

Skript: Strukturierte Datentypen in C# (Arrays und Enums)

1. Problemstellung und Motivation

In den ersten Wochen der Programmierung werden in der Regel einfache Datentypen (sogenannte *primitive types*) wie `int`, `double`, `bool` oder `string` verwendet. Diese Variablen zeichnen sich dadurch aus, dass sie **atomar** sind – eine Variable speichert genau einen Wert.

In realen Softwareprojekten besteht jedoch fast immer die Notwendigkeit, Gruppen von zusammengehörigen Daten zu verwalten. Beispiele hierfür sind:

- Eine Liste aller Matrikelnummern eines Semesters.
- Die Pixeldaten eines Bildes.
- Der Verlauf von Temperaturmessungen einer Wetterstation über einen Monat.

Das Skalierbarkeitsproblem

Ein naiver Ansatz wäre, für jeden dieser Werte eine eigene Variable anzulegen (`note1`, `note2`, `note3` usw.). Dieser Ansatz stößt jedoch schnell an seine Grenzen:

1. **Schreibaufwand:** Bei 100 oder 1000 Datensätzen ist der Quellcode nicht mehr handhabbar.
2. **Fehlende Flexibilität:** Algorithmen zur Verarbeitung der Daten (z. B. "Berechne den Durchschnitt") müssen für jede Anzahl von Variablen neu geschrieben werden.
3. **Wartbarkeit:** Der Code wird unübersichtlich und fehleranfällig.

Die Lösung in der Informatik sind **strukturierte Datentypen**, die es erlauben, eine Sammlung von Werten unter einem einzigen Bezeichner zu verwalten. Die fundamentalste dieser Strukturen ist das **Array** (deutsch: Feld).

2. Das Array (Feld)

2.1 Definition und theoretische Grundlagen

Ein Array ist eine Datenstruktur, die eine **feste Anzahl** von Elementen des **gleichen Datentyps** in einer linearen Abfolge speichert.

Daraus ergeben sich drei wesentliche Eigenschaften:

1. **Homogenität:** Ein Array ist typgebunden. Ein `int`-Array kann ausschließlich ganze Zahlen speichern. Es ist nicht möglich, Datentypen innerhalb eines Arrays zu mischen (z. B. Strings und Zahlen in demselben Array).
2. **Statische Speicherallokation:** Die Größe des Arrays muss zum Zeitpunkt der Erstellung (Initialisierung) festgelegt werden. In C# kann die Größe eines einmal erstellten Arrays nicht mehr verändert werden. Möchte man das Array vergrößern, muss ein neues, größeres Array erstellt und die alten Daten kopiert werden.
3. **Index-Zugriff:** Jedes Element im Array besitzt eine eindeutige Position, den sogenannten **Index**.

2.2 Speicherorganisation

Das Verständnis der Speicherverwaltung ist essenziell für das Verständnis der Performance von Arrays. Wenn ein Array erstellt wird, reserviert das Betriebssystem einen **zusammenhängenden Speicherblock** (Contiguous Memory).

Da der Datentyp homogen ist, ist dem Computer bekannt, wie viel Speicherplatz jedes einzelne Element benötigt (z. B. 4 Byte für einen `int` in einem 32-Bit System). Dadurch kann der Computer die Speicheradresse jedes beliebigen Elements mathematisch berechnen, ohne die Liste durchsuchen zu müssen:

$$\text{Adresse} = \text{Startadresse} + (\text{Index} \times \text{Größe des Datentyps})$$

Dies ermöglicht den sogenannten **Random Access** (wahlfreien Zugriff) in konstanter Zeit ($O(1)$), was Arrays zu einer der effizientesten Datenstrukturen für Lesezugriffe macht.

3. Eindimensionale Arrays (1D)

3.1 Deklaration

Die Deklaration macht dem Compiler lediglich bekannt, dass eine Variable existieren soll, die auf ein Array verweist. Es wird zu diesem Zeitpunkt noch kein Speicher für die Daten selbst reserviert.

```
// Syntax: Datentyp[] Variablename;
int[] noten;
string[] namen;
```

Die eckigen Klammern `[]` hinter dem Datentyp signalisieren, dass es sich um ein Array handelt.

