4 Datentypen

In diesem Kapitel stellen wir Ihnen die grundlegenden Datentypen des .NET Frameworks vor. Außerdem werden Umgang und Deklaration von Variablen und Konstanten behandelt und es wird gezeigt, wie Werte konvertiert werden können.

4.1 Werte- und Referenztypen

4.1.1 Wertetypen

Die meisten Programmiersprachen bieten eine Reihe so genannter *primitiver Datentypen*. Dazu gehören beispielsweise ganzzahlige- oder Gleitkommatypen. Diese Datentypen speichern Werte, im Gegensatz zu Verweistypen, die lediglich eine Referenz speichern.

Im .NET Framework werden diese Datentypen *Wertetypen* genannt. Es gibt zwei Arten von Wertetypen, einmal die im .NET Framework integrierten Datentypen und die selbst definierten. Die Unterscheidung, ob ein Datentyp ein Werte- oder Referenztyp ist, erfolgt durch seinen Ursprung. Im .NET Framework ist eine Klasse namens `ValueType` deklariert, von der alle Wertetypen abstammen.

Da des Öfteren behauptet wird, C# sei vollständig objektorientiert, alles sei ein Objekt usw. werden Sie sich sicherlich fragen, warum denn nicht alles auch ein Referenztyp ist. Ein Objekt ist ein Referenztyp, und wenn alles ein Objekt ist, müsste ja auch jeder Datentyp ein Referenztyp sein.

Die Unterscheidung erfolgt intern, durch die Verhaltensweise. Die Klasse `ValueType` ist direkt von `Object`, der Basisklasse aller Datentypen und Klassen in .NET, abgeleitet. Sie enthält jedoch andere Definitionen für die Verhaltensweise nach außen, also zum Programmierer hin.

Bei einem Vergleich zweier Wertetypen beispielsweise wird wirklich verglichen, ob beide Variablen den gleichen Wert beinhalten. Zu diesem Zweck wurde die Methode `Equals()`, die `ValueType` von `Object` erbt, überschrieben. Geändert wurde auch das Verhalten des Datentyps, was die interne Behandlung von Daten angeht. Wertetypen speichern ihre Daten direkt auf dem so genannten *Stack*, einem Speicher, der für jedes Programm existiert und auf dem Daten abgelegt werden können. Dadurch ist der Zugriff auf die Werte schnell und effizient.

Dass sich die Wertetypen von den primitiven Datentypen anderer Programmiersprachen unterscheiden, wird dadurch deutlich, dass Wertetypen unter .NET durchaus Methoden beinhalten können. Bekanntester Vertreter ist die Methode `ToString()`, die jeder Wertetyp implementiert und die den enthaltenen Wert als Zeichenkette zurückliefert.

Die Unterscheidung zu Referenztypen wird vor allem durch drei Eigenschaften von Wertetypen deutlich:

- Wertetypen sind versiegelt, d.h. von ihnen kann nicht abgeleitet werden. Neue Wertetypen müssen von `ValueType` abgeleitet bzw. als `struct` implementiert werden. Von Referenztypen kann, wenn nicht explizit anders angegeben, immer abgeleitet werden.
- Bei der Verwendung von Wertetypen ist es nicht nötig, einen Konstruktor aufzurufen. Bei Verweistypen ist das Pflicht.
- Wertetypen muss vor ihrer erstmaligen Verwendung ein Wert zugewiesen werden.

4.1.2 Referenztypen

Referenz- oder Verweistypen sind die Datentypen, die bei der Programmierung mit .NET am häufigsten verwendet werden. Da .NET keine Zeiger kennt (bzw. die gesamte Speicherverwaltung intern erfolgt), können solche Datentypen mit einigen Einschränkungen genauso verwendet werden wie Wertetypen.

Einen gravierenden Unterschied (der später noch an Bedeutung gewinnen wird) gibt es dennoch. Variablen, deren Datentyp ein Referenztyp ist, werden auch als *Objektvariablen* bezeichnet. Die Daten einer Objektvariable werden nicht auf dem Stack abgelegt, sondern auf dem so genannten *verwalteten Heap*. Das ist ein Speicherbereich, der vollständig von der Garbage-Collection kontrolliert wird. Auf dem Stack befindet sich lediglich der Verweis auf das eigentliche Objekt.

Das Verständnis des Unterschieds zwischen Referenz- und Wertetypen ist elementar für die Programmierung mit .NET. Ein Unterschied besteht in der Initialisierung eines Referenztyps. Während es bei Wertetypen einfach so ist, dass der entsprechenden Variablen ein Wert zugewiesen wird, muss bei Referenztypen explizit ein Objekt erzeugt oder ein bestehendes Objekt zugewiesen werden. Das geschieht über den Operator `new`. Intern wird durch die Verwendung von `new` der Konstruktor der zugrunde liegenden Klasse aufgerufen.

Ein weiterer Unterschied besteht in der Art und Weise, wie bei einer Zuweisung verfahren wird. Während bei Wertetypen der wirkliche Inhalt einer Variablen kopiert wird, wird bei Referenztypen nur der Verweis kopiert. Nach einer Zuweisung verweisen also beide Variablen auf das gleiche Objekt – eine Änderung des Werts über die erste Variable würde demnach ergeben, dass auch der Wert der zweiten Variable geändert wäre.

Das hat natürlich Auswirkungen. Da zwei Variablen nach einer Zuweisung auf das gleiche Objekt verweisen, bewirkt eine Änderung beim ersten Objekt, dass auch das zweite Objekt geändert wird. Bei Wertetypen ist das nicht der Fall.

HINWEIS

Dieses Verhalten ist enorm wichtig, wenn es um die Übergabe von Parametern an Methoden geht. Standardmäßig werden solche Parameter in C# »by value« übergeben, also als Wert. Bei Objektvariablen ist dieser Wert aber die Referenz auf das Objekt, d.h. eine Änderung am Objekt in der aufgerufenen Methode bewirkt immer eine Änderung des ursprünglichen Objekts.

Analog dazu wird bei einem Vergleich zweier Objektvariablen nicht der Inhalt des Objekts verglichen, sondern nur der Verweis. Zwei Objektvariablen sind demnach gleich, wenn sie auf das gleiche Objekt verweisen (und nicht etwa, wenn beide verglichenen Objekte die gleichen Werte enthalten).

Tipp

Falls Sie sicher sein wollen, ob es sich bei einer Variablen um einen Wertetyp oder einen Referenztyp handelt, können Sie die Eigenschaft `IsValueType` der Klasse `Type` auswerten. Den Datentyp einer Variablen können Sie über die Methode `GetType()` ermitteln:

```
int i = 5;
bool valType = i.GetType().IsValueType
```

4.2 Integrierte Datentypen

Die integrierten Datentypen werden vom .NET Framework direkt bereitgestellt. Es handelt sich dabei ausnahmslos um Wertetypen. Daneben gibt es noch selbst definierte Typen, wie `structs` und `enums`, die ebenfalls Wertetypen darstellen.

C# kennt 13 integrierte Datentypen, die in der folgenden Tabelle aufgelistet sind.

Datentyp	Größe	Wertebereich	Alias
`bool`	1 Bit	`true`, `false`	`System.Boolean`
`byte`	8 Bit	-128 bis 127	`System.Byte`
`sbyte`	8 Bit	0 bis 255	`System.SByte`
`char`	16 Bit	ein Unicode-Zeichen	`System.Char`
`decimal`	128 Bit	$\pm 1.0 \times 10^{28}$ to $\pm 7.9 \times 10^{28}$	`System.Decimal`
`double`	8 Bit	$\pm 5.0 \times 10^{324}$ to $\pm 1.7 \times 10^{308}$	`System.Double`
`float`	4 Bit	$\pm 1.5 \times 10^{45}$ to $\pm 3.4 \times 10^{38}$	`System.Single`
`int`	32 Bit	-2,147,483,648 bis 2,147,483,647	`System.Int32`
`uint`	32 Bit	0 bis 4,294,967,295	`System.UInt32`
`long`	64 Bit	-9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807	`System.Int64`
`ulong`	64 Bit	0 bis 18,446,744,073,709,551,615	`System.UInt64`
`short`	16 Bit	-32768 bis 32767	`System.Int16`
`ushort`	16 Bit	0 bis 65535	`System.UInt16`

Auffallend an obiger Tabelle ist, dass der Datentyp `decimal` offensichtlich einen kleineren Wertebereich besitzt als beispielsweise der Datentyp `double`, obwohl es sich um einen 128 Bit breiten Datentyp handelt. Die größere Anzahl verfügbarer Bits wurde bei `decimal` allerdings nicht in den Wertebereich umgesetzt, sondern in die Genauigkeit.

Während der Datentyp `double` mit einer Genauigkeit von 15-16 Stellen nach dem Komma arbeitet (`float` arbeitet mit 7 Stellen hinter dem Komma), liefert `decimal` eine Genauigkeit von 28 bzw. 29 Stellen hinter dem Komma. Daher ist dieser Datentyp sehr gut für Finanzkalkulationen geeignet.

Ebenso mag dem einen oder anderen geschätzten Leser auffallen, dass der Datentyp `string` fehlt, der eine Zeichenkette repräsentiert und vermutlich am häufigsten verwendet wird. Das hat natürlich einen Grund. `string` ist kein Wertetyp, auch wenn er sich so verhält. Es handelt sich (eigentlich zwangsläufig) um einen Referenztyp, da die Größe der Zeichenkette erst zur Laufzeit festgelegt wird und ein String somit keine feste Größe haben kann. In Abschnitt 4.2.4 ab Seite 98 werden Strings angesprochen, Abschnitt 8.2 ab Seite 227 liefert weitere Informationen zum Arbeiten mit Strings.

