

Microsoft  
**Visual C# 2008**

Datenfelder (Arrays)

## **Inhaltsverzeichnis**

Vorwort .....	3
Datenfelder (Arrays).....	4
Deklaration und Initialisierung .....	4
Literale Initialisierung .....	5
Werte auslesen.....	5
Datenfelder sind Referenztypen .....	6
Werttypen .....	6
Referenztypen .....	6
Mehrdimensionale Datenfelder .....	7
Literale Initialisierung .....	7
Werte auslesen.....	8
Initialisierung zur Laufzeit .....	8
Größe der Dimensionen ermitteln .....	9
Gesamtanzahl der Elemente ermitteln.....	10
Letzten Indexwert eines Datenfeldes ermitteln .....	10
Verzweigte Datenfelder .....	11

## Vorwort

Wer kennt das nicht: Man braucht mal eben eine kurze Info zu einem bestimmten Thema und hofft, dass im Internet etwas zu finden sein müsste. Dann fängt man an zu surfen und surft und surft und surft. Oft braucht man buchstäblich Stunden, um alle benötigten Informationen zu einem bestimmten Thema zu finden. Ist Ihnen das nicht auch schon so gegangen?

Um Ihnen die Arbeit etwas zu erleichtern, habe ich mir die Mühe gemacht, Informationen zu einigen ausgewählten Themen in sogenannte Kurzdokumentationen zusammenzustellen. Dabei darf nicht unerwähnt bleiben, dass ich natürlich auch nicht ohne Fehler bin und ebenfalls einige Dinge nicht weiß. Falls Ihnen beim Studium einer Kurzdokumentation also irgendwelche Lücken oder Irrtümer begegnen, würde ich mich sehr über einen kurzen Hinweis freuen, damit ich die Kurzdokumentationen auf den neuesten Stand bringen kann.

Nun bleibt mir nur noch, Ihnen viel Freude beim Lesen der vorliegenden Unterlage zu wünschen.

Viele Grüße,  
Michael Scholz

## Datenfelder (Arrays)

Was ist ein Datenfeld?

Ein Datenfeld (Array) ermöglicht es, eine nahezu beliebige Anzahl von Variablen gleichen Datentyps in einer Art Container zu speichern.

Merke: Datenfelder sind Container für Variablen des gleichen Datentyps

Und wozu benötigt man Datenfelder?

Datenfelder (als Container von Variablen) werden zum Beispiel benötigt, wenn die gleiche Operation auf alle (oder mehrere) der im Container gespeicherten Variablen ausgeführt werden sollen.

## Deklaration und Initialisierung

Bevor man ein Datenfeld benutzen kann muss man es wie jede Variable deklarieren. Wie eine Variable hat auch das Datenfeld einen Datentyp den man bei der Deklaration voranstellt.

Damit der C#-Compiler erkennt, dass es sich bei der Variablen um ein Datenfeld handelt, wird hinter dem Datentyp eine Klammerung geschrieben.

```
int[] Datenfeld;
```

Mit dieser Anweisung wird ein Datenfeld deklariert, das Ganzzahlen (Integer) beschreibt. Um wie viele Einträge es sich dabei handelt, wurde nicht festgelegt.

Um mit dem Datenfeld arbeiten zu können, braucht der C#-Compiler noch die Anzahl an Variablen, die das Datenfeld speichern soll. Dazu muss das Datenfeld initialisiert werden.

```
int[] Datenfeld;  
Datenfeld = new int[5];
```

Wie man am Schlüsselwort `new` erkennen kann, behandelt die Laufzeitumgebung ein Datenfeld als Objekt. Datenfelder werden also genauso initialisiert, wie Objekte initialisiert werden.

Merke: Datenfelder sind Objekte

Die Anzahl der Elemente, die in einem Datenfeld gespeichert werden können, also die Größe des Arrays, geht aus der Zahlenangabe in den eckigen Klammern hervor.

Merke: Die Angabe in den eckigen Klammern der Initialisierung ist immer eine Zahl vom Typ `int`.