3.2 Initialisierung (Instanzierung)

Um das Array tatsächlich nutzen zu können, muss Speicher auf dem **Heap** (dynamischer Speicher) reserviert werden. Dies geschieht mit dem Schlüsselwort `new`. Hierbei **muss** die Größe des Arrays festgelegt werden.

```
// Reserviert Speicher für exakt 5 Integer-Werte
noten = new int[5];
```

Standardwerte (Default Values): Nach der Initialisierung sind die Plätze im Array nicht leer, sondern mit dem Standardwert des jeweiligen Datentyps gefüllt:

- Numerische Typen (`int`, `double`, etc.): `0`
- `bool`: `false`
- Referenztypen (`string`, Objekte): `null`

3.3 Kombinierte Initialisierung

In der Praxis werden Deklaration und Initialisierung oft zusammengefasst. Wenn die konkreten Werte bereits zum Entwicklungszeitpunkt bekannt sind, bietet C# die **Array-Initialisierungs-Syntax** (Array Initializer) an. Hierbei ermittelt der Compiler die notwendige Größe des Arrays automatisch anhand der Anzahl der übergebenen Elemente.

```
// Variante A: Größe explizit, Werte implizit (0)
int[] zahlen = new int[3];

// Variante B: Größe implizit, Werte explizit
// Erstellt ein Array der Länge 3 mit den Werten "Anna", "Ben", "Clara"
string[] namen = { "Anna", "Ben", "Clara" };
```

3.4 Zugriff auf Elemente

Der Zugriff auf die Daten erfolgt über den Namen der Variable und den Index in eckigen Klammern.

Wichtig: Die 0-basierte Indizierung In fast allen modernen Programmiersprachen (C#, Java, C++, Python) beginnt die Zählung beim Index `0`.

- Das 1. Element hat den Index `0`.
- Das 2. Element hat den Index `1`.
- Das n -te Element hat den Index $n - 1$.

```
string[] namen = { "Anna", "Ben", "Clara" };

// Lesen
Console.WriteLine(namen[0]); // Ausgabe: Anna

// Schreiben (Überschreiben des alten Wertes)
namen[1] = "Bob"; // Aus "Ben" wird "Bob"
```

Fehlerbehandlung: `IndexOutOfRangeException` Ein sehr häufiger Fehler ist der Zugriff auf einen Index, der nicht existiert. Beispiel: Bei einem Array der Länge 3 sind die gültigen Indizes 0, 1 und 2. Ein Zugriff auf `namen[3]` führt zu einer `IndexOutOfRangeException`. Dieser Fehler tritt erst zur Laufzeit auf und führt zum Absturz des Programms, wenn er nicht abgefangen wird.

4. Iteration: Durchlaufen von Arrays

Um Operationen auf allen Elementen eines Arrays durchzuführen (z. B. Summenbildung, Suche, Ausgabe), werden Schleifen verwendet.

4.1 Die `for`-Schleife

Die `for`-Schleife ist die klassische Methode zur Iteration. Sie ist besonders nützlich, wenn:

- Der Index innerhalb der Schleife benötigt wird (z. B. "Gib jeden zweiten Wert aus" oder "Setze Wert an Stelle X").
- Das Array verändert werden soll (Schreibzugriff).

Jedes Array besitzt die Eigenschaft `.Length` (ohne Klammern), die die Gesamtzahl der Elemente als Integer zurückgibt.

```
int[] werte = { 10, 20, 30, 40 };

// Wichtig: Bedingung ist i < werte.Length (NICHT <=)
for (int i = 0; i < werte.Length; i++)
{
    // Schreibzugriff: Wir verdoppeln jeden Wert im Array
    werte[i] = werte[i] * 2;

    Console.WriteLine($"Index {i}: {werte[i]}");
}
```

4.2 Die `foreach`-Schleife

Die `foreach`-Schleife abstrahiert den Index. Sie bedeutet semantisch: "Nimm dir nacheinander jedes Element aus der Sammlung".

Vorteile:

- Höhere Lesbarkeit.
- Schutz vor `IndexOutOfRangeException` (man kann nicht versehentlich zu weit zählen).

Nachteile:

- **Read-Only:** Die Iterationsvariable (im Beispiel `name`) ist eine Kopie des Wertes. Eine Zuweisung an `name` ändert nicht das Element im Array.
- Der Index ist nicht bekannt (man weiß nicht, ob man gerade beim 1. oder 5. Element ist).

```
string[] namen = { "Anna", "Ben", "Clara" };

foreach (string name in namen)
{
    // name = "X"; // Dies würde NICHT das Array ändern!
}
```

5. Mehrdimensionale Arrays (2D)

Während ein 1D-Array einer Liste entspricht, entspricht ein 2D-Array einer Tabelle oder Matrix mit Zeilen und Spalten.