4.2.1 Der Datentyp bool

Da es sich bei C# um eine typsichere Sprache handelt, muss dem Datentyp `bool` ein wenig mehr Aufmerksamkeit geschenkt werden. Es handelt sich dabei nämlich wirklich um einen eigenständigen Datentyp, der *nicht* durch einen Zahlenwert interpretiert werden kann.

Vor allem C++-Programmierer haben bei solchen Werten gerne so gearbeitet, dass sie Verzweigungen anhand des Zahlenwerts einer Variable durchgeführt haben. Ein Zahlenwert von 0 entspricht in C++ dem booleschen Wert `false`. Es kam daher häufig zu dem Fehler, dass statt eines Vergleichs, der mit doppeltem Gleichheitszeichen durchgeführt wird, eine Zuweisung stattfand. Das Programm erkannte diesen Fehler nicht, denn mit der Zuweisung hatte die Variable einen Wert, der wiederum als boolescher Wert ausgewertet wurde.

Sollte beispielsweise der Wert einer Variablen daraufhin überprüft werden, ob er 0 ist, würde der Vergleich folgendermaßen aussehen:

```
// C++ - Code
if ( a == 0 ) {
  // Anweisungen ...
}
```

Der Fehler, der oft passierte, war, dass eines der Gleichheitszeichen vergessen wurde:

```
// C++ - Code
if ( a = 0 ) {
  // Anweisungen
}
```

In diesem Fall wird der Variablen `a` im Kopf des Vergleichs der Wert 0 zugewiesen. Das entspricht aber (unter C++) dem Wert `false`, wodurch sich genau das Gegenteil des gewünschten Verhaltens einstellt – der Wert ist 0, aber die Anweisungen werden nie ausgeführt, obwohl die Bedingung (eigentlich) erfüllt ist.

In C# kann dies nicht passieren. Der Compiler mahnt die Zuweisung an und beschwert sich, dass ein Vergleich stattfinden muss. Ein Zahlenwert kann nicht als boolescher Wert interpretiert werden.

4.2.2 Der Datentyp char

Der Datentyp `char` hat in C# (bzw. im .NET Framework) eine Größe von 16 Bit oder 2 Byte. Das liegt daran, dass .NET vollständig auf Unicode basiert. Alle Zeichen (auch in der Entwicklungsumgebung selbst) werden mit 2 Bytes pro Zeichen dargestellt.

HINWEIS

Die Verwendung von Unicode ist durchaus logisch, denn mit ASCII oder ANSI sind sprachenübergreifende Anwendungen aufgrund der unterschiedlichen Sonderzeichen nur sehr schwer zu realisieren. Mithilfe von Unicode können nun alle Sonderzeichen aller Weltsprachen in einem Zeichensatz untergebracht werden – und es ist sogar noch Platz für weitere Sprachen.

Zuweisungen an eine Variable vom Typ `char` geschehen in einfachen Anführungszeichen. Geschieht eine Zuweisung mithilfe von doppelten Anführungszeichen, so betrachtet der Compiler das zugewiesene Zeichen als `string` und meldet einen Fehler.

```
char c = 'a';  // Zeichen a wird zugewiesen - korrekt
char c = "a";  // "a" wird als string repräsentiert - Fehler
```

Escape-Sequenzen

Mithilfe spezieller Literale, die auch als *Escape-Sequenzen* bezeichnet werden, können Sonderzeichen als `char` dargestellt werden. Ihre Verwendung ist sowohl als einzelnes Zeichen als auch innerhalb von Strings möglich.

Sequenz	Bedeutung
`\a`	Alarm – Ein Signalton wird ausgegeben
`\b`	Backspace
`\c`	Entspricht einem Zeichen zusammen mit [STRG].
`\f`	Seitenumbruch
`\r`	Carriage Return (Wagenrücklauf)
`\n`	Zeilenumbruch (NewLine)
`\t`	Horizontaler Tabulator
`\"`	Anführungszeichen innerhalb eines Strings
`\'`	Einfaches Anführungszeichen innerhalb eines Strings
`\\`	Backslash
`\v`	Vertikaler Tabulator
`\e`	Die Taste [ESC]
`\uXXXX`	Entspricht einem Unicode-Zeichen. XXXX entspricht dem Hex-Wert des Zeichens.

Die Möglichkeit, beliebige Unicode-Sequenzen anzugeben, ist nicht auf die Datentypen `char` und `string` beschränkt. Da auch die Entwicklungsumgebung mit Unicode arbeitet, werden solche Sequenzen auch dort ausgewertet, beispielsweise als Bestandteil eines Variablenbezeichners. Als Beispiel:

```
int \u0041\u0042\u0043 = 10;
Console.WriteLine("Wert von ABC: {0}", ABC);
```

liefert als Ausgabe:

```
Wert von ABC: 10
```

Eine solche Vorgehensweise empfiehlt sich allerdings nicht, da es kaum eine Möglichkeit gibt, Code noch schlechter lesbar zu machen als durch die Verwendung von Unicode-Sequenzen in Variablenbezeichnern.

4.2.3 Numerische Datentypen

Bei den numerischen Datentypen wird zwischen *integralen* und *Gleitkommatypen* unterschieden. Integrale Typen sind alle ganzzahligen Typen wie z.B. `int` oder `long`, zu den Gleitkommatypen gehören `float` und `double`. Der Datentyp `decimal` nimmt eine Sonderstellung ein, da er speziell für finanzmathematische Funktionen vorgesehen ist. Es handelt sich jedoch ebenfalls um einen Gleitkommatyp.

Suffixe

Suffixe dienen der genauen Festlegung des Datentyps bei numerischen Werten. Beispielsweise ist standardmäßig festgelegt, dass ein Gleitkommawert, so nicht anders angegeben, immer als `double` gehandhabt wird. Bei einem ganzzahligen Wert ist `int` der Standard-Datentyp. Durch Suffixe können Sie dieses Verhalten ändern und den Datentyp des Werts festlegen.

Die Groß-/Kleinschreibung spielt dabei keine Rolle, außer beim Suffix `L`. Hier sollte in jedem Fall der Großbuchstabe verwendet werden, da es ansonsten zu Verwechslungen kommen kann. Das kleine `l` ähnelt zu stark der Ziffer 1. Die folgende Tabelle zeigt die Suffixe und die repräsentierten Datentypen.

Suffix	Datentyp
D, d	Der Wert wird als `double` interpretiert.
F, f	Der Wert wird als `float` interpretiert.
L,(l)	Der Wert wird als `long` interpretiert.
UL, ul	Der Wert wird als `ulong` interpretiert.
M,m	Der Wert wird als `decimal` interpretiert.

ACHTUNG

Vor allem bei der Zuweisung von `float`-Werten sind diese Suffixe wichtig. Da Werte mit Kommastelle vom Compiler als `double` angesehen werden, `float` aber einen kleineren Wertebereich als `double` besitzt, ergibt sich bei folgender Zuweisung ein Fehler:

```
float f = 2.5;
```

Daher sollten Sie sich die Verwendung der Suffixe fast schon grundsätzlich angewöhnen. Vor allem im Grafik-Kapitel werden wir häufiger Gebrauch von dieser Möglichkeit machen.

4.2.4 Der Datentyp string

Zeichenketten, üblicherweise im Fachjargon auch *Strings* genannt, sind der wohl am häufigsten verwendete Datentyp in nahezu jeder Programmiersprache. Im .NET Framework hat der Datentyp `string` eine ganz besondere Bedeutung, weil es sich bei ihm um einen Zwitter handelt. Eigentlich ist `string` ein Referenztyp, nach außen hin verhält er sich allerdings wie ein Wertetyp. Die Basisklasse des .NET Frameworks befindet sich im Namespace `System` und heißt `String`.

Das Verhalten eines String nach außen (also zum Programmierer) hin ist das Gleiche wie bei einem Wertetyp. Dennoch handelt es sich intern um einen Referenztyp. Wenn einer Variablen vom Typ `string` ein Wert zugewiesen wird, wird intern auf dem Heap entsprechend Speicherplatz bereitgestellt – der Wert wird nicht auf dem Stack gespeichert, wie bei Wertetypen üblich.

Weiterhin ist string *immutable*, d.h. unveränderlich. Einmal zugewiesen, kann der Wert eines Strings nicht mehr verändert werden. Vermutlich haben Sie es aber schon einmal versucht oder in einem Beispiel gesehen, und wissen, dass ein String sehr wohl erweitert werden kann. Nach außen hin ja, intern geschieht etwas völlig anderes. Bei der Zuweisung eines Wertes an eine String-Variable wird der String initialisiert. Soll der Inhalt nun verändert werden, z.B. indem ein weiterer String angehängt wird, geschieht Folgendes:

- Die Gesamtlänge des neuen Strings wird ermittelt.
- Auf dem Heap wird entsprechend Speicher reserviert.
- Die Daten werden an diesen neuen Speicherplatz kopiert.
- Die Referenz wird auf den neuen Platz auf dem Heap umgelegt.

Bei jeder Änderung des Inhalts einer String-Variablen wird also intern eine Kopie der Zeichenkette erzeugt. Das beeinflusst die Performance, obwohl es im Falle weniger Zuweisungen nicht auffällt. Wir werden aber später ein Beispiel sehen, bei dem recht schnell deutlich wird, wie groß dieser Performanceverlust wirklich ist.