Alternativ kann die Deklaration und Initialisierung in einer einzigen Zeile vorgenommen werden.

```
int[] Datenfeld = new int[5];
```

Nach der Deklaration sind alle Elemente des Datenfelds mit dem Wert 0 vorbelegt.

## Literale Initialisierung

Wenn Sie schon wissen, welche Daten sie in das Datenfeld aufnehmen möchten, können sie auch die so genannte literale Initialisierung durchführen. Dabei werden die Daten in geschweiften Klammern angegeben.

```
int[] Datenfeld = new int[5]{23,9,7,4,17};
```

Da der C#-Compiler bei der literalen Initialisierung selbst erkennen kann, um wie viele Elemente es sich handelt, kann die Größenangabe in den eckigen Klammern weggelassen werden.

```
int[] Datenfeld = new int[]{23,9,7,4,17};
```

Da der C#-Compiler auch schon weiß, um welchen Datentyp es sich handelt, benötigen Sie auch die Angabe des Datentyps vor den geschweiften Klammern nicht.

```
int[] Datenfeld = {23,9,7,4,17};
```

Natürlich müssen Sie darauf achten einem Datenfeld vom Typ `int` keine Werte zu übergeben, die von einem anderen Datentyp sind.

**Merke:** Die literale Initialisierung setzt voraus, dass den Elementen ein gültiger Wert übergeben wird!

## Werte auslesen

Nun wissen wir, wie wir die Daten in das Datenfeld bekommen. Aber wie lesen wir sie wieder aus?

Dazu muss man erst einmal wissen, dass die Elemente eines Datenfelds bei der Initialisierung automatisch durchnummeriert werden. Das erste Element hat dabei den Wert 0. Diese Nummerierung Wert nennt man auch INDEX.

**Merke:** Das erste Element eines Datenfelds hat den Index 0

Wie greifen wir aber auf das letzte Element im Datenfeld zu?

Wenn die Nummerierung der Elemente mit 0 beginnt, ergibt sich logischerweise ff. Formel zur Ermittlung des letzten Elements im Datenfeld:

**Merke:** Das letzte Element eines Datenfelds hat den Index: Anzahl der Elemente – 1

Ein Datenfeld, das mit fünf Elementen belegt ist ...

```
int[] Datenfeld = new int[5];
```

... enthält also die folgenden fünf Elemente, die von 0 bis 4 durchnummeriert sind:

```
Datenfeld[0], Datenfeld[1], Datenfeld[2], Datenfeld[3], Datenfeld[4]
```

Sowohl der lesende also der schreibende Zugriff erfolgt über die Angabe der Indexnummer.

Möchten wir beispielsweise das erste Element des Datenfelds "Gewinnzahlen" auf die Zahl 27 setzen und anschließend in einer MessageBox ausgeben, ginge das wie folgt:

```
Gewinnzahlen[0] = 27;
MessageBox.Show(Gewinnzahlen[0]);
```

## Datenfelder sind Referenztypen

Wir hatten bereits gehört, dass Datenfelder vom C#-Compiler als Objekte behandelt werden. Objekte sind in .NET so genannte Referenztypen. Referenztypen unterscheiden sich von Werttypen durch die Art und Weise, wie die Daten im Arbeitsspeicher des Computers abgelegt werden.

### Werttypen

Erstellt man eine Werttyp-Variable, wird im Arbeitsspeicher Platz für den zu speichernden Wert reserviert. Weist man der Variablen dann einen Wert zu, wird dieser im dafür reservierten Platz gespeichert.

Erstellt man nun eine weitere Werttyp-Variable, wird für diese ebenfalls Platz im Arbeitsspeicher reserviert. Weist man dieser zweiten Variablen dann als Wert die erste Variable zu, wird der Wert der ersten Variablen kopiert, und die Kopie in den Speicherbereich, der zweiten Variable gelegt.

```
int a = 5;
int b = a;           // b bekommt den Wert 5 der Variablen a zugewiesen
                    // Dieser wird allerdings in einem eigenen Speicher-
                    // bereich abgelegt. Der Wert wird also kopiert.
                    // Speicherbereich b ist völlig unabhängig von
                    // Speicherbereich a.

a = 6;              // Nun wird der Wert der Variablen a geändert
                    // Der Wert in b bleibt dabei gleich (also 5), da
                    // b über einen eigenen Speicherbereich verfügt
```

### Referenztypen

Anders verhält es sich mit Referenztypen. Legt man eine Referenztyp-Variable an, werden im Arbeitsspeicher zwei Bereiche reserviert. In einem Bereich werden die Daten gespeichert, in dem anderen speichert der C#-Compiler die Adresse des Speicherbereiches, indem die Daten liegen.

