Container-Klassen

Für die Speicherung einer Vielzahl von Objekten gibt es in den objektorientierten Programmiersprachen wie z.B. C#, Java, C++ u.a. sogenannte Container-Klassen.

Hierbei sind zwei Arten von Container-Klassen zu unterscheiden:

1. nicht-generische Containerklassen
2. generische Containerklassen

Bei den nicht-generischen Container-Klassen können verschiedenartige Objekte im gleichen Container gespeichert werden.

Bei den generischen Container-Klassen wird im Programm vorgegeben, von welchen Datentyp die zu speichernden Objekte sein müssen.

Der wesentliche Nachteil der nicht-generischen Container ist die fehlende Typsicherheit. Wird auf ein Element des Containers zugegriffen, ist der Datentyp des betreffenden Elementes u.U. unbekannt und muss ggf. vor der Nutzung des Objekts erst ermittelt werden.

Eine Gegenüberstellung von nicht-generischen und generischen Container-Klassen findet sich zum Beispiel unter

`http://www.c-sharpcorner.com/UploadFile/736bf5/collection-in-C-Sharp/`

Es sollen folgende Container-Klassen betrachtet werden:

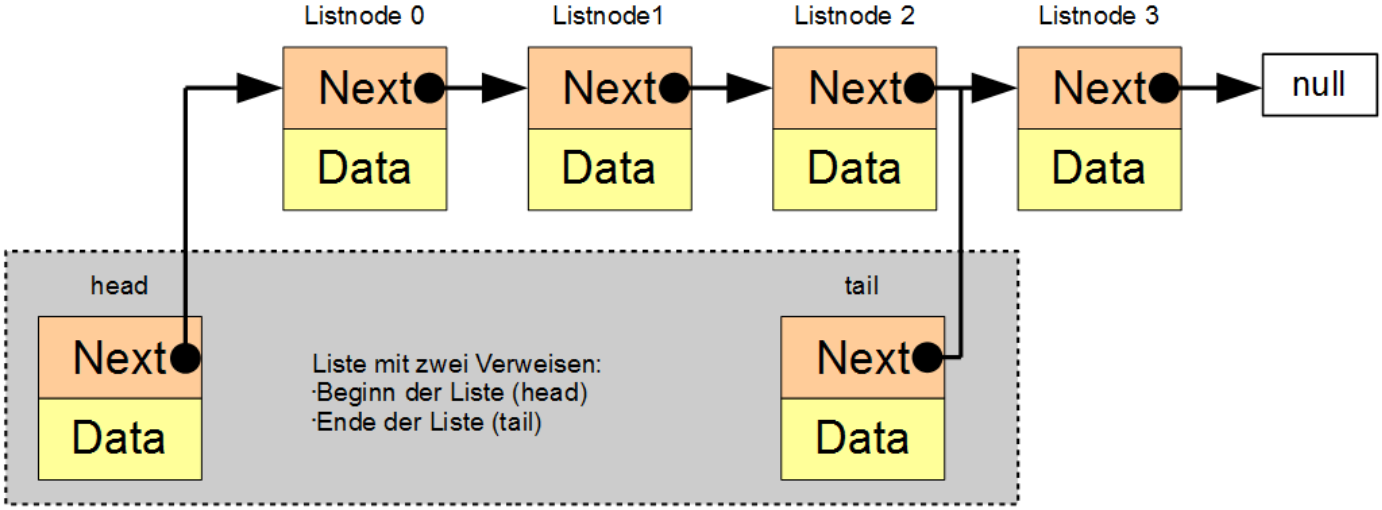
- generische Container-Klassen: **`List<T>`**, **`Stack<T>`**, **`Queue<T>`**
- nicht-generische Container-Klassen: **`ArrayList`**, **`SortedList`**

In der Informatik bezeichnet man diese Klassen auch als **„Abstrakter Datentyp“** (kurz: ADT). Ein abstrakter Datentyp ist ein Verbund von Daten un den zugehörigen Operationen auf diesen Daten. In der Objektorientierung wird dies durch das Klassenkonzept realisiert.