HINWEIS

Mehr über Strings und ihre Verwendung finden Sie in Abschnitt 8.2 ab Seite 227. Weil `string` ein häufig verwendeter Datentyp ist und weil dieser Datentyp umfangreiche Methoden zur Verfügung stellt, erhält die Behandlung von Strings einen eigenen Bereich in diesem Buch.

4.3 Variablen

Zum Speichern und Verarbeiten der Daten innerhalb eines Programms werden Variablen verwendet. In C# besitzt jede Variable einen expliziten Datentyp, der ihr bei der Deklaration zugewiesen wird.

Es gibt im Großen und Ganzen drei Arten von Variablen:

- *Lokale Variablen* werden innerhalb von Methoden verwendet. Sie sind so lange gültig, wie der Block, in dem sie deklariert wurden, abgearbeitet wird, danach werden sie aus dem Speicher entfernt. Auch Parameter von Methoden werden als lokale Variablen angesehen.
- *Instanzvariablen* sind Bestandteil einer Klassendeklaration (Klassen werden im Detail in Abschnitt 6.3 ab Seite 145 beschrieben). Ihre Lebensdauer entspricht der des Objekts, also der Instanz der jeweiligen Klasse. Instanzvariablen werden häufig auch als Felder einer Klasse bzw. eines Objekts bezeichnet. Auch in diesem Buch werden wir diese Bezeichnung verwenden.
- *Klassenvariablen* sind ebenfalls Bestandteil der Klassendefinition, allerdings nicht auf Instanzebene, sondern auf Klassenebene. Ihre Lebensdauer entspricht der des Programms, in der die Klasse deklariert wurde. Klassenvariablen existieren jeweils nur einmal (nämlich im Bezug auf die Klasse), nicht für jedes Objekt. Man bezeichnet sie häufig auch als statische Variablen oder statische Felder.

Auf Klassen- bzw. Instanzvariablen wird im Bezug auf Klassen noch genauer eingegangen. Dieser Abschnitt widmet sich in der Hauptsache den lokalen Variablen.

HINWEIS

Die Bezeichnung *Objektvariable* bedeutet nicht das Gleiche wie *Klassenvariable*. Als Objektvariable werden die Variablen bezeichnet, deren zugewiesener Typ ein Referenztyp ist (und die somit ein Objekt referenzieren). Klassenvariablen hingegen sind auf Klassenebene deklarierte Felder, die unabhängig von einer Instanz verwendet werden können.

4.3.1 Deklaration und Initialisierung

Die Deklaration einer Variablen erfolgt durch die Angabe des Datentyps, gefolgt vom Bezeichner der Variablen. Als Datentyp kann dabei sowohl der .NET-Datentyp als auch der entsprechende Alias der Sprache C# verwendet werden. Die Deklaration einer 32-Bit-Integer-Variablen kann demnach auf zwei Arten erfolgen:

```
int a;
```

oder

```
System.Int32 a;
```

Beide Male handelt es sich um den gleichen Datentyp.

Es ist auch möglich, mehrere Variablen des gleichen Typs auf einen Schlag zu deklarieren. Dazu werden die Bezeichner durch Komma getrennt:

```
int a, b, c;
```

Wo die Deklaration einer Variablen innerhalb einer Methode erfolgt, ist irrelevant. Der Compiler sieht eine Deklaration als eine Anweisung an und führt sie aus, sobald er darauf stößt. Anders als beispielsweise in Delphi gibt es keine Notwendigkeit, Variablen am Anfang einer Methode bekannt zu machen.

Allerdings müssen Variablen vor der ersten Verwendung sowohl deklariert als auch initialisiert sein. Unter Initialisierung versteht man das erste Zuweisen eines Werts. Die Initialisierung kann auch bereits bei der Deklaration erfolgen, indem einfach die Zuweisung angehängt wird:

```
int a = 10;
```

oder, bei einer mehrfachen Deklaration:

```
int a=10, b=15, c;
```

In diesem Fall wären drei Variablen deklariert, zwei davon wurden auch initialisiert. Alle drei sind vom Typ `int` (bzw. `System.Int32`).

4.3.2 Bezeichner

Variablenbezeichner (und auch Bezeichner von Klassen, Methoden und anderen Bestandteilen eines Programms) unterliegen bestimmten Regeln, was sowohl ihre Verwendung als auch ihren Aufbau angeht. Ein gültiger Bezeichner beginnt entweder mit einem alphanumerischen Zeichen oder einem Unterstrich. Innerhalb des Bezeichners dürfen auch Zahlen auftauchen. Ebenso ist es erlaubt, Sonderzeichen der jeweiligen Landessprache zu verwenden (ein Dank an Unicode). Leerzeichen innerhalb eines Bezeichners sind hingegen nicht erlaubt. Die folgende Liste zeigt einige gültige und ungültige Bezeichner:

```
int _myValue;     // korrekt, beginnt mit Unterstrich
double 1Wert;     // Fehler, Bezeichner beginnt mit einer Ziffer
string Währung;   // korrekt, beinhaltet Sonderzeichen
int ein Wert;     // Fehler, Leerzeichen innerhalb des Bezeichners
```

Auch hier unterscheidet C# zwischen Groß- und Kleinschreibung. Deshalb sollten Sie sich gewisse Konventionen für Schreibweisen aneignen, die Sie in Ihren eigenen Programmen verwenden. Die Schreibkonventionen in diesem Buch und auch die verschiedenen verwendeten Schreibweisen finden Sie in Abschnitt 2.1.3 ab Seite 55.

Reservierte Wörter als Bezeichner

C# enthält eine große Anzahl reservierter Wörter, die nicht als Bezeichner verwendet werden dürfen. Trotzdem ist es möglich. Durch Voranstellen des *at-Zeichens* (auch als Klammeraffe bezeichnet, `@`) können Sie festlegen, dass ein Bezeichner »wörtlich« genommen, also nicht vom Compiler interpretiert wird. Die folgende Deklaration wäre durchaus möglich (und würde auch ohne weiteres vom Compiler akzeptiert):

```
string @string = "";
for ( int @int=1; @int<10; @int++ ) {
  @string = @string+@int.ToString();
}
Console.WriteLine(@string);
```

Eine solche Art der Programmierung ist allerdings für jeden Programmierer nahezu undurchschaubar, sogar schon bei einem derart kleinen Beispiel. Aus diesem Grund empfehlen wir Ihnen, diese Möglichkeit gleich wieder zu vergessen und sie auf keinen Fall anzuwenden.

Hinweis

In C# ist die Verwendung des Zeichens `@` zwar möglich, aber in keinem Fall erforderlich. Unter VB.NET, wo der gleiche Effekt dadurch erzielt wird, dass der entsprechende Bezeichner in eckige Klammern gesetzt wird, ist das nicht so. VB.NET kennt beispielsweise das reservierte Wort `Module`. Im Namespace `System.Reflection` existiert allerdings auch eine gleichnamige Klasse. Um diese zu verwenden muss sie daher entweder immer komplett qualifiziert oder ihr Name muss in eckige Klammern gesetzt werden: `[Module]`.

4.3.3 Gültigkeitsbereich

Auf den Gültigkeitsbereich lokaler Variablen soll hier nochmals kurz eingegangen werden, denn hier ergeben sich einige Besonderheiten. Lokale Variablen sind, wie schon weiter oben angemerkt, in dem Block gültig, in dem sie deklariert wurden. Sie sind *nicht* gültig in dem Block, der dem Deklarationsblock übergeordnet ist. Ein kleines Beispiel soll das veranschaulichen:

```
class Class1 {

  static void Main(string[] args) {
    int i = 5;  // i ist deklariert
    for ( int u=0; u<10; u++ ) {
      i = i + u;
      Console.WriteLine( i );
    }
    Console.WriteLine ( u );  //Fehler!!!

    Console.ReadLine();
}
```

Der Compiler meldet hier bereits beim Kompilieren einen Fehler (die entsprechende Zeile ist durch einen Kommentar markiert). Die Variable `u` wurde im Kopf der `for`-Anweisung deklariert, einer Schleifenanweisung. Variablen, die im Kopf einer Schleife (oder einer anderen Anweisung, die einen Block beinhaltet) deklariert wurden, gehören zum Block der Schleife, nicht zum übergeordneten Block der Methode. Die Variable `u` ist dem Compiler daher unbekannt, ihre Existenz endete mit dem Verlassen des Blocks.

Anders verhält es sich mit der Variable `i`, die im Block der Methode, vor der Schleife, deklariert wurde. Diese ist sehr wohl innerhalb des untergeordneten Blocks sichtbar und dürfte dort auch nicht mehr deklariert werden.

Umgekehrt funktioniert es aber auch nicht. Wenn eine Variable in einem Programmblock deklariert ist und dann in einem untergeordneten Block erneut deklariert würde, würde die gleiche Variable des übergeordneten Blocks verdeckt. Das ist nicht erlaubt, daher meldet der Compiler auch hier einen Fehler.

HINWEIS

Instanzvariablen einer Klasse werden im Vergleich zu Methoden auch in einem übergeordneten Block deklariert. Sie können jedoch sehr wohl verdeckt werden, weil man auf Instanzvariablen explizit mithilfe des reservierten Wortes `this` zugreifen kann. Es handelt sich dabei sogar um eine übliche und weit verbreitete Vorgehensweise. Mehr darüber in Abschnitt 6.3 über Klassen ab Seite 145.

