Der schnelle Einstieg in C#

Inhalt

Vorwort

Liebe Leserin, lieber Leser,

mit dem Erwerb dieses Buches haben Sie den ersten Schritt in die Welt der Programmierung gewagt. Ich werde Ihnen so kompakt und verständlich wie möglich, alle wichtigen Grundlagen der Programmiersprache C# beibringen, sodass Sie direkt damit anfangen können, eigene kleine Anwendungen zu schreiben. Sie werden sehen, dass die Programmiersprache C# gar nicht so schwer ist wie es auf dem ersten Blick scheint. Das Buch ist so strukturiert, dass alle Themen aufeinander aufbauen und es werden absolut keine Vorkenntnisse benötigt. Dieses Buch richtet sich also wirklich an absolute Einsteiger in die Programmierung, es kann aber trotzdem auch als ein kleines Nachschlagwerk verwendet werden. Ich wünsche Ihnen viel Spaß beim Lesen und einen angenehmen Einstieg in die Programmiersprache C#!

Was ist das .NET-Framework?

Bevor wir die Grundlagen der Sprache erlernen können, müssen wir natürlich erstmal wissen was C# überhaupt ist. C# ist eine Programmiersprache von Microsoft und gehört zur sogenannten .NET-Technologie. Die .NET-Technologie ist eine Sammlung von Frameworks (Programmiergerüsten), Programmiersprachen und Werkzeugen für die Software-Entwicklung. C# verwendet man immer zusammen mit dem .NET-Framework, welches ebenfalls Teil der .NET-Technologie ist und dementsprechend können wir nicht direkt die Programmiersprache lernen. Wir müssen uns zuerst einmal

verinnerlichen was das .NET-Framework überhaupt ist, da C# ohne diesem gar nicht existieren könnte.

Achtung: Wenn wir mit C# Programmieren, arbeiten wir mit dem .NET-Framework!

Das .NET-Framework

Das .NET-Framework stellt verschiedene Werkzeuge für die Entwicklung und Ausführung von eigener Software zur Verfügung. Dazu zählen zum Beispiel verschiedene Programmiersprachen, eine gigantische Klassenbibliothek (dazu später mehr) und eine Laufzeitumgebung, mit welcher man die eigene Software ausführen kann. Auf die Laufzeitumgebung möchte ich an dieser Stelle zuerst eingehen, da diese die größte Besonderheit des .NET-Frameworks darstellt.

Die Laufzeitumgebung

Die Laufzeitumgebung des .NET-Frameworks heißt „Common Language Runtime“ (kurz CLR). Sie wird dazu verwendet, um Software auszuführen, die mit dem .NET-Framework entwickelt wurde. Aber wie funktioniert das eigentlich? Schauen wir uns zuerst einmal an, wie ein Programm auf dem ursprünglichen Weg ohne .NET, programmiert und ausgeführt wird.

Beim Programmieren gibt man seinem Computer sogenannte Anweisungen (Befehle), die dieser durchführen soll um ein bestimmtes Ziel zu erreichen. Jetzt gibt es hier allerdings ein Problem: Wir verstehen keine Maschinensprache und der Computer keine Menschensprache. Dementsprechend musste

man einen Weg finden, diese Sprachbarriere zu überwinden. Die Lösung sind Programmiersprachen wie C#, Java, C++, C usw...

Programmiersprachen dienen sozusagen als Kompromiss. Sie können gut in Maschinensprache umgewandelt werden mithilfe eines Compilers (den Übersetzungsvorgang nennt man auch „kompilieren“) und trotzdem sind sie für uns Menschen verständlich genug, sodass wir damit Computer-Programme formulieren bzw. coden können. Der Weg von einer Programm-Idee zum fertigen Produkt sieht dabei folgendermaßen aus:

- Zuerst hat der Programmierer eine Idee für ein Programm.
- Danach programmiert er dieses Programm mit einer Programmiersprache.
- Der Quellcode für das Programm, also der Code, den der Programmierer geschrieben hat, wird mit einem Compiler in Maschinensprache übersetzt.
- Das Ergebnis dieses Kompiliervorgangs wird in einer ausführbaren Datei gespeichert (z.B. einer .exe-Datei).

