Skript: Dynamische Datentypen und Generische Kollektionen in C#

1. Theoretische Grundlagen: Statische vs. Dynamische Speicherverwaltung

In der Informatik ist die Wahl der Datenstruktur entscheidend für die Effizienz eines Algorithmus. Bisher haben wir das **Array** verwendet. Arrays sind sogenannte **statische Datenstrukturen**.

1.1 Das Problem der Statik

Ein Array wird im Arbeitsspeicher (RAM) als ein **kontinuierlicher Block** alokiert.

- **Vorteil:** Da die Elemente direkt hintereinander liegen, kann die CPU extrem effizient darauf zugreifen (Cache Locality). Der Zugriff auf `Array[i]` erfolgt in $O(1)$.
- **Nachteil:** Die Größe ist fest. Will man ein Array vergrößern, ist dies physikalisch meist nicht möglich, da der Speicher hinter dem Block bereits von anderen Variablen belegt sein könnte.

1.2 Implementierungsvergleich: Array vs. List

Um dem Komfortgewinn dynamischer Listen zu verstehen, vergleichen wir den Prozess des "Hinzufügens eines Elements zu einer vollen Struktur".

Der "Manuelle Weg" (Array)

Möchte man einem Array der Größe N ein $(N+1)$-tes Element hinzufügen, muss der Programmierer die Speicherverwaltung selbst übernehmen.

```
// Ausgangslage: Ein volles Array
int[] zahlenArray = new int[3] { 10, 20, 30 };

// ZIEL: Die Zahl 40 hinzufügen.

// 1. Neues, größeres Array anlegen (Größe + 1 oder Verdopplung)
int[] tempArray = new int[4];

// 2. Daten kopieren (Rechenintensiv: O(n))
for (int i = 0; i < zahlenArray.Length; i++)
{
    tempArray[i] = zahlenArray[i];
}

// 3. Neuen Wert eintragen
tempArray[3] = 40;

// 4. Referenz austauschen (Altes Array wird vom Garbage Collector gelöscht)
zahlenArray = tempArray;
```

Der "Dynamische Weg" (List)

Die `List<T>` kapselt genau diesen Algorithmus. Für den Entwickler reduziert sich der Aufwand auf eine Zeile.

```
// Ausgangslage
List<int> zahlenListe = new List<int>() { 10, 20, 30 };

// ZIEL: Die Zahl 40 hinzufügen.
zahlenListe.Add(40);

// Was passiert intern?
// Die Liste prüft ihre "Capacity". Ist sie voll, führt sie
// die Schritte 1-4 des Array-Beispiels automatisch durch.
```

2. Generics (`<T>`)

Die Klasse `List<T>` und alle folgenden Strukturen befinden sich im Namespace `System.Collections.Generic`. Das `<T>` ist ein Platzhalter (Typ-Parameter). Es erlaubt uns, Klassen zu schreiben, die typsicher sind, ohne für jeden Datentyp eine eigene Klasse (z.B. `IntList`, `StringList`, `MyDoubleList`) schreiben zu müssen.

- **Typsicherheit:** Der Compiler verhindert, dass Sie einen `string` in eine `List<int>` stecken.
- **Performance:** Es verhindert "Boxing" (das teure Verpacken von Werten in Objekte), was bei alten C#-Versionen (`ArrayList`) üblich war.

3. Die dynamische Liste (`List<T>`)

Die Liste ist ein **dynamisches Array**. Sie bietet wahrhaften Zugriff über einen Index (0 bis `Count-1`).

3.1 Interne Funktionsweise

Die Liste verwaltet zwei Größen:

1. **Count:** Die Anzahl der Elemente, die der Benutzer sieht.
2. **Capacity:** Die Größe des internen Arrays.
   - Wenn `Count == Capacity` und ein `Add` erfolgt, wird die `Capacity` verdoppelt (z.B. $4 \rightarrow 8 \rightarrow 16$). Dies garantiert eine amortisierte Laufzeit von $O(1)$ für das Hinzufügen.