4.3.4 Konstanten

Konstanten werden mithilfe des reservierten Wortes `const` deklariert. Es handelt sich dabei um unveränderliche Werte, deren Initialwert bereits bei der Deklaration zugewiesen werden muss.

```
const int myConstant = 10;
```

Konstanten werden fast nicht verwendet. Da es globale Variablen nicht gibt (und somit auch keine globalen Konstanten), müssen sie entweder innerhalb einer Methode verwendet werden (wo ohnehin der Wert selbst verwendet werden könnte) oder als Bestandteil einer Klasse.

VERWEIS

Der bessere Weg, konstante Werte zu verwenden, ist der Weg über einen Aufzählungstyp (`enum`). Mehr über diesen Datentyp, bei dem es sich ebenfalls um einen Wertetyp handelt, erfahren Sie in Abschnitt 4.6 auf Seite 114.

4.4 Konvertierungen und Boxing

4.4.1 Implizite und explizite Konvertierung

Als typsichere Sprache erfordert es C#, dass jede Variable einen bestimmten Datentyp hat, dessen Werte sie aufnehmen kann. Es gibt keine Einstellung `Option Strict` wie z.B. in Visual Basic .NET – C# verhält sich immer entsprechend der Einstellung `Option Strict On`, d.h. die Typen einer Variablen und der ihr zugewiesenen Werte müssen identisch sein.

Da dies nicht immer vorausgesetzt werden kann, gibt es verschiedene Konvertierungsmechanismen, nämlich

- implizite Konvertierung und
- explizite Konvertierung (Casting)

Von der impliziten Konvertierung bekommen Sie normalerweise nichts mit. Bei einer Zuweisung kontrolliert der Compiler, ob der Datentyp einer Variablen den zugewiesenen Wert problemlos und ohne Wert- oder Genauigkeitsverlust aufnehmen kann. Ist das der Fall, wird implizit konvertiert, also ohne dass der Programmierer etwas davon mitbekommt. Damit ist es beispielsweise möglich, einer Variablen vom Datentyp `long` einen Wert vom Datentyp `int` zuzuweisen.

Anders sieht es aus, wenn Verluste auftreten können, entweder was die Genauigkeit (bei Fließkommawerten) oder die Größe der Daten betrifft. In diesem Fall meldet der Compiler einen Fehler. Die Konvertierung kann jedoch nach wie vor durchgeführt werden, nur muss man den Compiler jetzt dazu zwingen. Das geschieht, indem man den zu konvertierenden Datentyp in Klammern vor den Wert schreibt.

```
// Beispiel für explizite Konvertierung
short s = 0;
int   i = 10;  // Der Wertebereich von i ist größer als der von s

s = i;         // Fehler !!
s = (short)i;  // ok
```

HINWEIS

Zahlen gelten in .NET natürlich auch als Datentypen und besitzen auch einen Typ, den der Compiler überprüfen kann. Standardmäßig sind alle ganzen Zahlen vom Datentyp `System.Int32` und alle Fließkommazahlen vom Datentyp `System.Double`. Sie können das leicht testen, mit je einer Zeile Code:

```
Console.WriteLine(10.GetType().ToString());
Console.WriteLine(10.0.GetType().ToString());
```

Zeichen

Casting wird auch angewandt bei der Umwandlung von Zahlenwerten in das entsprechende Zeichen des Alphabets. Das Zeichen `A` beispielsweise wird durch den Wert 65 repräsentiert. Die folgende Zuweisung ist damit korrekt und führt zum gewünschten Ergebnis:

```
char c = (char)65;
```

In die andere Richtung funktioniert es natürlich auch:

```
int i = (int)'A';
```

4.4.2 Boxing und Unboxing

Beim Vorgang des Boxing müssen wir nochmals auf den Unterschied zwischen Werte- und Referenztypen zurückkommen. Bisher wurde lediglich mit Wertetypen gearbeitet, es gab aber im Verlaufe dieses Kapitels auch schon den Fall, dass eine Methode den Datentyp `Object` zurückgeliefert hat, der die Basis aller Datentypen in .NET ist. In diesem Fall musste ebenfalls ein Casting durchgeführt werden, um den korrekten Wert zu erhalten.

Eigentlich müsste man Boxing zusammen mit Klassen und Objekten behandeln, vom Thema her gehört es aber in diesen Abschnitt.

Obwohl Werte- und Referenztypen sich unterschiedlich verhalten, sind sie innerhalb des .NET Frameworks wie eigentlich alles auch als Klassen (bzw. Strukturen) implementiert. Sie stammen von einer Klasse ab (`ValueType`), die selbst wiederum von `Object` abgeleitet ist. Der Datentyp `object`, so viel sollte mittlerweile klar geworden sein, steht für jeden beliebigen anderen Datentyp. Er ist quasi der Ersatz für den Datentyp *Variant* aus VB6, auch wenn es ein wenig anders funktioniert. Es handelt sich allerdings um einen Referenztyp.

Viele Methoden erwarten als Übergabeparameter einen Wert vom Typ `Object`, da es sich um universelle Methoden handelt, die mit jedem Datentyp umgehen können. Wird einer solchen Methode ein Wertetyp übergeben, wird dieser in einem Objekt »verpackt«, d.h. das Objekt dient wirklich als Hülle um einen Datentyp, der eigentlich ein Wertetyp ist. Diese Möglichkeit basiert auf einem der Programmierparadigmen der objektorientierten Programmierung. Diese Art, einen Wertetyp zu verpacken, quasi in eine »Box« zu stecken, bezeichnet man als *Boxing*.

Boxing ist eine Form der impliziten Konvertierung, d.h. der Compiler kümmert sich darum. Anders sieht es aus, wenn der Wert wieder entnommen werden soll. In diesem Fall, dem *Unboxing*, wird explizit konvertiert, d.h. es muss Casting angewendet werden. Das folgende Beispiel zeigt, wie das dann funktioniert.

```
int i = 100;
object o = i;    // Boxing
int u = (int)o;  // Unboxing
```

Das Objekt `o`, ein Referenztyp, beinhaltet also einen Wert vom Typ `int`. Dass dem so ist und dass der Compiler das auch genau weiß, wird deutlich, wenn der Typ von `o` ausgegeben wird:

```
Console.WriteLine(o.GetType());  //liefert als Ausgabe: System.Int32
```

Das Casting zurück muss demnach in den korrekten Datentyp erfolgen, da der Compiler den Datentyp kennt und C# als typsichere Sprache keine Vermischung von Datentypen erlaubt. Soll also der oben angegebene Wert 100 in eine Variable vom Typ `byte` überführt werden, muss doppelt gecastet werden, nämlich einmal wegen des Unboxing und dann zur expliziten Konvertierung:

```
int i = 100;
object o = i;  //Boxing
byte b = (byte)(int)o;  //Doppeltes Casting
```

4.4.3 Konvertierungsmethoden

Die häufigste Konvertierungsform ist vermutlich die Konvertierung einer Zahl in einen String und umgekehrt. Die Konvertierung einer Zahl zu einem String ist dabei nicht weiter schwierig, jeder Datentyp beinhaltet eine Methode `ToString()`, die genau diese Konvertierung durchführt (zumindest bei Wertetypen, bei Objekten ist die Methode überschreibbar und kann damit jedes beliebige Ergebnis liefern). `ToString()` ist sogar eine sehr flexible Methode, da sie auch die Formatierung des zu konvertierenden Werts erlaubt. Mehr Informationen zu diesen Formatierungen erhalten Sie in Abschnitt 8.4 ab Seite 252.

Von einem String zu einem Wertetyp ist das Ganze schon ein wenig schwieriger. In Beta1 des .NET Frameworks hatte die `String`-Klasse noch einige sehr nützliche Methoden, die diese Konvertierung direkt erledigten. Ab Beta 2 waren diese Methoden verschwunden (was damals etwa eine halbe Stunde Suche nach einer Konvertiermöglichkeit mit sich brachte).

Grundsätzlich bestehen zwei Möglichkeiten. Entweder über die Methode `Parse()` des Zieldatentyps oder über eine der Methoden der Klasse `Convert`. Alle Methoden von `Convert` sind statisch, sie werden direkt über den Klassennamen aufgerufen und nicht über eine Instanz. Das gleiche gilt für die Methode `Parse()` eines Datentyps. Die Konvertierung eines `string` zu einem `int` kann demnach auf folgende Art vor sich gehen:

```
int i = System.Int32.Parse(s);
// oder:
int u = Convert.ToInt32(s);
```

Die Klasse `Convert` bietet eine große Anzahl solcher Konvertierungsmethoden. Der Unterschied besteht darin, dass bei der Methode `Parse()` auch noch kulturabhängige Informationen berücksichtigt werden, bei den Methoden von `Convert` ist das nicht der Fall.

Kontrollieren des Werts

Vor der Konvertierung eines Strings in eine Zahl sollte man kontrollieren, ob die `string`-Variable auch wirklich eine Zahl im entsprechenden Format zur Verfügung stellt. Eine entsprechende Methode scheint auf den ersten Blick nicht zu existieren, an ungewöhnlicher Stelle findet man sie aber doch, nämlich in dem Datentyp `Double`. Es handelt sich um die statische Methode `TryParse()`. Sie erwartet als Parameter den String, der konvertiert wer-

den soll, eine Variable des Typs NumberStyles, einen Parameter des Typs IFormatProvider und die Ergebnisvariable als out-Parameter. Der zurückgelieferte Wert ist ein boolescher Wert, wenn dieser true ist, war die Konvertierung erfolgreich.