Da dieser zweite Speicherbereich lediglich die Zugangsadresse zu den eigentlichen Daten speichert, spricht man auch von einem Zeiger oder eine Referenz auf die eigentlichen Daten. Daher stammt der Name Referenztyp. Eine Referenztyp-Variable speichert also lediglich einen Zugang zu den eigentlichen Daten.

Merke: Referenztyp-Variablen speichern lediglich eine Referenz auf die eigentlichen Daten

Das ist z.B. immer dann sinnvoll, wenn man ein bestimmtes Objekt oder bestimmte Daten mit verschiedenen Namen ansprechen möchte.

Die besondere Speicherung bei den Referenztypen hat einige Vorteile, aber auch einige Nachteile. Wir möchten uns nun ansehen was passiert, wenn man einem Datenfeld ein anderes Datenfeld zuweist.

```
string[] a = {"Original"}; // hinterlegt den Text "Original"
string[] b = a;           // weist dem Datenfeld b das Datenfeld a zu
                           // in Wirklichkeit haben wir nur die ADRESSE
                           // der Variablen kopiert, so dass beide nun
                           // auf den Speicherbereich zeigen, in dem der
                           // Text gespeichert ist
a[0] = "Fälschung";     // nun ändern wir den Text im Datenspeicher
                           // auf den die Variable mit der Adresse zeigt
                           // daher geben nun sowohl a[0] als auch b[0]
                           // den Text "Fälschung" aus
```

## Mehrdimensionale Datenfelder

Die Datenfelder, die wir bisher kennengelernt haben, waren sogenannte eindimensionale Datenfelder.

Zur Darstellung komplexer Datenstrukturen sind eindimensionale Datenfelder aber nicht immer ausreichend. Stellen Sie sich nur einmal vor, Sie wollten beispielsweise die Daten einer Tabelle in einem Datenfeld ablegen. Tabellen speichern ihre Daten in Zeilen und Spalten, also in zwei Dimensionen.

20	40	60
30	50	70

Für die Speicherung dieser Daten bräuchten Sie ein zweidimensionales Datenfeld. In der Praxis werden sehr oft zwei- oder mehrdimensionale Datenfelder benötigt. Daher die Bezeichnung mehrdimensionales Datenfeld.

Blieben wir der Einfachheit halber aber beim zweidimensionalen Datenfeld. Ein zweidimensionales Datenfeld kann man sich als Matrix vorstellen. Eine Dimension beschreibt die Zeile, die andere Dimension die Spalte.

Die Tabelle in unserem Beispiel hatte zwei Zeilen und drei Spalten, also

```
int[,] Tabellendaten = new int[2,3];
```

Wie Sie sehen, werden bei der Initialisierung bereits die Dimensionsgrenzen festgelegt.

### Literale Initialisierung

Auch die literale Initialisierung muss bei der Verwendung mehrdimensionaler Datenfelder angepasst werden. Rufen wir uns noch einmal kurz in Erinnerung, wie die literale Initialisierung bei eindimensionalen Datenfeldern funktionierte.

```
int[] Datenfeld = {23,9,7,4,17};
```

Wie muss nun die literalie Initialisierung für mehrdimensionale Datenfelder aussehen? Sehen Sie sich bitte an, wie wir die Tabelle aus unserem Beispiel abbilden:

```
int[,] Datenfeld = {{20,40,60},{30,50,70}};
```

Jede Dimensionsebene wird durch ein Paar geschweifeter Klammern dargestellt. Da ein zweidimensionales Datenfeld als ein Feld zu verstehen ist, bei dem jedes Element selbst wieder ein eigenes Feld definiert, finden wir innerhalb der umschließenden geschweiften Klammer genau zwei Paare von Daten, die wiederum in eigene geschweifte Klammern gefasst sind. Diese Systematik setzt sich mit jeder weiteren Dimension fort.