Im .NET-Framework sieht die Sache allerdings ein bisschen anders aus. Dank der „Common Language Runtime“, also der Laufzeitumgebung, kommt noch ein Zwischenschritt dazu. Ein Programm wird hier nicht direkt vom Programmcode in Maschinensprache übersetzt, sondern zuerst in eine Zwischensprache. Diese Zwischensprache nennt man „Intermediate Language“ und sie wird erst zur Laufzeit (also während unsere programmierte Anwendung läuft) in den vom

Computer verstandenen Maschinencode übersetzt. Der Weg den ein Programm im .NET-Framework geht sieht also folgendermaßen aus:

- Zuerst hat der Programmierer eine Idee für ein Programm.
- Danach programmiert er dieses Programm mit einer .NET-Programmiersprache wie z.B. C#.
- Dieser Programmcode wird vom Compiler in die „Intermediate Language“ übersetzt.
- Dieser Zwischencode wird in eine ausführbare Datei geschrieben.
- Wenn der Benutzer das Programm ausführt, wird dieses von der Common Language Runtime während der Laufzeit in Maschinencode übersetzt.

Dieser Zwischenschritt hat den Sinn, dass man an einem Projekt mit mehreren vom .NET-Framework unterstützten Programmiersprachen gleichzeitig arbeiten kann. Der Zwischencode hat nämlich immer die gleiche Form, egal welche Programmiersprache verwendet wird. Deswegen kommt es bei der Arbeit mit verschiedenen Sprachen zu keinerlei Konflikten. Programmierer A kann also beispielsweise mit C# arbeiten, während Programmierer B die Sprache Visual Basic bevorzugt.

Die Klassenbibliothek

Eine weitere sehr wichtige Komponente des .NET-Frameworks ist die umfangreiche Klassenbibliothek. Diese enthält über 10000 Klassen, welche bereits vorprogrammierte Funktionen beinhalten. Diese Klassen kann man für seine eigenen Projekte

verwenden und man muss sich dementsprechend nicht um jede einzelne Kleinigkeit selber kümmern, wenn man eine Anwendung programmiert. Ein paar Beispiele für solche Klassen wären die folgenden:

- Math: Enthält viele einfach zu verwendende Mathematischen Funktionen und Konstanten.
- Random: Ermöglich die Erzeugung von Zufallszahlen.
- Console: Ermöglicht die Ein- und Ausgabe von Text in Konsolen-Anwendungen.
- FileInfo: Ermöglicht die Arbeit mit Dateien.
- Und viele mehr…

Sie merken also, dass fast alle grundlegenden Dinge, die jedes Programm für gewöhnlich benötigt, schon vorhanden sind. Man kann sich dementsprechend vollkommen auf die Features seiner Anwendungen fokussieren und an diesen arbeiten, ohne ständig irgendwelche grundlegenden Dinge programmieren zu müssen. Das beschleunigt die Entwicklungszeit enorm!

Achtung: Auf Klassen gehen wir natürlich noch in einem eigenen Kapitel ein!

Was ist C#?

C# ist eine Programmiersprache und Teil des .NET-Frameworks. Es handelt sich dabei um eine sogenannte Allzweckprogrammiersprache, was bedeutet, dass die Sprache nicht für ein bestimmtes Anwendungsgebiet konzipiert wurde. Theoretisch kann man mit C# also „alles“ programmieren. Das Hauptanwendungsgebiet der Sprache sind Windows-

Anwendungen, dennoch findet man sie auch in anderen Bereichen der Software-Entwicklung wieder. Hier ist mal eine Auflistung für die häufigsten Anwendungsgebiete von C#:

- Konsolenanwendungen für die Automatisierung von alltäglichen Arbeiten
- Anwendungen mit grafischer Benutzeroberfläche (GUI-Anwendungen)
- Videospiele
- Windows-Dienste (Services)
- Webanwendungen für Windows-Server
- Und vieles mehr…

C# wird hauptsächlich für die Entwicklung von Windows-Software eingesetzt. Im Prinzip kann man damit zwar auch Plattformunabhängige Software entwickeln, dafür benötigt man allerdings eine andere Implementierung des .NET-Frameworks. Ein Beispiel für eine solche Alternativ-Implementierung wäre das „Mono-Projekt“.