NumberFormat ist ein Aufzählungstyp, genauer gesagt ein Bitfeld, und im Namespace System.Globalization deklariert. Aufzählungstypen (enum) wurden noch nicht behandelt, die entsprechende Erklärung folgt in Abschnitt 4.6 auf Seite 114 bzw. speziell zu Bitfeldern in Abschnitt 4.6.3 auf Seite 117. An dieser Stelle nur die Information, dass über die Konstanten dieses Aufzählungstyps festgelegt wird, wie die Zahl aussehen muss, damit eine Konvertierung erfolgen darf. In diesem Fall wird bestimmt, ob ein Tausendertrennzeichen vorhanden sein darf, ob Leerzeichen ignoriert werden, ob ein Dezimalpunkt vorhanden sein darf usw.

Im Fall von double-Werten gehen Kontrolle und Konvertierung in einem hin, denn der letzte Parameter (ein out-Parameter, mehr zu diesen Parameterarten in Abschnitt 6.3.3 ab Seite 148) enthält nach dem Aufruf den konvertierten Wert. Falls es sich um einen int-Wert handelt, kann TryParse() natürlich nur die Kontrolle übernehmen. Ein out-Parameter vom Typ double muss aber dennoch übergeben werden. Sie können sich nach der Kontrolle dann entscheiden, ob Sie diesen nach int casten, Int32.Parse() mit dem kontrollierten String aufrufen oder Convert.ToInt32() verwenden.

Der Parameter vom Typ IFormatProvider ist ebenfalls noch nicht bekannt. An dieser Stelle wird hier einfach die Kulturinformation für Deutschland übergeben. IFormatProvider ist ein Interface, das von verschiedenen Klassen (darunter auch die Klasse CultureInfo) implementiert wird. Mehr zu Interfaces erfahren Sie in Kapitel 7 ab Seite 203.

Der Vorteil von TryParse() ist, dass keine Exception bei einer fehlerhaften Konvertierung ausgelöst wird. Es wird lediglich false zurückgeliefert.

Ein Beispiel für die Kontrolle bei int-Werten:

```
double result;
string s = "120";
bool isOk;
CultureInfo ci = new CultureInfo("de-DE");
isOk = double.TryParse(s, NumberStyles.Integer, ci, result);
int res = (int)result;
```

4.5 Arrays

Arrays dienen dazu, mehrere Werte gleichen Datentyps zusammenzufassen. Anders als in vielen Sprachen, bei denen Arrays ein Bestandteil der Sprache selbst sind, handelt es sich in C# dabei um Instanzen der Klasse Array, die im Namespace System deklariert ist. Ein Array ist also ein Objekt, ein Referenztyp. Elemente von Arrays hingegen können sowohl Wertetypen als auch Referenztypen beinhalten. Weil sie ein häufig genutztes Mittel zur Gruppierung von Daten sind, werden sie an dieser Stelle besprochen.

4.5.1 Eindimensionale Arrays

Die Deklaration eines Arrays sieht fast so aus wie die Deklaration einer herkömmlichen Variablen, mit dem Unterschied, dass an den Datentyp selbst eckige Klammern angehängt werden:

```
<Datentyp>[] <Bezeichner>;
```

Allein durch die Deklaration erhält ein Array noch keine Größe. Diese wird bei der Initialisierung des Arrays festgelegt. Wie auch bei den anderen Variablen gilt, dass Deklaration und Initialisierung zusammengefasst werden können. Die folgenden Zeilen deklarieren jeweils ein Array aus `int`-Variablen mit einer Größe von 5 Elementen.

```
int[] arr1;               // Deklaration
arr1 = new int[5];        // Initialisierung
int[] arr2 = new int[5];  // Deklaration und Initialisierung
```

Wenn bereits bei der Deklaration feststeht, welche Werte die Elemente des Arrays haben sollen, können diese sofort zugewiesen werden. In diesem Fall ist es nicht mehr notwendig, die Größe des Arrays anzugeben, da diese durch die Anzahl der übergebenen Elemente bestimmt wird. Der Operator `new` ist weiterhin notwendig, da es sich bei Arrays um Objekte handelt und daher eine Instanz erzeugt werden muss.

Die Werte für die Elemente werden in geschweiften Klammern direkt hinter die Deklaration geschrieben. Eine solche Zuweisung muss zwingend bei der Deklaration erfolgen.

```
int[] arr = new int[] {1, 1, 2, 3, 5, 8};
```

Dieses Beispiel initialisiert ein Array mit sechs Elementen des Typs `int`. Bei dieser Art der Initialisierung kann eine verkürzte Schreibweise angewendet werden, bei der die `new`-Klausel entfällt:

```
int[] arr = {1, 1, 2, 3, 5, 8};
```

Hinweis

Obwohl es sich bei den Elementen der hier als Beispiel verwendeten Arrays um Wertetypen handelt, die ja eigentlich initialisiert werden müssten, ist das bei einer Array-Deklaration nicht notwendig. Jeder Wert des Arrays wird automatisch mit dem Standardwert des jeweiligen Datentyps initialisiert (im Falle des Datentyps `int` ist das der Wert 0). Der Grund hierfür ist, dass es sich bei einem Array um einen Referenztyp handelt – die Initialisierung mit einem Standardwert für die enthaltenen Variablen ist hier der Standard.

Achtung

Anders als in VB.NET ist die Größe eines Arrays in C# final. VB.NET kennt die (sprachspezifischen) Anweisungen `ReDim` bzw. `ReDim Preserve`, mit denen die Größe eines Arrays nachträglich verändert werden kann. Das ist in C# nicht möglich, hier müssen ein neues Array erzeugt und die Elemente kopiert werden.

4.5.2 Mehrdimensionale Arrays

Ein Array muss nicht zwangsläufig nur eine Dimension haben. Es ist beispielsweise auch denkbar, die Werte einer Tabelle in einem Array zu speichern. In diesem Fall werden zwei Dimensionen benötigt, eine für die Spalten und eine für die Zeilen.

Die Deklaration eines mehrdimensionalen Arrays ist sehr ähnlich zur Deklaration eines eindimensionalen Arrays. Dass mehrere Werte angegeben werden, das Array also mehrere Dimensionen hat, wird durch ein Komma signalisiert:

```
int[,] multiArray = new int[5, 7];
```

Diese Programmzeile deklariert ein Array mit 5 Spalten und jeweils 7 Zeilen (oder 5 Zeilen mit jeweils 7 Spalten, ganz wie Sie es sehen wollen). Weitere Dimensionen sind durch eine weitere Verwendung des Kommas möglich:

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

Das obige Array besitzt vier Dimensionen. Die Gesamtanzahl der Werte dieses Arrays ist also 24 (die Anzahl der Elemente der einzelnen Dimensionen wird multipliziert).

Hinweis

Prinzipiell ist es möglich, ein Array mit so vielen Dimensionen wie gewünscht zu deklarieren. In der Regel machen mehr als drei Dimensionen allerdings kaum Sinn. Mit jeder weiteren Dimension wird ein Array schlechter durchschaubar.

Auch für mehrdimensionale Arrays gilt, dass den einzelnen Elementen bereits bei der Deklaration ein Wert zugewiesen werden kann. Auch hier geschieht dies durch Werte in geschweiften Klammern. Dimensionen und Werte werden dabei durch Kommata getrennt. Die Größe des Arrays wird durch die Anzahl der Werte festgelegt.

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

Das deklarierte Array besitzt drei Dimensionen mit je zwei Werten. Die Werte zugeordnet zum jeweiligen Element sind:

```
arr[0,0] : 0
arr[0,1] : 1
arr[1,0] : 2
arr[1,1] : 3
arr[2,0] : 4
arr[2,1] : 5
```

Wieder ist auch hier die verkürzte Schreibweise zulässig. Das gleiche Array hätte also auch folgendermaßen deklariert werden können:

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

4.5.3 Ungleichförmige Arrays

Alle oben genannten Arrays haben eine Gemeinsamkeit: Sie sind gleichförmig. Jede Dimension hat die gleiche Anzahl Elemente. Deklariert man beispielsweise ein Array folgendermaßen:

```
int[,] arr = new int[2, 2];
```

dann besitzt dieses Array zwei Dimensionen mit je zwei Werten.

Es ist mit C# allerdings auch möglich, Arrays zu deklarieren, bei denen die Anzahl der Elemente pro Dimension unterschiedlich ist. Solche Arrays nennt man dann ungleichförmige oder *jagged Arrays*.

HINWEIS

In der Online-Hilfe des Visual Studios wird dieser Arraytyp als »verzweigtes Array« bezeichnet. In anderen Programmiersprachen spricht man jedoch auch zumeist von »ungleichförmigen« Arrays, weshalb dieser Begriff auch hier verwendet wird. Die Übersetzungsstrategie von Microsoft im Bezug auf die Online-Hilfe ist manchmal eine undurchsichtige Sache ...

Ein ungleichförmiges Array wird als »Array eines Arrays« deklariert. Derartige Arrays sind allerdings mit Vorsicht zu genießen. Es gibt die Möglichkeit, die Verwendung ist aber häufig nicht notwendig oder sinnvoll.

