbotene Kovarianz ist bei Arrays in C# (und auch Java) leider erlaubt.
  Der folgende Quellcode wird ohne Kritik übersetzt:

```
object[] oarr = new string[] { "a", "b" };
oarr[0] = 13;
foreach ( string s in oarr)   Console.WriteLine(s.Length ); // Kovarianz!
```

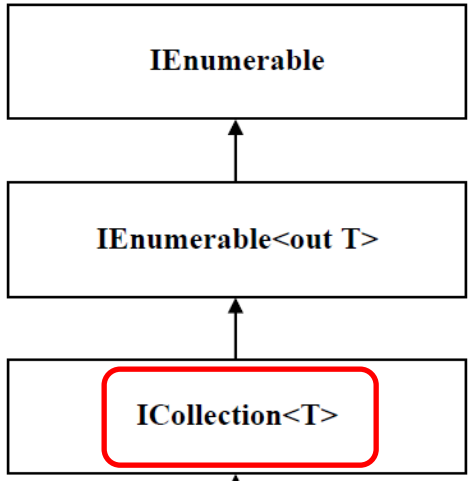
Unbehandelte Ausnahme: System.ArrayTypeMismatchException:
Es wurde versucht, auf ein Element zuzugreifen, dessen Typ mit dem Array nicht kompatibel ist.

Collections: Classes

Dictionary<TKey,TValue>	Collection von Key-Value-Paaren (Hash-Tabelle).
HashSet<T>	Duplikatfreies Set von Werten, ungeordnet
LinkedList<T>	Doppelt verlinkte Liste.
List<T>	Stark typisierte Objektliste, erlaubt Indexzugriff. Bietet Methoden für Suche, Sortieren und Listenmanipulation.
PriorityQueue<TElement,TPriority>	Collection von Items mit Wert und Priorität. Mit Dequeue() wird das Item mit der niedrigsten Priorität entnommen.
Queue<T>	First-in, First-out - Collection von Objekten.
SortedDictionary<TKey,TValue>	Collection von Key-Value-Paaren (Hash-Tabelle), nach dem Key sortiert.
SortedList<TKey,TValue>	Sortierte Collection von Listelementen
SortedSet<T>	Collection von sortierten Objekten ohne Duplikate.
Stack<T>	Last-in-First-out (LIFO) - Collection variabler Größe von Instanzen gleichen Typs.

Das Interface ICollection<T>: IEnumerable<T>

IEnumerable<T>: eine implementierende Klasse verpflichtet sich, als Rückgabe der Methode **GetEnumerator()** ein Objekt vom Typ **IEnumerator<T>** zu liefern, das ein Iterieren durch die Kollektionselemente erlaubt (in einer **foreach**-Schleife)



ICollection<T> bietet aufbauend darauf folgende Methoden:

- **public void Add (T *element*):** Parameterelement wird in die Kollektion aufgenommen (Liste: am Ende).
- **public bool Contains (T *element*):** informiert darüber, ob das Parameterelement in der Kollektion vorhanden ist.
- **public bool Remove (T *element*):** Das erste Vorkommen des Elements wird aus der Kollektion entfernt, falls enthalten. Rückgabe: wurde Kollektion verändert?
- **public void CopyTo(T[] *array*, int *index*):** Alle Kollektionselemente werden in ein kompatibles Array ab der angegebenen Indexposition kopiert. Vorhandene Array-Elemente werden dabei überschrieben.
- **public void Clear():** Alle Elemente werden entfernt.