3.2 Methoden-Referenz (List)

Methode / Eigenschaft	Beschreibung	Laufzeit
`Add(T item)`	Fügt ein Element am Ende der Liste hinzu.	$O(1)$ (amortisiert)
`Insert(int index, T item)`	Fügt ein Element an einer beliebigen Stelle ein. Alle nachfolgenden Elemente werden verschoben.	$O(n)$
`Remove(T item)`	Sucht das erste Vorkommen des Elements und löscht es. Nachfolgende Elemente rücken auf.	$O(n)$
`RemoveAt(int index)`	Löscht das Element am angegebenen Index.	$O(n)$
`Contains(T item)`	Prüft linear, ob das Element existiert. Gibt `true`/`false` zurück.	$O(n)$
`Clear()`	Entfernt alle Elemente (`Count` wird 0), behält aber meist die `Capacity` (Speicher) bei.	$O(n)$ (für Referenzen)
`Sort()`	Sortiert die Liste (Standardmäßig aufsteigend).	$O(n \log n)$
`ToArray()`	Erstellt eine Kopie der Liste als klassisches Array	$O(n)$

3.3 Code-Beispiel

```
List<string> namen = new List<string>();

// Füllen
namen.Add("Alice");
namen.Add("Bob");
namen.Insert(1, "Carlos"); // Schiebt 'Bob' auf Index 2

// Zugriff & Ausgabe
Console.WriteLine(namen[0]); // "Alice"

// Iteration
foreach (string s in namen) {
    Console.WriteLine(s);
}

// Suche
if (namen.Contains("Bob")) {
    Console.WriteLine("Bob ist da.");
}
```

4. Das Dictionary (`Dictionary<TKey, TValue>`)

Das Dictionary ist ein **assoziativer Speicher**. Es mappt einen eindeutigen Schlüssel (Key) auf einen Wert (Value).

4.1 Theorie: Hashing

Anders als Listen, die linear durchsucht werden müssen, nutzt das Dictionary eine **Hash-Funktion**. Diese mathematische Funktion errechnet aus dem Key direkt die Speicheradresse.

- **Konsequenz:** Der Zugriff auf einen Eintrag dauert immer gleich lang, egal ob das Dictionary 10 oder 1.000.000 Einträge hat ($O(1)$).
- **Bedingung:** Keys müssen eindeutig sein (z.B. ID, E-Mail, Matrikelnummer).

4.2 Methoden-Referenz (Dictionary)

Methode / Eigenschaft	Beschreibung
`Add(TKey key, TValue value)`	Fügt ein neues Paar hinzu. Wirft eine `Exception`, wenn der Key schon existiert.
`[key] = value` (Indexer)	Schreibend: Fügt hinzu oder überschreibt, falls Key existiert. Lesend: Gibt Wert zurück (wirft Fehler, wenn Key fehlt).
`ContainsKey(TKey key)`	Prüft extrem schnell, ob ein Schlüssel existiert (`true`/`false`).
`TryGetValue(key, out val)`	Versucht sicher, einen Wert zu holen. Gibt `false` zurück, statt abzustürzen, wenn der Key fehlt.
`Remove(TKey key)`	Löscht den Eintrag mit dem entsprechenden Schlüssel.
`Keys` / `Values`	Liefert eine Liste aller Schlüssel bzw. aller Werte zurück.

4.3 Code-Beispiel

```
var telefonbuch = new Dictionary<string, string>();

// Hinzufügen
telefonbuch.Add("Polizei", "110");
telefonbuch["Feuerwehr"] = "112"; // Sicherer Weg (Upsert)

// Zugriff
// Console.WriteLine(telefonbuch["Pizza"]); // WÜRDE ABSTÜRZEN! Key fehlt.

// Sicherer Zugriff
if (telefonbuch.ContainsKey("Polizei")) {
    Console.WriteLine("Nummer: " + telefonbuch["Polizei"]);
}

// Iteration (über KeyValuePair)
foreach (KeyValuePair<string, string> eintrag in telefonbuch) {
    Console.WriteLine($"{eintrag.Key} ruft man unter {eintrag.Value}");
}
```

5. Die Menge (`HashSet<T>`)

Das `HashSet<T>` ist die Implementierung einer mathematischen Menge. Technisch ist es ein Dictionary, das nur Keys und keine Values speichert.

5.1 Charakteristik

1. **Einzigartigkeit:** Duplikate werden stillschweigend ignoriert.
2. **Ungeordnet:** Es gibt keinen Index (kein `set[0]`). Die Reihenfolge ist zufällig.
3. **Performance:** `Contains` ist $O(1)$. In einer `List` ist `Contains` $O(n)$. Bei 10.000 Elementen ist das HashSet also bis zu 10.000-mal schneller beim Suchen.