Bei der Deklaration eines ungleichförmigen Arrays wird zunächst die Anzahl der Elemente der ersten Dimension festgelegt. Danach erfolgt die Initialisierung der Elemente mit jeweils einem weiteren Array, für das dann die Größe festgelegt wird:

```
int[][] myArray = new int[2];
int[0] = new int[3];
int[1] = new int[5];
int[2] = new int[7];
```

Da es sich dabei eigentlich um Deklarationen herkömmlicher Arrays handelt, ist es auch möglich, die Werte gleich bei der Deklaration festzulegen. Das funktioniert dann analog zu einem eindimensionalen Array:

```
int[][] myArray = new int[2];
int[0] = new int[] {1, 2, 4, 3};
int[1] = new int[] {2, 7};
int[2] = new int[] {3, 9, 1, 2, 4, 2};
```

Schließlich ist es auch möglich, die Initialisierung eines Arrays wie bei den anderen Array-Arten auch direkt an die Deklaration anzuhängen. Verkürzte Schreibweisen sind auch hier wieder erlaubt. Die Deklaration zweier (eigentlich gleicher) ungleichförmiger Arrays sieht folgendermaßen aus:

```
int[][] myArray1 = new int[] {new int[3],
                              new int[5],
                              new int[7] };
```

```
int[][] myArray2 = { new int[3],
                     new int[5],
                     new int[7] };
```

Möglich, aber wiederum verkomplizierend, ist, statt eindimensionaler Arrays mehrdimensionale Arrays zu verwenden. Eine solche Deklaration könnte auf die gleiche Art wie schon beschrieben vorgenommen werden (nur eben mit einem mehrdimensionalen Array). Sie sehen aber sicherlich schon jetzt, dass ein »jagged« Array eine komplizierte Sache werden kann, weshalb diese Möglichkeit nur selten benutzt werden sollte.

4.5.4 Arbeiten mit Arrays

Arrayinformationen ermitteln

Alle Arrays sind abgeleitet von der Klasse `Array` aus dem Namespace `System`. Damit besitzt jedes Array Methoden und Eigenschaften, mit deren Hilfe verschiedene Werte ermittelt oder Funktionen ausgeführt werden können. Die Größe eines Arrays kann beispielsweise über die Eigenschaft `Length` ermittelt werden:

```
int[] myArray = new int[5];
Console.WriteLine(myArray.Length);  //Liefert den Wert 5.
```

Die Anzahl der Dimensionen eines Arrays ist ebenfalls sofort verfügbar. Die entsprechende Eigenschaft heißt `Rank`.

HINWEIS

Da ein Array eine Klasse ist, können Sie auch auf einfache Art und Weise kontrollieren, ob das Array überhaupt bereits initialisiert wurde:

```
bool arrayIsInitialized = ( aArray != null );
```

Arrays löschen

Auch die Klasse Array selbst bietet einige statische Methoden zur Bearbeitung von Arrays. Eine dieser Methoden ist `Clear()`, die zum Löschen eines Arrays dient. Damit ist nicht gemeint, das Array aus dem Speicher zu löschen, sondern die einzelnen Elemente auf ihren Standardwert zurückzusetzen. Die Methode erwartet das Array, die Nummer des Elements, ab dem gelöscht werden soll, und die Anzahl der zu löschenden Elemente.

```
Array.Clear(myArray,3,6);  // Löscht 6 Elemente ab dem dritten Element
```

Arrays kopieren

Es gibt zwei Arten, ein Array zu kopieren. Die erste Möglichkeit ist eine Kopie des gesamten Arrays, die zweite Möglichkeit das Kopieren nur bestimmter Elemente eines Arrays. Zum Kopieren des gesamten Arrays ist die Methode `Clone()` zuständig. Sie liefert eine In-

stanz der Klasse `Object` zurück, die die Array-Kopie enthält. Der Inhalt muss durch Casting wieder in ein Array des entsprechenden Typs konvertiert werden.

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

int[] arr2 = (int[])arr.Clone();
```

Hinweis

`Clone()` erzeugt eine so genannte *flache Kopie* (shallow copy) eines Arrays. Das bedeutet, dass lediglich die Elemente kopiert werden, nicht aber Objekte, auf die diese Elemente unter Umständen verweisen. Damit verweisen Elemente in einem kopierten Array auf die gleichen Objekte wie die Elemente des Ursprungsarrays.

Das Kopieren einzelner Bestandteile eines Arrays in ein anderes geschieht über die Methode `Copy()` der Klasse `Array`. `Copy()` liefert zwei Möglichkeiten, Elemente zu kopieren. Bei der ersten Möglichkeit wird eine bestimmte Anzahl Elemente von einem in das andere Array transferiert, wobei die Anzahl angegeben werden kann.

Bei der zweiten Möglichkeit ergeben sich weitere Parameter. Hier kann angegeben werden, wie viele Elemente ab welcher Position im ersten Array in das zweite Array kopiert werden. Weiterhin wird auch angegeben, an welche Position im zweiten Array sie kopiert werden. Das folgende Beispiel zeigt die Anwendung dieser Methoden:

```
int[] arr = new int[10];

... // Befüllen der Elemente

int[] arr2 = new int[5];

// Kopieren von zwei Elementen beginnend bei Element 0:
Array.Copy(arr, arr2, 2);

// Kopieren von Elementen von einem bestimmten Index:
Array.Copy(arr, 1, arr2, 0, 2);
```

Die letzte Anweisung kopiert 2 Elemente ab Position 1 im ersten Array nach Position 0 im zweiten Array.

Arrayinhalte umdrehen

Die Methode `Array.Reverse()` dreht ein Array um. Das letzte Element wird zum ersten Element und umgekehrt (natürlich wird die gesamte Reihenfolge umgedreht, nicht nur zwei Elemente). Vor allem bei sortierten Arrays kann diese Möglichkeit sinnvoll sein, beispielsweise wenn man die Sortierreihenfolge umkehren will. Statt einer zeitaufwändigen Neusortierung kann das Array so einfach gedreht werden.

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

   // Werte zuweisen

Array.Reverse( arr );
```

Arrays sortieren

Die Methode `Array.Sort()` ermöglicht das Sortieren entweder eines kompletten Arrays oder nur eines Teils desselben. Diese Methode existiert in mehreren Varianten. Für Elemente, die selbst keine Vergleichsroutine implementieren (z.B. Instanzen eigener Klassen), können Sie mithilfe des Interfaces `IComparer` die Sortierung beeinflussen. Da Interfaces noch nicht besprochen wurden, erfolgt an dieser Stelle hierzu noch kein Beispiel. Mehr über Interfaces erfahren Sie in Kapitel 7 ab Seite 203.

Das folgende Beispiel sortiert ein Array aus `string`-Elementen und gibt das sortierte Array auf der Konsole aus.

```
using System;

namespace StringSort {

  class Class1 {

    [STAThread]
    static void Main(string[] args) {
      // Deklaration
      string[] values = new string[5];

      // Werte einlesen
      for (int i=0;i<5;i++) {
        Console.Write("Wert "+(i+1).ToString()+": ");
        values[i] = Console.ReadLine();
      }

      // Sortieren
      Array.Sort(values);

      // Ausgabe
      foreach (string s in values)
        Console.WriteLine(s);

      Console.ReadLine();
    }
  }
}
```

4.5.5 Syntaxzusammenfassung

Arrays deklarieren und verwenden	
`Datentyp[] arr = new Datentyp[n]`	Deklariert und initialisiert ein Array mit `n` Elementen des Datentyps `Datentyp`. Die Werte der Elemente werden auf den Standardwert des Datentyps gesetzt.
`Datentyp[,] arr = new Datentyp[Wert1, Wert2];`	Deklariert und initialisiert ein zweidimensionales, gleichförmiges Array.
`Datentyp[][] arr;`	Deklariert ein ungleichförmiges, oder wie es in der Dokumentation heißt, »verzweigtes« Array. Die Anzahl der Elemente wird erst bei der Initialisierung festgelegt.
`Datentyp[] arr = new Datentyp[] {Wert1,Wert2,Wert3};`	Deklariert ein eindimensionales Array des Typs `Datentyp` mit drei Elementen, deren Werte durch `Wert1`, `Wert2` und `Wert3` vorgegeben sind.

Eigenschaften und Methoden der Klasse Array (aus System)	
`Array.BinarySearch(arr, obj)` `Array.BinarySearch(arr, o, ICompare)`	durchsucht das Feld `arr` nach dem Eintrag `obj`. Die Methode setzt voraus, dass das Feld sortiert ist, und liefert als Ergebnis die Indexnummer des gefundenen Eintrags. Optional kann eine eigene Vergleichsmethode angegeben werden.
`Array.Clear(arr, n, m)`	setzt `m` Elemente beginnend mit `arr[n]` auf den Standardwert des zugrunde liegenden Datentyps.
`arr2 = (Datentyp[])(arr1.Clone())`	weist `arr2` eine Kopie von `arr1` zu.
`Array.Copy(arr1, n1, arr2, n2, m)`	kopiert `m` Elemente vom Feld `arr1` in das Feld `arr2`, wobei `n1` der Startindex in `arr1` und `n2` der Startindex in `arr2` ist.
`arr = Array.CreateInstance ( Typ, n [,m [,o]] )`	erzeugt ein Feld der Größe `(n,m,o)`, wobei in den einzelnen Elementen Objekte des Typs `type` gespeichert werden können.
`arr.GetLength(dimension);`	ermittelt die Anzahl der Elemente der Dimension `dimension` des Arrays `arr`.
`arr.GetUpperBound(dimension);`	liefert die obere Grenze der Dimension `dimension` des Arrays `arr`.
`arr.GetLowerBound(dimension);`	liefert die untere Grenze der Dimension `dimension` des Arrays `arr`. Diese Funktion ist zwar selten nützlich, es kann jedoch vorkommen, dass ein Array nicht die Untergrenze 0 hat.
`Array.Reverse(arr)`	vertauscht die Reihenfolge der Elemente des Arrays `arr`.
`arr.SetValue(data, n [,m [,o]])`	speichert im Element `arr(n, m, o)` den Wert `data`.
`Array.Sort(arr [,ICompare] )`	sortiert `arr` (unter Anwendung der Vergleichsfunktion des `ICompare`-Objekts, falls angegeben).