Zunächst aber soll das Prinzip eines generischen Containers am Beispiel einer sogenannten „einfach verketteten Liste“ betrachtet werden.

Prinzip einer generischen verketteten Liste

Eine verkettete Liste mit z.B. vier Listenelementen kann man sich folgendermaßen vorstellen:



Jede Liste ist eine Aneinanderreihung von Listenelementen, zumeist Listnode oder einfach nur

Node genannt. Jede Liste beginnt mit einem Listnode, der meist mit dem Namen **`head`** bezeichnet ist. Das Ende einer solchen Liste wird durch einen Listnode mit dem Namen **`tail`** markiert. Zwischen **`head`** und **`tail`** befinden sich nun die Listenelemente in Form einer Aneinanderreihung. Jedes einzelne Listenelement kann durch einen Index identifiziert werden; das erste Listenelement hat den Index 0, das zweite Listenelement hat den Index 1 usw.

Jeder Listnode enthält zwei Informationen: Einmal enthält er die Daten, die dem betreffenden Listenelement zugewiesen wurden **und** er enthält einen Verweis auf den Ort des nächsten Listenelements. Dieser Verweis ist eine Speicheradresse! Nämlich die Speicheradresse, an der ein Objekt vom Typ Listnode gespeichert ist. Nur das letzte Element in der Liste zeigt auf die Speicheradresse **`null`**.

Die Adresse des ersten Listenelements wird als Verweis in dem Listnode **`head`** gespeichert und die Adresse des letzten Listenelements wird in dem Listnode **`tail`** gespeichert. Sowohl **`head`** als auch **`tail`** können ebenfalls vom Typ Listnode sein; allerdings tragen diese beiden keine Daten![1]

Implementation einer generischen Liste

Prinzipiell besteht eine generische Liste aus zwei Klassen; eine Klasse, die den Aufbau der Liste an sich beschreibt und eine zweite Klasse, die den Aufbau eines Listenelements (Listnode) beschreibt.

Die Liste selbst wird im Folgenden **`SimpleList`** genannt; das Listenelement wird als eine Klasse **`SimpleListNode`** bezeichnet.

Betrachten wir zuerst die Klasse **`SimpleListNode`**, da Objekte dieser Klasse dann in der „umgebenden Struktur“ **`SimpleList`** verwendet werden.

Das Erste, was möglicherweise direkt auffällt, ist der Ausdruck „**`<T>`**“ direkt hinter dem Namen der Klasse.

Um generische Klassen zu erstellen, wird ein generischer Typparameter verwendet. Das von spitzen Klammern eingerahmte **`T`** ist hierbei der Platzhalter für den später jeweils zu verwendenden Typ.

Jedes Listenelement besteht wie zuvor in der Grafik dargestellt und im Text erläutert aus mindestens zwei Komponenten.

Einmal aus den Daten, die das Listenelement aufnehmen soll, hier als Property **`Data`** mit **`get`** und **`set`** implementiert. Natürlich kann dies auch als Attribut realisiert werden.

Und zweitens aus einem Property (oder eben auch als Attribut) namens **`Next`** vom eigenen Datentyp (ebenfalls mit einem Zugriff per **`get`** und **`set`**), wodurch die Verkettung zustande kommt.

```
public class SimpleListNode<T>
{
    public SimpleListNode<T> Next { get; set; }
    public T Data { get; set; }

    public SimpleListNode()
    {
        this.Data = default(T);
        this.Next = null;
    }
    public SimpleListNode(T tmp)
    {
        this.Data = tmp;
        this.Next = null;
    }
}
```

1 Das Design lässt sich auch anders gestalten, nämlich in dem man eine spezielle Klasse Listkopf ohne Datenbereich, dafür aber mit zwei Verweisen, nämlich head und tail entwirft.