5.2 Methoden-Referenz (HashSet)

Methode	Beschreibung
`Add(T item)`	Gibt `true` zurück, wenn das Element neu war. Gibt `false` zurück, wenn es schon da war.
`Remove(T item)`	Entfernt ein Element.
`Contains(T item)`	Prüft auf Existenz (Sehr schnell).
`UnionWith(IEnumerable<T>)`	Vereinigt das Set mit einer anderen Sammlung ($A \cup B$).
`IntersectWith(IEnumerable<T>)`	Behält nur die Schnittmenge ($A \cap B$).
`ExceptWith(IEnumerable<T>)`	Entfernt alle Elemente, die in der anderen Sammlung sind ($A \setminus B$).

5.3 Code-Beispiel (Mengenlehre)

```
HashSet<int> gruppeA = new HashSet<int>() { 1, 2, 3, 4 };
HashSet<int> gruppeB = new HashSet<int>() { 3, 4, 5, 6 };

// Duplikate testen
bool hinzugefuegt = gruppeA.Add(1); // false, da 1 schon existiert

// Schnittmenge bilden (Wer ist in beiden Gruppen?)
gruppeA.IntersectWith(gruppeB);
// gruppeA enthält jetzt nur noch {3, 4 }

foreach (int z in gruppeA) {
    Console.Write(z + " ");
}
```

6. Spezialisierte Listen: Queue und Stack

Manchmal benötigen Algorithmen eine strengere Ordnung als Listen. Queue und Stack schränken den Zugriff ein, um logische Fehler zu vermeiden. Sie erlauben keinen Zugriff auf die Mitte der Liste (kein Indexer `[i]`).

6.1 Die Queue (Warteschlange)

Prinzip: FIFO (First-In, First-Out). Stellen Sie sich eine Schlange an der Kasse vor. Wer zuerst kommt, wird zuerst bedient. Neue Elemente stellen sich hinten an.

Methoden-Referenz:

- `Enqueue(T item)`: Fügt ein Element am **Ende** hinzu.
- `Dequeue()`: Entfernt das **vorderste** Element und gibt es zurück. Wirft Fehler, wenn leer.
- `Peek()`: Liefert das vorderste Element, ohne es zu entfernen.

Code-Beispiel:

```
Queue<string> tasks = new Queue<string>();

tasks.Enqueue("Rechnung schreiben"); // 1.
tasks.Enqueue("Kaffee trinken");     // 2.

// Verarbeiten
while (tasks.Count > 0) {
    string currentTask = tasks.Dequeue(); // Holt "Rechnung schreiben"
    Console.WriteLine("Erledige: " + currentTask);
}
```

6.2 Der Stack (Stapel)

Prinzip: LIFO (Last-In, First-Out). Wie ein Stapel Teller oder Karten. Man kann nur oben etwas drauflegen und nur von oben etwas wegnehmen. Das unterste Element ist unerreichbar, bis alles darüber weg ist.

Methoden-Referenz:

- `Push(T item)`: Legt ein Element **oben** auf den Stapel.
- `Pop()`: Nimmt das **oberste** Element weg und gibt es zurück.
- `Peek()`: Schaut sich das oberste Element an, ohne es zu entfernen.

Code-Beispiel (Rückgängig-Funktion):

```
Stack<string> aktionen = new Stack<string>();

aktionen.Push("Wort tippen");
aktionen.Push("Wort fett markieren");
aktionen.Push("Wort löschen"); // Das war ein Fehler!

// Undo (Rückgängig)
string letzteAktion = aktionen.Pop(); // "Wort löschen" wird entfernt
Console.WriteLine("Rückgängig gemacht: " + letzteAktion);

// Nächster Status oben
Console.WriteLine("Aktueller Stand: " + aktionen.Peek()); // "Wort fett markieren"
```

7. Zusammenfassung: Welche Struktur wann?

Anforderung	Datenstruktur	Warum?
Standardfall	`List<T>`	Flexibel, Index-Zugriff, einfach zu nutzen.
Suche nach ID / Schlüssel	`Dictionary<K,V>`	Extrem schneller Zugriff ($O(1)$) statt Suchen ($O(n)$).
Duplikate verhindern	`HashSet<T>`	Prüft automatisch und schnell auf Einzigartigkeit.
Reihenfolge bewahren (FIFO)	`Queue<T>`	Warteschlangen, Puffer, Druckaufträge.
Historie / Zurück (LIFO)	`Stack<T>`	Undo-Funktionen, Navigations-Verlauf, Rekursion.