5.1 Syntax und Deklaration

In C# werden mehrdimensionale Arrays (Rectangular Arrays) durch Kommas innerhalb der eckigen Klammern definiert. Ein Komma `[,]` bedeutet 2 Dimensionen, zwei Kommas `[,,]` bedeuten 3 Dimensionen usw.

```
// Deklaration eines 2D-Arrays (3 Zeilen, 4 Spalten)
int[,] matrix = new int[3, 4];
```

Auch hier ist eine direkte Initialisierung möglich. Dabei werden geschweifte Klammern verschachtelt:

```
int[,] spielfeld =
{
    { 1, 2, 3 }, // Zeile 0
    { 4, 5, 6 } // Zeile 1
}; // Ergebnis: 2 Zeilen, 3 Spalten
```

5.2 Zugriff

Der Zugriff erfolgt über die Angabe beider Koordinaten: `[Zeile, Spalte]`.

```
int wert = spielfeld[1, 2]; // Zeile 1, Spalte 2 -> Wert: 6
```

5.3 Iteration über 2D-Arrays

Um alle Elemente zu besuchen, werden **verschachtelte Schleifen** benötigt. Die äußere Schleife iteriert meist über die Zeilen, die innere über die Spalten.

Da `.Length` die Gesamtzahl aller Elemente zurückgibt (bei 2×3 also 6), muss für die Schleifengrenzen die Methode `.GetLength(dimension)` verwendet werden:

- `.GetLength(0)`: Länge der 1. Dimension (Zeilenzahl).
- `.GetLength(1)`: Länge der 2. Dimension (Spaltenzahl).

```
for (int zeile = 0; zeile < spielfeld.GetLength(0); zeile++)
{
    for (int spalte = 0; spalte < spielfeld.GetLength(1); spalte++)
    {
        Console.Write(spielfeld[zeile, spalte] + " ");
    }
    Console.WriteLine(); // Umbruch am Ende jeder Zeile
}
```

6. Enums (Aufzählungstypen)

6.1 Das Anti-Pattern: "Magic Numbers"

oft müssen Zustände oder Optionen in Code abgebildet werden (z. B. Wochentage, Bestellstatus, Farben). Ein Anfängerfehler ist die Verwendung von bloßen Zahlen ("Magic Numbers"):

```
int status = 2; // Was bedeutet 2? "Versendet"? "Abgelehnt"?
```

Probleme:

1. **Schlechte Lesbarkeit:** Ein anderer Programmierer muss raten oder in der Dokumentation suchen.
2. **Mangelnde Typsicherheit:** Man könnte versehentlich `status = 99` setzen, obwohl dieser Zustand nicht existiert.

6.2 Lösung durch Enums

Ein `enum` (Enumeration) ist ein benutzerdefinierter Datentyp, der eine feste Menge von benannten Konstanten bündelt.

Definition: Enums werden meist auf Klassenebene definiert (außerhalb von Methoden). Innerhalb der geschweiften Klammern können wir beliebige Konstanten eintragen, die als mögliche Zustände genutzt werden. Der enum-Namen darf hierbei als neuer Datentyp agieren.

```
enum Bestellstatus
{
    Neu,           // Interne Wert: 0
    InBearbeitung, // Interne Wert: 1
    Versendet,     // Interne Wert: 2
    Storniert      // Interne Wert: 3
}
```

Obwohl Enums intern als Ganzzahlen (Standard: `int`) gespeichert werden, arbeiten wir im Code mit den sprechenden Namen.

6.3 Verwendung

Variablen vom Typ des Enums können nur die definierten Werte annehmen. Dies garantiert, dass keine ungültigen Zustände entstehen.

```
// Deklaration und Zuweisung
Bestellstatus aktuellerStatus = Bestellstatus.Neu;
```

```
// Verwendung im Switch (sehr gebräuchlich)
switch (aktuellerStatus)
{
    case Bestellstatus.Neu:
        // Logik für neue Bestellungen
        break;
    case Bestellstatus.Versendet:
        // Logik für Versand
        break;
    default:
        // Fehlerbehandlung
        break;
}
```

Die Verwendung von Enums macht den Code **selbst dokumentierend**. Statt `if (day == 6)` liest man `if (day == Wochentag.Samstag)`, was die Intention des Programmierers sofort klar macht.