Im Beispiel sind zwei Konstruktor-Methoden vorgesehen. Die Konstruktor-Methode ohne Parameter erstellt ein Objekt vom Typ **`SimpleListNode`**, das keine Daten enthält. Der Verweis auf das nächste **`SimpleListNode`**-Objekt ist bei beiden Konstruktoren erstmal **`null`**.

Bei der zweiten Konstruktor-Methode werden die zu speichernden Daten als Parameter (hier: **`tmp`**) übergeben. Der Datentyp dieser Daten bleibt hierbei erstmal unbestimmt und wird durch die Angabe **`T`** quasi als Platzhalter für den unbestimmten Typ verwendet.

Die Struktur zur Verwaltung der **`SimpleListNode`**-Elemente heißt **`SimpleList`**. Auch in dieser Klasse wird der Angabe **`<T>`** als generischer Typparameter verwendet. Hier sind nun auch die Methoden zur Benutzung der Liste enthalten, aber momentan noch nicht dargestellt.

```
public class SimpleList<T>
{
    private SimpleListNode<T> head { get; set; }
    private SimpleListNode<T> tail { get; set; }

    public int Count { get; set; }

    public SimpleList()
    {
        head = new SimpleListNode<T>();
        tail = new SimpleListNode<T>();
    }
}
```

Wie bereits zuvor in der Grafik dargestellt und im Text erläutert, enthält diese Klasse zwei **`SimpleListNode`**-Objektvariablen, die auf das erste (**`head`**) und letzte (**`tail`**) Listenelement zeigen. Beide Objektvariablen sind als Property implementiert. Des Weiteren gibt es das Property **`Count`**, welches die Anzahl der in der Liste enthaltenen Listenelemente zählt.

Die Klasse enthält momentan lediglich eine Konstruktor-Methode zur Erstellung einer neuen leeren Liste.

Hier werden **`head`** und **`tail`** erzeugt. Die verwendete Konstruktor-Methode sorgt dafür, dass in diesen beiden **`SimpleListNode`**-Objekten keine Daten gespeichert sind. Sie werden ja auch nur zur Verwaltung der Liste benötigt! Sowohl **`head`** als auch **`tail`** zeigen initial auf **`null`**.

Um auf die Daten eines bestimmten Listenelementes zugreifen zu können, muss nun eine geeignete Methode implementiert werden. Ein üblicher Name wäre **`getElementAt`**, wobei das gewünschte Element durch eine Nummer, einen Index, anzugeben ist.

Eine Beispiel-Implementation könnte wie rechts dargestellt aussehen.

Die Methode liefert ein **`T`**-Objekt (die Daten des Listenelements) zurück. Die Objektvariable für das betreffende Listenelement hierzu heißt **`rc`** und wird mit **`null`** initialisiert.

Wie dargestellt, sind zwei Fälle zu unterscheiden: Ist die Liste noch leer oder nicht!

Im Fall einer leeren Liste kann kein **`T`**-Objekt zurück geliefert werden, so dass der Default-Wert (**`default(T)`**)zurück geliefert wird.

```
public T getElementAt(int idx)
{
    // Zwei Fälle:
    // 1. Liste ist leer: Rückgabe default(T)
    // 2. Liste hat Elemente: Rückgabe des
    // Element(i)
    SimpleListNode<T> rc = null;
    if (Count > 0)
    {
        rc = head.Next;
        for (int i = 0; i < idx; i++)
        {
            rc = rc.Next;
        }
    }
    return rc.Data;
}
```

Ist die Liste nicht leer (`Count > 0`), wird die Liste solange durchlaufen, bis das Element mit dem gewünschten Index gefunden ist. Die Objektvariable `rc` enthält dann den Verweis auf das entsprechende `SimpleListNode`-Element, dessen Daten `T` zurück geliefert werden.

Um einer Liste Elemente hinzuzufügen, wird üblicherweise eine Add-Methode (oder eine ähnlich benannte Methode) verwendet: `Add(T data):void`.