C# ist eine vollkommen objektorientiere Programmiersprache und vertritt dementsprechend einen sehr modernen und „menschlichen“ Programmierstil. Auf das Thema Objektorientierung gehe ich in einem eigenen Kapitel noch genauer ein. Falls Sie also noch nicht mit dem Konzept der Objektorientierten Programmierung vertraut sind, lassen Sie sich nicht davon einschüchtern. Das Thema wird in diesem Buch noch umfangreich behandelt.

Sehen wir uns zu guter Letzt noch an, wie C#-Code überhaupt aussieht. Hier ist mal ein kleiner Code-Ausschnitt:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Hallo_Welt
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hallo Welt");
            Console.ReadKey();
        }
    }
}
```

Achtung: Natürlich müssen Sie den gezeigten Code noch nicht verstehen. Ich möchte an dieser Stelle nur zeigen, wie C#-Code überhaupt aussieht.

Wie man auf dem Code-Ausschnitt erkennen kann, ist C# eine typische „C-Sprache“. Der Code wird aufgeteilt in kleine Abschnitte (sogenannte Code-Blöcke), welche mit geschweiften Klammern gebildet werden. Diese Code-Blöcke enthalten Anweisungen (Befehle) für den Computer, die dieser der Reihe nach ausführt. Was genau in dem Code-Beispiel passiert erkläre ich Ihnen, sobald wir unser erstes kleines Programm schreiben. Keine Sorge, bald ist es so weit!

Die benötigte Software installieren

Genug also von der Theorie! In den kommenden Kapiteln werden Sie die Programmiersprache C# endlich kennenlernen. Bevor wir aber so richtig mit dem Programmieren beginnen können, müssen wir uns die benötigte Software herunterladen. Alles was wir brauchen ist im Grunde in einem

Download enthalten. Wir downloaden im Rahmen dieses Buches nämlich „Visual Studio Community“, eine kostenlose Entwicklungsumgebung von Microsoft. Visual Studio ist eine sogenannte IDE (Integrated Development Environment), welche alle benötigten Werkzeuge für die Software-Entwicklung in einem Programm zusammenfasst. Das Programm enthält einen Code-Editor, einen Compiler, Werkzeuge zum Debuggen und vieles mehr.

„Visual Studio Community“ können Sie unter folgender Adresse kostenlos downloaden:

https://www.visualstudio.com/de/vs/community/

Ist das Programm gedownloadet, müssen wir es noch installieren. Die Installation ist allerdings ziemlich einfach.