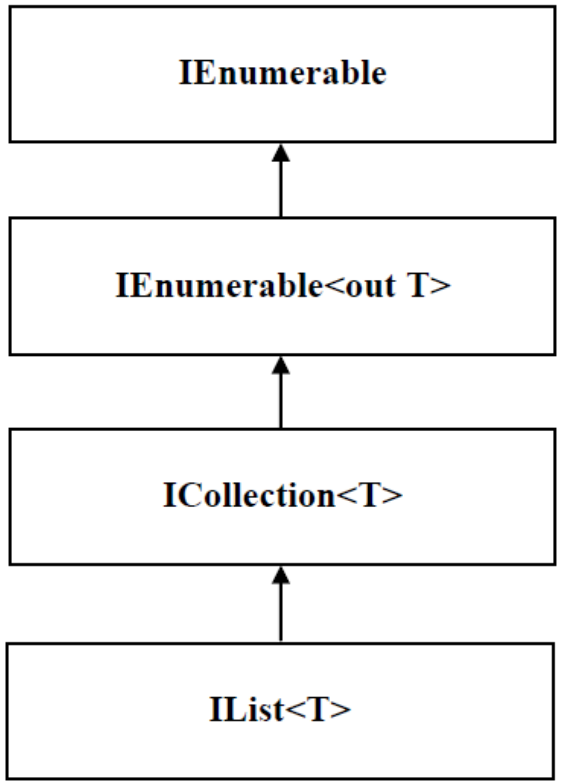
Properties von Collections

- **public int Count { get; }:** Es wird die Anzahl der Elemente geliefert.
- **public bool IsReadOnly { get; }:** Sagt, ob die Kollektion schreibgeschützt ist.

Verwaltung einer Liste List<T>

- enthält eine Sequenz von Elementen vom **selben, wählbaren Typ** (Klasse, Struktur oder Schnittstelle) mit einer **definierten Reihenfolge**
- ist Interface **IEnumerator<T>** implementiert, dann kann man auf die Elemente wahlfrei über einen nullbasierten **Index zugreifen** (analog Array)
- Liste wird bei Bedarf automatisch vergrößert
- Duplikate können enthalten sein
- **List<T>** speichert seine Elemente in einem internen Array **T[]** (Zugriff performant, Veränderung inperformant)
- Property **Capacity** ist meist größer als die Property **Count,** der aktuell belegten Anzahl

Verwaltung einer Liste List<T>



IList-Interface unterstützt:

- **public int IndexOf (T *element*)**
  Falls das Element in der Liste vorhanden ist, wird der Index des ersten Auftretens geliefert, anderenfalls der Wert -1.

- **public void Insert(int *pos*, T *element*)**
  Das Parameterelement wird an der gewünschten Indexposition eingefügt.

- **public void RemoveAt(int *pos*)**
  Das Element an der angegebenen Indexposition wird entfernt.

Indexer:

- **public T this[int *index*] { get; set; }**

Instanzmethode für schreibgeschützten Zugriff:

- **public ReadOnlyCollection<T> AsReadOnly()** , gibt eine readOnly-Sicht auf die Liste heraus

Verwaltung einer Liste: Suchen und Sortieren

BinarySearch(Int32, Int32, T, IComparer<T>)	Searches a range of elements in the sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(T)	Searches the entire sorted List<T> for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(T, IComparer<T>)	Searches the entire sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
Exists(Predicate<T>)	Determines whether the List<T> contains elements that match the conditions defined by the specified predicate.
Find(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.
FindAll(Predicate<T>)	Retrieves all the elements that match the conditions defined by the specified predicate.
FindIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
FindIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List<T> that extends from the specified index to the last element.
FindIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the entire List<T>.
FindLast(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List<T>.
FindLastIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.
FindLastIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
FindLastIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the entire List<T>.
LastIndexOf(T)	Searches for the specified object and returns the zero-based index of the last occurrence within the entire List<T>.
LastIndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
LastIndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.
Reverse()	Reverses the order of the elements in the entire List<T>.
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
Sort()	Sorts the elements in the entire List<T> using the default comparer.
Sort(Comparison<T>)	Sorts the elements in the entire List<T> using the specified Comparison<T>.
Sort(IComparer<T>)	Sorts the elements in the entire List<T> using the specified comparer.
Sort(Int32, Int32, IComparer<T>)	Sorts the elements in a range of elements in List<T> using the specified comparer.
TrueForAll(Predicate<T>)	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

Verwaltung einer Liste: LinkedList<T>

Klasse **LinkedList<T>** für doppelt verkettete Elemente :

- macht den eigentlichen Inhalt über eine get/set - Eigenschaft namens **Value** vom Typ **T** zugänglich
- kennt über die get-only - Eigenschaften **Previous** bzw. **Next** vom Typ **LinkedListNode<T>** den vorherigen bzw. nächsten Knoten.
- geeignet für Algorithmen, die …
  - häufig innere Elemente einfügen oder entfernen,
  - Elemente nur sequentiell aufsuchen.
- implementiert die Schnittstelle **ICollection<T>**, aber *nicht* die Schnittstelle **IList<T>**, sodass z.B. kein Indexer zur Verfügung steht

Verwaltung einer Liste: LinkedList<T>

- **First, Last** : Properties zeigen auf den ersten bzw. auf den letzten Knoten, bei einer leeren Liste auf **null**.