## Werte auslesen

Beim Zugriff auf die Daten eines mehrdimensionalen Datenfeldes muss man jede Dimension des entsprechenden Elements einzeln angeben.

Wie greift man also in unserer Beispieltabelle ...

20	40	60
30	50	70

... auf die dritte Spalte der zweiten Zeile (70) zu? Sehen wir es uns an:

```
Console.WriteLine(Tabelle[1,2]);
```

Vergessen Sie dabei nicht, dass die dritte Spalte bei einem 0-basierten Datenfeld durch den Index 2 dargestellt wird (Spaltennummerierung: 0,1,2) und die zweite Zeile entsprechend durch den Index 1 (Zeilennummerierung: 0,1) !

## Initialisierung zur Laufzeit

In der Praxis kommt es oft vor, dass Sie zu einem bestimmten Zeitpunkt während der Laufzeit Daten zwischenspeichern müssen, bei denen Sie erst zur Laufzeit die Anzahl der Einträge ermitteln können. Bisher hatten wir gelernt, dass man mit einem Datenfeld erst arbeiten kann, wenn das Datenfeld initialisiert ist und damit dessen Größe angegeben wurde.

```
Datenfeld = new int[5];
```

Glücklicherweise lässt sich die Größe eines Datenfeldes aber auch zur Laufzeit festlegen. Da zur Entwurfszeit noch nicht bekannt ist, welche Anzahl an Einträgen verwendet werden soll (wie hier in unserem Beispiel 5 Einträge), verwendet man anstelle dessen eine Variable.

Vielleicht möchte man den Benutzer zur Laufzeit fragen, wie viele Preise er aus einer Staffelpreisliste angezeigt haben möchte, oder man fragt nach, wie viele Mitarbeiter der Nachtschicht zugeteilt werden sollen.

In diesen Fällen könnte der Code so ähnlich aussehen, wie hier:

```
string[] Mitarbeiter;  
  
// Anzahl der Mitarbeiter erfragen  
Console.WriteLine("Geben Sie die Anzahl der Mitarbeiter ein: ");  
  
int number = Convert.ToInt32(Console.ReadLine());  
Mitarbeiter = new string[number];  
  
// Alle Elemente in einer Schleife durchlaufen  
for(int i=0; i<number; i++) {  
    Mitarbeiter[i] = "Mitarbeiter " + (i+1);  
    Console.WriteLine("Mitarbeiter[{0}] = {1}", i, Mitarbeiter[i]);  
}
```

In diesem Beispiel kommt es uns vor allem darauf an, dass das Datenfeld zunächst nur deklariert, aber nicht initialisiert wird.

```
string[] Mitarbeiter;
```

Nachdem wir vom Anwender erfragt haben, wie viele Mitarbeiter er einteilen möchte, legen wir die Größe des eindimensionalen Datenfeldes zur Laufzeit anhand der Variablen NUMBER fest.

```
Mitarbeiter = new string[number];
```

Bitte achten Sie aber darauf, dass der Index des Datenfeldes immer vom Typ Integer ist:

```
int number = Convert.ToInt32(Console.ReadLine());
```

## Größe der Dimensionen ermitteln

Hin und wieder werden Sie in der Praxis wissen wollen, wie viele Einträge Ihr Datenfeld eigentlich fassen kann.

Im einleitenden Teil zum Thema Datenfelder hatten wir bereits gelernt, dass es sich bei den Datenfeldern um Objekte handelt. Das ist uns nun zum Vorteil, denn Objekte verfügen bekanntlich über Eigenschaften (Einstellungswerte) und Methoden (Befehle). Dazu gehört auch die Methode GETLENGTH, die uns für die Dimension eines Datenfeldes die Anzahl der jeweiligen Elemente zurückgibt.