Im Installer werden Sie nur dazu aufgefordert, die zu installierenden Komponenten auszuwählen. Wählen Sie hier nur die ersten beiden Optionen aus:

- **Entwicklung für die Universelle Windows-Plattform**
- **.NET-Desktopentwicklung**

Klicken Sie anschließend auf „Installieren“ um die Installation zu starten. Der Vorgang wird eine Weile dauern.

Nach der Installation können Sie Visual Studio starten. Beim ersten Start werden Sie darum gebeten sich mit ihrem kostenlosen Microsoft-Account einzuloggen, dieser Schritt kann aber fürs erste auch übersprungen werden. Danach müssen Sie nur noch das gewünschte Farb-Schema der

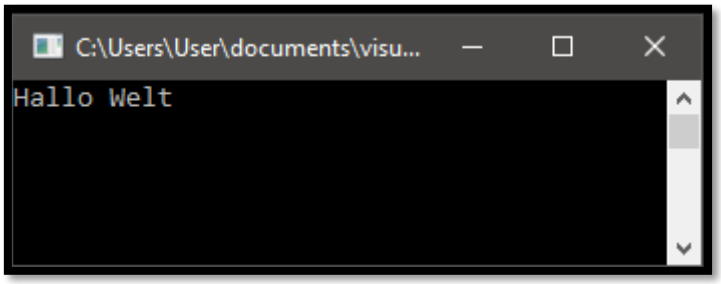
Entwicklungsumgebung auswählen und schon kann es losgehen!

C# Programmcode

In diesem Kapitel werden Sie Ihren ersten Kontakt mit Programmcode haben. Sie werden Ihr erstes kleines Programm entwickeln und erfahren, wie man C#-Code richtig schreibt.

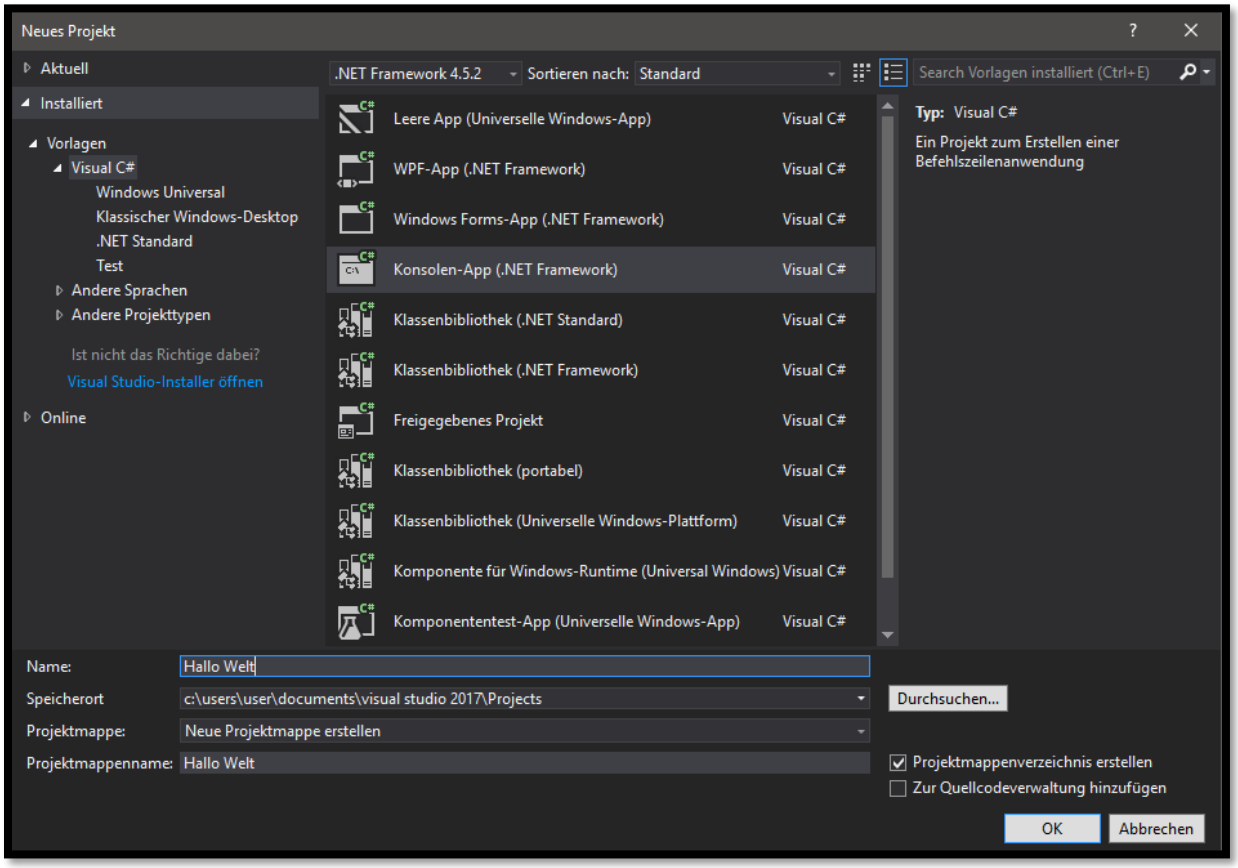
Das erste Programm erstellen

Bevor wir unseren ersten Code tippen können, müssen wir natürlich ein Projekt anlegen. Das Projekt wird eine einfache Konsolenanwendung, welche den Text „Hallo Welt“ ausgibt. Konsolenanwendungen werden wir im gesamten Verlauf des Buches verwenden, da diese die einfachste und schnellste Möglichkeit darstellen, Sprachfeatures auszuprobieren. Sie sind Anwendungen ohne jeglichem Schnick-Schnack wie z.B. einer Grafischen Benutzeroberfläche und man kann sich voll und Ganz auf die eigentliche Programmlogik fokussieren. Konsolenanwendungen stellen also die Ideale Spielwiese eines Programmier-Anfängers dar.



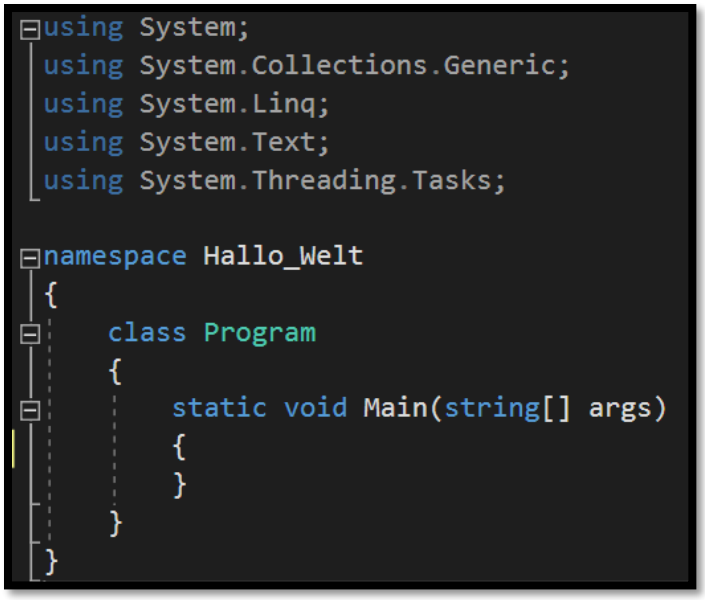
Auf der Abbildung sehen Sie eine einfache Konsole, welche den Text „Hallo Welt“ anzeigt. Dieses Programm werden wir nun nachprogrammieren.

Öffnen Sie dazu Visual Studio und klicken Sie auf **„Datei → Neu → Projekt“**. Daraufhin öffnet sich ein Menü, in welchen Sie die Programmiersprache und den gewünschten Projekttyp auswählen müssen.



In der Mitte des Fensters sehen Sie die verschiedenen Projekttypen, die für die auf der linken Seite gewählte Programmiersprache zur Verfügung stehen. Wählen Sie als Projekttyp die Option „Konsolen-App (.NET-Framework)“ und geben Sie dem Projekt den Namen „Hallo Welt“. Klicken Sie anschließend auf „OK“ um das Projekt zu erstellen.