`arr.Length`	ermittelt die Gesamtzahl der Elemente des Arrays `arr` in allen Dimensionen.
`arr.Rank`	gibt die Anzahl der Dimensionen des Arrays `arr` an.

4.6 Aufzählungstypen (enum)

4.6.1 Grundlagen

Aufzählungen dienen dazu, mehrere konstante Werte zu gruppieren. Der Vorteil dieser Methode ist, dass die Konstanten in einem Datentyp gekapselt sind und somit als Klartext verwendet werden können. Da es sich um einen eigenständigen Datentyp handelt, kann die Deklaration auch außerhalb einer Klasse geschehen. Enums können damit global zur Verfügung gestellt werden.

Die Verwendung von Aufzählungen erhöht die Lesbarkeit eines Programms. Die Zeile

```
aDay = WeekDays.Monday;
```

ist leichter zu lesen als die Zeile

```
aDay = 1;
```

Leichter zu lesen bedeutet in diesem Falle, dass derjenige, der die Zeile liest, sofort weiß, welcher Tag denn jetzt wirklich zugewiesen wird. Um den eigentlichen Wert muss er sich nicht mehr kümmern. Wenn die Aufzählung konstant über das Programm hinweg verwendet wird, sind die Werte immer korrekt, aber vor allem auch immer lesbar.

Da, wie schon öfter angesprochen, in .NET alles eine Klasse ist, wundert es nicht, dass es auch für Aufzählungen eine Basisklasse gibt. Diese befindet sich im Namespace `System` und heißt `Enum`. Der Unterschied zum Schlüsselwort `enum` besteht nur darin, dass der erste Buchstabe großgeschrieben wird. Die Klasse `Enum` bietet einige Methoden, die im Zusammenhang mit Aufzählungen nützlich sind.

Da es sich bei einem `enum` um einen benutzerdefinierten Datentyp handelt, kann dieser sowohl außerhalb einer Klasse als auch innerhalb einer Klasse deklariert werden. Für einen `enum`, der innerhalb einer Klasse deklariert wird, gilt, dass es sich dabei immer um einen statischen Typ handelt, d.h. der Zugriff erfolgt über die Klasse selbst (nicht über eine Instanz der Klasse).

HINWEIS

Das .NET Framework bietet in den Namespaces `System.Collections` und `System.Collections.Specialized` eine umfangreiche Anzahl verschiedener Listen an (z.B. einen `Stack`, eine `ArrayList` usw.). Diese Listen werden oftmals ebenfalls als Aufzählungen bezeichnet, was eigentlich nicht richtig ist. In diesem Buch wird daher streng zwischen den Begriffen »Liste« bzw. »Collection« und »Aufzählung« unterschieden.

4.6.2 Deklaration und Anwendung

Die Deklaration einer Aufzählung hat folgende Syntax:

```
enum <Bezeichner> [:<Datentyp>] { <enum-Liste> }
```

Der Datentyp ist optional, standardmäßig wird der Datentyp `int` verwendet. Die Deklaration einer einfachen Aufzählung sieht folgendermaßen aus:

```
enum WeekDays {
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday
}
```

Dabei wird `WeekDays.Sunday` automatisch der Wert 0 zugewiesen und für jeden weiteren Eintrag um eins erhöht. `WeekDays.Monday` enthält dementsprechend den Wert 1, `WeekDays.Tuesday` den Wert 2 usw.

Die Zuweisung eines anderen Datentyps als Grundlage für die Aufzählung ist ebenfalls möglich. Es muss sich dabei um einen zählbaren Datentyp, also einen der Datentypen handeln, die ganze Zahlen repräsentieren. Wie aus der Syntaxbeschreibung folgt, wird dieser nach dem Bezeichner angegeben:

```
public enum WeekDays : long {
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday
  Saturday
}
```

Der Datentyp ist jetzt als `long` (`System.Int64`) festgelegt. Die Werte innerhalb der Aufzählung sind allerdings die gleichen geblieben, es wird mit 0 begonnen und dann in Einerschritten weitergearbeitet.

Benutzerdefinierte Werte

Die verwendeten Werte sind nicht verbindlich. Für die Konstanten von Aufzählungen gilt, dass sich der Wert auch individuell festlegen lässt. Dabei sind mehrere Szenarien möglich:

- Nur der erste Wert wird verbindlich festgelegt. Alle anderen Werte werden automatisch zugewiesen, indem der Initialwert jeweils um eins erhöht wird.
- Alle Werte werden verbindlich zugewiesen, jeder Wert ist eindeutig

- Alle Werte werden verbindlich zugewiesen, wobei Werte doppelt vorkommen.

Bei einer Aufzählung, die Monate darstellen soll, könnte es beispielsweise nützlich sein, die Anzahl der Tage des jeweiligen Monats als Wert festzulegen. Dass es dabei zu Dopplungen kommt, ist in diesem Fall nicht relevant, da von außen nur die Konstanten sichtbar sind.

```
public enum Months {
  Januar = 31,
  Februar = 28,
  Maerz = 31,
  April = 30,
  Mai = 31,
  Juni = 30,
  Juli = 31,
  August = 31,
  September = 30,
  Oktober = 31
  November = 30,
  Dezember = 31
};
```

Hinweis

Bei der Zuweisung mehrerer gleicher Werte an verschiedene Konstanten ist allerdings Vorsicht geboten. Diese Vorgehensweise ist zwar hilfreich an den Stellen, an denen es angebracht und gewünscht ist, kann aber zu Fehlern an den Stellen führen, wo die Werte wirklich eindeutig sein sollen. Da es eine korrekte und erlaubte Vorgehensweise ist, wird kein Fehler angezeigt.

Es ist nicht notwendig, alle Werte explizit zuzuweisen. Beispielsweise könnte es möglich sein, dass die konstanten Werte statt automatisch mit 0 mit einer anderen Zahl beginnen sollen. In diesem Fall genügt es, dem ersten Element eine Zahl zuzuweisen, C# erhöht diesen Wert dann für jedes nachfolgende Element um 1.

```
public enum Quality {
  LowQuality = 1, // hat automatisch Wert 1
  MediumQuality,  // hat automatisch Wert 2
  HighQuality,    // hat automatisch Wert 3
};
```

Eine solche Zuweisung ist auch innerhalb der Aufzählung möglich.

```
public enum Quality {
  LowQuality,          // Wert 0
  MediumQuality = 10,  // Wert 10
  HighQuality          // automatisch Wert 11
};
```

4.6.3 Bitfelder

Normalerweise schließen sich die Werte einer Aufzählung gegenseitig aus, d.h. es kann immer nur ein Wert relevant sein. Es gibt aber Situationen, in denen durchaus mehrere Werte einer Aufzählung gleichzeitig Anwendung finden können. Ein solches Beispiel ist die Aufzählung `FileAttributes` aus dem Namespace `System.IO`. Diese Aufzählung (die eigentlich ein Bitfeld ist) repräsentiert die verschiedenen Attribute einer Datei, und eine Datei kann ja bekanntlich mehrere Attribute aufweisen.

Alles, was zur Umwandlung einer »normalen« Aufzählung in ein Bitfeld zu tun ist, ist das Attribut `Flags` zu verwenden. Bei Attributen handelt es sich um Klassen, die die Eigenschaften beispielsweise von Datentypen erweitern. Die Deklaration eines solchen Bitfelds sieht folgendermaßen aus:

```
[Flags()]
enum Bitfeld { ... }
```

Damit die Auswertung eines solchen Bitfelds effizient ist, sollten Sie als Werte Zweierpotenzen zuweisen. Im Falle des Bitfelds `FileAttributes` könnte das z.B. folgendermaßen aussehen:

```
[Flags()]
public enum FileAttributes {
  Archive  = 0,
  Hidden   = 1,
  Readonly = 2,
  System   = 4
};
```

Sie können zur Zuweisung auch die bereits deklarierten Konstanten verwenden. Wenn Sie beispielsweise einen Wert `All` hinzufügen wollen, der alle Attribute zurückliefert, können Sie das folgendermaßen tun:

```
[Flags()]
public enum FileAttributes {
  Archive  = 0,
  Hidden   = 1,
  Readonly = 2,
  System   = 4,
  All = Archive | Hidden | Readonly | System
};
```

Wertzuweisung und -vergleich

Das Zuweisen eines Werts an ein Bitfeld gestaltet sich etwas anders als bei einer herkömmlichen Aufzählung. Da dort die Werte eindeutig sind, genügt eine Zuweisung der Art

```
WeekDays myDay = WeekDays.Monday;
```

Bei Bitfeldern ist es nicht ganz so einfach, da eine solche Zuweisung einen bereits vorhandenen Wert überschreiben würde. Um also einen Wert hinzuzufügen, muss der Operator | (logisches Oder) herangezogen werden:

```
FileAttributes myAttributes = FileAttributes.Archive;

// Hinzufügen eines weiteren Bits
myAttributes = myAttributes | FileAttributes.Hidden;
```