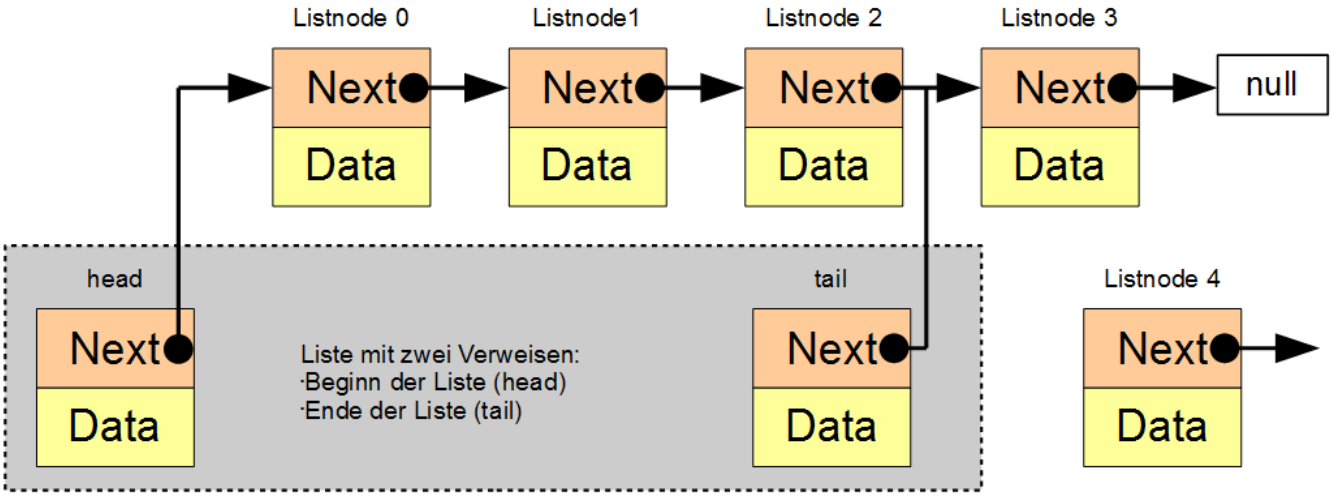
Diese Methode erhält das zu speichernde Datenobjekt, wobei der generische Typparameter `T` zu verwenden ist. Der Parametername sei `data`. Üblicherweise werden **neue Listenelemente am Ende einer Liste** eingefügt.

Aufgaben

1. Die Methode `getElementAt(idx:int):T` ist noch nicht ganz fehlerfrei. Sollte ein Index-Wert als Parameter verwendet werden, der ein nicht vorhandenes Listenelement angibt (beispielweise hat die Liste drei Elemente; es wird als Index der Wert 10 benutzt) kommt es zu einem Fehler.

   Korrigieren Sie diesen Fehler! (Hinweis: es wird der Standardwert von `T` zurück gegeben)

2. Analysieren Sie die möglichen Fälle beim Einfügen eines neuen Listenelements am Ende der Liste, die bei der Implementation der `Add`-Methode zu berücksichtigen sind.
   In der Grafik soll das Listenelement 4 der Liste hinzu gefügt werden.



   Ändern Sie die Skizze so ab, dass das neue Listenelement in der Liste enthalten ist.

   Entwerfen Sie für die `Add`-Methode ein Struktogramm.
   Implementieren Sie die Methode `Add(T data):void`.

3. Analysieren Sie die zu berücksichtigenden Fälle beim Entfernen eines Listenelements mit dem Index `idx`.
   Entwerfen Sie für eine Methode `removeElementAt(idx:int):T` ein Struktogramm.
   Das entfernte Listenelement soll hierbei zurück geliefert werden.
   Implementieren Sie diese Methode.

4. Erweitern Sie das Test-Programm unter Verwendung der zusätzlichen Methoden und überprüfen Sie deren Funktionsfähigkeit mit verschiedenen Werten für den Index; insbesondere für das erste und letzte Element.