- **public LinkedListNode<T> AddFirst(T** *element***):** fügt ein neues Objekt der Klasse **LinkedListNode<T>** am Anfang ein und liefert eine Referenz auf das neu erstellte Objekt zurück.

- **public LinkedListNode<T> AddLast(T** *element***):** hängt ein neues Objekt der Klasse **LinkedListNode<T>** am Ende an und liefert eine Referenz auf dieses Objekt.

- **public LinkedListNode<T> AddBefore(LinkedListNode<T>** *node***, T** *element***):** Legt vor dem im ersten Parameter angegebenen Knoten ein neues Objekt der Klasse **LinkedListNode<T>** an und liefert eine Referenz auf den eingefügten Knoten zurück.

- **public LinkedListNode<T> AddAfter(LinkedListNode<T>** *node***, T** *element***):** Nach dem im ersten Parameter angegebenen Knoten wird ein neues Objekt der Klasse **LinkedListNode<T>** angelegt und eine Referenz auf den eingefügten Knoten zurückgeliefert.

- **public void RemoveFirst():** Der erste Knoten wird entfernt.
- **public void RemoveLast():** Der letzte Knoten wird entfernt.
- **public void Remove(LinkedListNode<T>** *element***):** Der per Aktualparameter angegebene Knoten wird entfernt.

Verwaltung eines Sets

- darf im Unterschied zu einer Liste *keine Duplikate* enthalten
- generische Klassen **HashSet<T>** (unsortiert) und **SortedSet<T>**
- nützlich, wenn Operationen für die Modellierung von Mengen im Sinne der Mathematik benötigt werden (z. B. Durchschnitt, Vereinigung oder Differenz von zwei Mengen)
- können Mengenzugehörigkeitsprüfungen sehr schnell ausführen

- Klasse **HashSet<T>**: sinnvoll, wenn für die Instanzen des Elementtyps keine Anordnung besteht bzw. interessiert (sehr schnelle Existenzprüfung)
- Klasse **SortedSet<T>**: sinnvoll, wenn die Instanzen des Elementtyps eine relevante Ordnung besitzen (z. B. Begriffe).

Testprogramm:

- eine Collection mit 20.000 **String**-Objekten füllen
- für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Collection vorhanden sind

Klasse	Füllen (ms)	Existenz-prüfungen
List<T>	4	4530
LinkedList<T>	5	4473
HashSet<T>	6	4
SortedSet<T>	42	43

Verwaltung eines Sets: HashSet

- basiert auf HashTabelle mit offener Sondierung (Key == Value)
- Einstellung der initialen Größe über Konstruktorüberschreibung: public **HashSet(int** *capacity***)**
- erhöhter Speicherbedarf im Vergleich zu List<T>, aber enorme Zeitersparnis, wenn häufige Existenzprüfungen nötig sind

Verwaltung eines Sets: HashSet

public bool Add(T *element***)**: Rückgabewert: wurde Menge durch den Aufruf verändert?

public bool Contains(T *element***)**

public void IntersectWith(IEnumerable<T> *other***):** ändert aktuelles HashSet in die Schnittmenge mit einem anderen

public void UnionWith(IEnumerable<T> *other***):** ändert aktuelles HashSet in Vereinigungsmenge mit einem anderen

public void ExceptWith(IEnumerable<T> *other***):** ändert aktuelles HashSet in Differenzmenge

public bool IsSubsetOf(IEnumerable<T> *other***)**

public bool IsProperSubsetOf(IEnumerable<T> *other***):** mindestens ein Element gehört nicht zu **M** gehört