Merke: Mit GETLENGTH lässt sich die Anzahl der Elemente in einer Dimension eines Datenfeldes ermitteln

Die Methode liefert eine Ganzzahl (int) zurück.

Damit GETLENGTH aber auch weiß, aus welcher Dimension des Datenfeldes es die Anzahl der Elemente ermitteln soll, müssen wir ihr in der Klammer mitteilen, um welche Dimension es sich handelt. Auch bei diesem Übergabewert handelt es sich um eine Ganzzahl vom Typ INT.

```
int[,] Tabelle = new int[20,45];
```

Um herauszufinden, über wie viele Elemente die zweite Dimension verfügt, rufen wir die Methode `GetLength` wie folgt auf:

```
Tabelle.GetLength(1);
```

Beispielsweise so:

```
Console.WriteLine(Tabelle.GetLength(1));
```

Das Ergebnis ist die Anzahl der Elemente in der zweiten Dimension des Datenfeldes `Tabelle`, also 45.

## Gesamtanzahl der Elemente ermitteln

Bei der Arbeit mit einem mehrdimensionalen Datenfeld ist der Umgang mit `GETLENGTH` schon etwas schwieriger. Stellen wir uns nur vor, wir wollten diesmal nicht die Anzahl der Elemente einer einzelnen Dimension, sondern die Gesamtanzahl der Elemente im entsprechenden Datenfeld ermitteln. Wie ginge man dann vor?

Normalerweise würden wir die Methode `GETLENGTH` auf jeder Dimension aufrufen und anschließend die jeweiligen Rückgabewerte multiplizieren. Das ist relativ aufwändig, je nach dem, wie viele Dimensionen das Datenfeld hat.

Glücklicherweise geht es auch anders. Die Klasse `Array`, die jedem Datenfeld zugrunde liegt, bietet mit der Eigenschaft `LENGTH` die Möglichkeit, die Gesamtanzahl der Elemente in allen Dimensionen des Datenfeldes zu ermitteln.

Merke: Mit `Length` ermittelt man die Gesamtanzahl aller Elemente eines Datenfeldes

Stellen wir uns wieder unser Datenfeld `Tabelle` vor:

```
int[,] Tabelle = new int[20,45];
```

Die Gesamtanzahl der Elemente würden wir nun wie folgt ermitteln:

```
Tabelle.Length;
```

... bzw. als Ausgabe auf der Konsole:

```
Console.WriteLine(Tabelle.Length);
```

Die Ausgabe dieses Codefragments ist 900, denn das Datenfeld enthält insgesamt  $20 * 45$  Elemente.

## Letzten Indexwert eines Datenfeldes ermitteln

Können wir uns noch an den Anfang unserer Ausführungen über Datenfelder erinnern?

Wir hatten gesagt: Wenn die Nummerierung der Elemente mit 0 beginnt, ergibt sich logischerweise folgende Formel zur Ermittlung des letzten Elements im Datenfeld:

Merke: Letzter Index im Datenfeld = Anzahl der Elemente - 1

Jetzt sind wir endlich in der Lage die Anzahl der Elemente zu ermitteln:

```
LetzterIndex = Datenfeld.Length - 1;
```

## Verzweigte Datenfelder

Ein verzweigtes Datenfeld ist ein Datenfeld, dessen Elemente wiederum Datenfelder sind.

Hört sich kompliziert an? Ist es auch! Nun ja, von der technischen Realisierung eigentlich nicht. Nur wir Menschen können manchmal komplexe Konstruktionen nicht mehr nachvollziehen und bilden dann Knoten im Kopf aus. Da sind uns die Maschinen dann doch wieder ein Stück voraus.

**Merke:** Ein verzweigtes Datenfeld ist ein Datenfeld, dessen Elemente wiederum Datenfelder sind.

Sehen wir uns doch einfach mal an, wie man ein Datenfeld anlegt, dass aus vier untergeordneten Datenfeldern besteht:

```
int[][] Komplex = new int[4][];
```

Das hier gezeigte Datenfeld KOMPLEX enthält insgesamt vier Elemente. Jedes dieser vier Elemente ist in der Lage wiederum ein Datenfeld zu speichern. Statt eines einfachen Wertes halten die vier Hauptelemente also wiederum Datenfelder. Das macht die Sache so komplex.