Wenn Sie alles richtiggemacht haben, öffnet sich der Code-Editor mit folgendem Inhalt:



Dieser vorgenerierte Code ist das Grundgerüst unserer Konsolenanwendung. Der Code mag für einen Anfänger auf dem ersten Blick erschlagend wirken, in Wahrheit ist das Ganze aber ziemlich einfach.

In den ersten Zeilen sehen Sie die sogenannten „Using-Direktiven“. Damit werden sogenannte „Namespaces“ in den Code eingebunden. Ein Namespace kann man sich fürs erste wie einen Ordner vorstellen. Dieser Ordner enthält sogenannte „Klassen“. Auf Klassen gehe ich natürlich auch noch genauer ein und deswegen stellen wir uns diese fürs erste einmal wie eine Datei vor, die Funktionen beinhaltet. Diese von der Klasse bereitgestellten Funktionen können wir für unser eigenes Projekt verwenden. Mit Using-Direktiven binden wir also bereits programmierten Code in unser eigenes Code-Dokument ein, den wir in unserem Programm benutzen möchten.

Nach den Using-Direktiven folgt der Namespace „Hallo_Welt“. Dies ist der Namespace, der alle Klassen für unser Projekt

beinhaltet. Dies ist also der „Ordner“ für die Klassen, die zu unserem Projekt gehören. Der Namespace besitzt einen Code-Block, der mit geschweiften Klammern gebildet wird. Alle Dinge die zu diesem Namespace gehören sollen, müssen in diesen Code-Block hineingeschrieben werden.

Innerhalb des Namespaces befindet sich die Klasse „Program“. Wie bereits erwähnt, werde ich auf Klassen noch in einem eigenen Kapitel eingehen. Eine sehr einfache Erklärung möchte ich dennoch geben: **Eine Klasse bündelt logisch zusammengehörenden Code**. Sie beinhaltet Werte und Funktionen (in der objektorientierten Programmierung Methoden genannt), die logisch zusammengehören. So stellt einem die Klasse „Math“ aus der .NET-Klassenbibliothek nur Methoden und Konstanten zur Verfügung, die mit Mathematik zutun haben. Ein Videospiel könnte auch eine Klasse namens „Auto“ enthalten, die den Code für Autos im Spiel beinhaltet. Darin könnten sich dann beispielsweise Methoden befinden, die das beschleunigen, bremsen usw. ermöglichen. Die Klasse „Program“ in unserem Namespace „Hallo_Welt“ stellt unser Programm dar. Diese Klasse hat auch wieder einen Code-Block in dem sich die sogenannte „Main-Methode“ befindet.