Die Funktionsweise dieser Verknüpfung ist recht einfach zu verstehen: Der Initialwert wird mit einem weiteren Wert logisch oder-verknüpft. Im Resultat ist damit jedes Bit gesetzt, das entweder im ersten oder im zweiten Wert 1 ist:

```
Erster Wert:  0001
Zweiter Wert: 0010
Resultat:     0011
```

Bei der Kontrolle muss immer darauf kontrolliert werden, ob ein bestimmtes Bit gesetzt ist. Das geschieht durch den Operator & (logisches Und).

```
// Kontrolle auf ein Attribut
if ( (myAttributes & FileAttributes.Hidden) = FileAttributes.Hidden )
  // FileAttributes.Hidden ist gesetzt.
```

Die Kontrolle basiert auf binärer Logik. Beim und-Vergleich enthält das Resultat jedes Bit, das in beiden Werten gesetzt ist. Dieses Resultat muss daher zwingend dem Wert entsprechen, auf den verglichen wird, wenn das entsprechende Bit gesetzt ist.

```
Erster Wert:  0011
Zweiter Wert: 0010
Resultat:     0010
```

Hinweis

Sie sollten einen Vergleich bei Bitfeldern immer auf die beschriebene Art durchführen. Dies erlaubt Ihnen nämlich, die Angabe der Werte zu vernachlässigen, d.h. es ist nicht notwendig, jedem Wert eine Zweierpotenz zuzuweisen. Bei einer anderen häufig verwendeten Vergleichsart, nämlich

```
if ((myAttributes & FileAttributes.Hidden) != 0)
```

ist die Verwendung von Zweierpotenzen zur eindeutigen Unterscheidung der Werte zwingend notwendig.

Hinweis

Weitere Informationen zu Attributen (die häufig auch als Meta-Attribute bezeichnet werden, da durch sie die Metadaten einer Assembly erweitert werden können) finden Sie in Abschnitt 6.11 ab Seite 193. Dort erfahren Sie auch, wie Sie eigene Attribute erstellen und verwenden können.

4.6.4 Arbeiten mit Aufzählungen

Aufzählungen sind generell von der Klasse `Enum` aus dem Namespace `System` abgeleitet. Diese Klasse bietet einige statische Methoden, mit denen Sie mehr Informationen über Aufzählungen ermitteln bzw. deren Verwendung optimieren können. In diesem Abschnitt werden einige dieser Methoden erläutert.

HINWEIS

Manchmal ist es in einem Fachbuch nicht ganz einfach, Vorgehensweisen zu beschreiben ohne ein etwas detaillierteres Wissen als zu diesem Zeitpunkt vorhanden vorauszusetzen. So ist es auch hier. Einige Dinge, die in diesem Abschnitt vorausgesetzt werden, werden es in den folgenden Kapiteln genauer erklärt. Dieser Abschnitt richtet sich daher an diejenigen, die bereits ein wenig Erfahrung mit C# gesammelt haben.

Als Basis für alle in der Folge beschriebenen Methoden dient die Aufzählung `WeekDays`, die wie folgt definiert ist:

```
public enum WeekDays {
  Montag,
  Dienstag,
  Mittwoch,
  Donnerstag,
  Freitag,
  Samstag,
  Sonntag
}
```

Namen und Werte ermitteln

Die Methode `Enum.GetNames()` liefert alle Namen der in einer Aufzählung deklarierten Konstanten. Analog liefert die Methode `Enum.GetValues()` die dazugehörigen Werte. Beide werden gleich sortiert, sodass eine Ausgabe prinzipiell leicht möglich wäre. Sowohl `GetValues()` als auch `GetNames()` erwarten als Parameter den Typ der Aufzählung, der in diesem Beispiel mithilfe des Operators `typeof` ermittelt wird.

Während `GetNames()` jedoch ein `string`-Array zurückliefert, liefert `GetValues()` eine Instanz der `Array`-Klasse. Das ist verständlich, denn der Datentyp ist ja nicht bekannt (es könnte sich z.B. um `byte` handeln). Daher ist ein wenig Umwandlungsarbeit angesagt. Über die Instanzmethode `GetValue()` der Klasse `Array` wird der Wert ermittelt. Dieser muss dann noch durch Casting in den korrekten Wert überführt werden, da `GetValue()` den Datentyp `object` zurückliefert.

```
string[] names;
Array values;

names = Enum.GetNames(typeof(WeekDays));
values = Enum.GetValues(typeof(WeekDays));
```

```
for (int i=0;i<names.Length;i++)
  Console.WriteLine("{0}:{1}",names[i],(int)values.GetValue(i));
```

Soll der Name eines bestimmten, vorhandenen Werts ermittelt werden, bietet sich die Methode `Enum.GetName()` an. Sie erwartet als Parameter den Typ der Aufzählung sowie den Wert des Elements, dessen Name ermittelt werden soll.

```
WeekDays myDay = WeekDays.Mittwoch;
Console.WriteLine(Enum.GetName(typeof(WeekDays),myDay));  // liefert "Mittwoch"
```

Wert aus dem Namen ermitteln

Die Verwendung von Aufzählungen ist häufig zum Füllen von Auswahlfeldern (Comboboxen) oder Listboxen interessant. Wenn ein solcher Wert ausgewählt wird, ist allerdings nur der Name des entsprechenden Aufzählungswerts bekannt, der dann üblicherweise auch noch als `string` vorliegt.

Die Methode `Enum.Parse()` ermöglicht es, aus einem solchen String, der dem Namen eines der enthaltenen Elemente enspricht, den dazugehörigen Wert zu ermitteln. Über einen optionalen booleschen Parameter kann zusätzlich angegeben werden, ob Groß-/Kleinschreibung berücksichtigt werden soll oder nicht. Das folgende Beispiel zeigt, wie der entsprechende Wert ermittelt werden kann.

```
string myDay = ComboBox1.Text;
int theValue = (int)(Enum.Parse(typeof(WeekDays), myDay, true));
Console.WriteLine("Wert von {0}: {1}", myDay, theValue);
```

Da ein Wert des Typs `object` zurückgeliefert wird, muss hier ein Casting durchgeführt werden. Alternativ könnte aber auch nach `WeekDays` gecastet werden:

```
WeekDays theValue = (WeekDays)(Enum.Parse(typeof(WeekDays), myDay, true));
```

`Parse()` kommt auch mit Bitfeldern zurecht. In diesem Fall müssen alle Werte innerhalb des Strings übergeben werden, getrennt durch Komma. Der zurückgelieferte Wert entspricht dann einem Bitfeld, in dem die angegebenen Werte gesetzt sind. Bei folgendem Bitfeld:

```
[Flags()]
public enum Privileges {
  Read,
  Write,
  Search
};
```

können Werte auf die folgende Weise zugewiesen und kontrolliert werden:

```
Privileges priv;
priv = (Privileges)(Enum.Parse(typeof(Privileges),"Read,Write"));
Console.WriteLine( (priv & Privileges.Read) == Privileges.Read  );
Console.WriteLine( (priv & Privileges.Write) == Privileges.Write );
Console.WriteLine( (priv & Privileges.Search) == Privileges.Search );
```

Die Ausgabe ist:

```
true
true
false
```

Kontrollieren, ob ein Wert definiert ist

Zum Überprüfen, ob ein bestimmter Wert in der Aufzählung enthalten ist, bietet die Klasse `Enum` die Methode `IsDefined()` an, die einen booleschen Wert zurückliefert. Übergeben werden der Typ der Aufzählung und der zu ermittelnde Wert. `IsDefined()` ist grundsätzlich nicht geeignet für Bitfelder.

```
bool check = Enum.IsDefined(typeof(WeekDays),5);
```

4.6.5 Syntaxzusammenfassung

Aufzählungen deklarieren und verwenden

```
enum AufzType <Datentyp> {
  element1 [ = wert1 ]
  element2 [ = wert2 ]
  ...
}
```

Deklariert eine Aufzählung. Den Elementen werden automatisch durchlaufende Zahlen zugewiesen, wenn Sie nicht explizit eigene Werte angeben. Der Datentyp kann optional angegeben werden und muss ein ordinaler Typ sein (`int`, `long`, `byte` oder `short`). Standard für alle Aufzählungen ohne Datentyp ist `int`.

```
[Flags()]
enum AufzType {
  element1 [ = 1 ]
  element2 [ = 2 ]
  element3 [ = 4 ]
   ...
}
```

Deklariert ein Bitfeld, d.h. eine Aufzählung, bei der mehrere Werte angenommen werden können. Wenn explizit Werte angegeben werden, sollte es sich um Zweierpotenzen handeln, in jedem Fall aber müssen sie eindeutig sein (wegen der Kontrolle).

Eigenschaften und Methoden der Klasse Enum (aus System)

`Enum.GetNames(typeof(AufzType));`	liefert ein Zeichenkettenfeld, das die Namen aller Konstanten der Aufzählung `AufzType` enthält.
`Enum.IsDefined(typeof(AufzType), n);`	testet, ob `n` ein gültiger Wert einer `Enum`-Konstante von `AufzType` ist.
`aufzObj = (AufzType)(Enum.Parse( typeof(AufzType), s [,true|false] );`	wertet die Zeichenkette `s` aus und liefert die entsprechende `Enum`-Konstante von `AufzType`. `AufzObj` ist eine Instanz von `AufzType`.
`Enum.GetValues(typeof(AufzType));`	Liefert alle Werte der Aufzählung `AufzType`.
`Enum.GetName(typeof(AufzType), AufzObj);`	Liefert den Namen des Elements mit dem Wert `AufzObj` der Aufzählung `AufzType`.