Kennzeichnend für verzweigte Datenfelder ist die doppelte Angabe der rechteckigen Klammern sowohl links als auch rechts vom Gleichheitszeichen. Im ersten Moment ist das verwirrend. Aber vergleichen wir doch einmal: Würden wir ein eindimensionales Datenfeld anlegen, würde die Anweisung wie folgt lauten:

```
int[] Komplex = new int[4];
```

Das ist einfach. Was wir nun tun ist nichts anderes, als dem Computer klar zu machen, dass wir in diesen vier Elementen nun gerne jeweils ein Datenfeld speichern möchten. Deshalb fügen wir einfach noch eine zweite Klammer hinzu, und zwar sowohl im deklarierenden als auch im initialisierenden Teil. Dadurch wird deutlich, dass jedes Element seinerseits ein eigenes Datenfeld repräsentiert.

Jetzt haben wir aber noch ein anderes Problem. In unserem Beispiel haben wir lediglich das eigentliche Datenfeld, bestehend aus den vier Haupteinträgen, initialisiert. Doch wie steht es mit den vier anderen Datenfeldern?

Um das festzulegen, müssen die Einträge einzeln bekannt gemacht werden.

```
Komplex[0] = new int[3];  
Komplex [1] = new int[4];  
Komplex [2] = new int[2];  
Komplex [3] = new int[5];
```

Sind die einzelnen Elemente aller Datenfelder bekannt, kann alternativ auch literal mit

```
Komplex[0] = new int[3] { 1, 2, 3 };
Komplex[1] = new int[4] { 1, 2, 3, 4 };
Komplex[2] = new int[2] { 1, 2 };
Komplex[3] = new int[5] { 1, 2, 3, 4, 5 };
```

oder mit

```
int[][] Komplex = { new int[] { 1, 2, 3 },
                   new int[] { 1, 2, 3, 4 },
                   new int[] { 1, 2 },
                   new int[] { 1, 2, 3, 4, 5 },
```

initialisiert werden.

Beim Zugriff auf das Element eines verzweigten Datenfeldes muss zuerst berücksichtigt werden, in welchem Unter-Datenfeld sich das entsprechende Element befindet.

Angenommen, Sie möchten den Wert 5 aus dem vierten Datenfeld auslesen. Dazu müsste man sich auf das fünfte Element (beim nullbasierten Index ist das die 4) im vierten Datenfeld (beim nullbasierten Index ist das die 3) zugreifen.

```
Console.WriteLine(Komplex[3][4]);
```

Verzweigte Datenfelder können nicht nur mit eindimensionalen Datenfeldern umgehen. Sie können in ein Datenfeld auch mehrdimensionale Datenfelder verschachteln.

Da Sie nun mit allen grundlegenden Techniken dazu vertraut sind möchten wir unsere Darlegungen aber an dieser Stelle beenden und dürfen Ihnen noch viel Freude beim Experimentieren mit Datenfeldern in C# wünschen!

## Zusammenfassung

Begriff	Erklärung
Datenfelder	Datenfelder sind Container für Variablen des gleichen Datentyps
Index	Auf die Daten in einem Datenfeld wird schreibend und lesend über den Index zugegriffen, der immer vom Typ INTEGER ist
Literale Initialisierung	Datenfelder können sofort bei der Initialisierung mit Daten gefüllt werden
Wertetyp	Eine Variable, die die Daten in seinem eigenen Speicherplatz speichert
Referenztyp	Eine Variable, die einen Zeiger auf die eigentlichen Daten speichert
Mehrdimensionale Datenfelder	Datenfelder mit mehreren Dimensionen, z.B. in Form einer Matrix
GetLength	Methode, die die Anzahl der Elemente einer Dimension ermittelt
Length	Methode, die die Anzahl aller Elemente eines Datenfeldes ermittelt
Verzweigte Datenfelder	Datenfelder, die als Elemente wiederum Datenfelder speichern