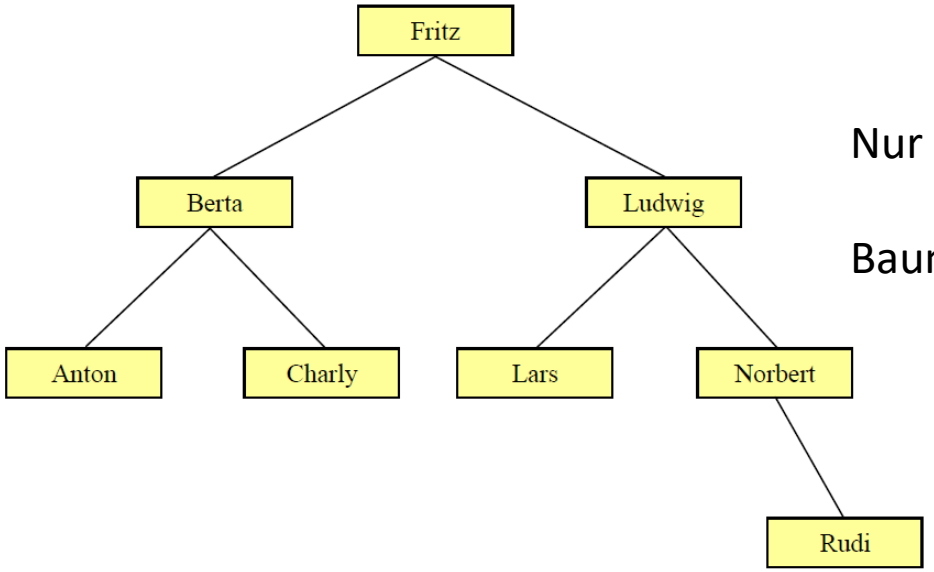
public bool IsSupersetOf(IEnumerable<T> *other***)**

public bool IsProperSupersetOf(IEnumerable<T> *other***):** Obermenge und enthält noch zusätzliche Elemente

Verwaltung eines Sets: SortedSet und balancierte B-Bäume

Existiert über den Elementen einer Menge eine vollständige Ordnung, dann kann man über einen sogenannten *Binärbaum* die Elemente im sortierten Zustand halten, ohne den Aufwand bei den Mengenverwaltungsmethoden (**Add()**, **Contains()** und **Remove()...**) zu groß werden zu lassen.

Balancierter Binärbaum: jeder Knoten, der kein Blattknoten ist, hat genau zwei direkte Nachfolger, der linke hat einen kleineren und der rechte einen höheren Rang:



Nur eine Identitätsprüfung pro Ebene nötig, dann nächste Ebene (log (n))

Baum wird nach jeder Aktion neu balanziert (Rot-Schwarz-Baum)

Verwaltung von (Schlüssel-Wert) – Paaren: Klasse Dictionary<K, V>

Duplikatfreiheit, keine Reihenfolge unterstützt

Instanzmethoden:

public void **Add(K** *key*, **V** *value***):** Aufnahme, wenn Schlüssel noch nicht existiert, sonst ArgumentException.

public bool **ContainsKey(K** *key*): Ist Schlüssel vorhanden? Sehr schnell, auch über Indexer.

public bool **ContainsValue(V** *value*): Ist Element mit dem Wert vorhanden? (kostet viel Zeit)

public bool **TryGetValue(K** *key***, out V** *value***):** Schlüssel vorhanden: Wert wird in *value* geschrieben (true)

public bool **Remove(K** *key***):** Rückgabewert: Kollektion durch den Aufruf geändert?

public void **Clear():** Es werden alle Elemente entfernt.

Übung 1

Sind die folgenden Aussagen richtig oder falsch?

1. Ein Objekt aus der Kollektionsklasse **List<T>** oder **LinkedList<T>** hält seine Elemente in sortiertem Zustand.

2. Die Klasse **HashSet<T>** bietet die beste Leistung bei Existenzprüfungen.

3. Die Klasse **List<T>** beherrscht zum Suchen und Sortieren dieselben Methoden wie ein Array.

4. Die zur Verwaltung von (Schlüssel-Wert) - Elementen dienende Klasse **Dictionary<K, V>** glänzt darin, dass zu einem Schlüssel sehr schnell der zugehörige Wert ermittelt werden kann.

5. Die Kollektionsklassen **SortedList<T>** und **SortedDictionary<K, V>** halten ihre Elemente in sortiertem Zustand.