Die Main-Methode ist eine ganz besondere Methode (Funktion). Sie stellt nämlich den Startpunk unseres Programms dar und ist dementsprechend das Herz unserer Anwendung. Eine Methode ist eine Folge von zusammengehörenden Anweisungen, also Befehlen, die unser Computer ausführen soll um eine bestimmte Aufgabe zu erfüllen. So enthält Beispielsweise eine Klasse namens „Auto“

eine Methode „GebeGas()“. Diese Methode würde Anweisungen enthalten, die ein Auto beschleunigen würden. Die Main-Methode ist sozusagen die „Haupt-Methode“ unserer Konsolenanwendung und enthält all die Anweisungen, die unser Programm im Endeffekt zu dem machen was es ist.

Achtung: Jedes C#-Programm muss eine Methode mit dem Namen „Main“ beinhalten. Ohne einer Main-Methode hat unser Programm nämlich keinen Startpunkt.

Anweisungen – Unsere ersten Befehle

Sie haben nun gelernt, dass jede Methode eine Reihe von Anweisungen darstellt. Eine Anweisung ist ein einzelner Befehl an den Computer. Es gibt verschiedene Arten von Anweisungen. Mal wollen wir einen Wert in einen Speicherplatz hineinschreiben, mal wollen wir eine andere Methode aufrufen. Fakt ist, dass jede Anweisung im Grunde einfach nur ein Befehl ist. In C# werden Anweisungen mit einem sogenannten Semikolon „;“ abgeschlossen. Schreiben wir doch einfach mal unsere ersten zwei Anweisungen in die Main-Methode:

```
static void Main(string[] args)
{
    Console.WriteLine("Hallo Welt");
    Console.ReadKey();
}
```

Wir haben in den Code-Block der Main-Methode die Methodenaufrufe für **„Console.WriteLine()“** und **„Console.ReadKey()“** geschrieben. „WriteLine()“ ist eine

Methode aus der Klasse „Console“. Dementsprechend kann man die Verkettung der beiden Wörter mit einem Punkt folgendermaßen interpretieren: „Führe die Methode **WriteLine()** aus der Klasse **Console** aus!“

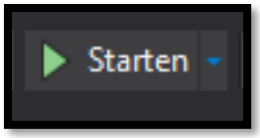
Die Methode WriteLine wird dazu verwendet, um einen Text in die Konsole zu schreiben, bzw. um Text für den Benutzer auszugeben. Wie Sie sehen hat jede Methode ein einfaches Klammerpaar am Ende, in welches man sogenannte Parameter hineinschreiben kann. Im Beispiel der WriteLine()-Methode kann man hier den Text hineinschreiben, der in der Konsole ausgegeben werden soll. In unserem Fall ist das der Text „Hallo Welt“.

Achtung: Textwerte müssen in C# immer zwischen Anführungszeichen stehen.

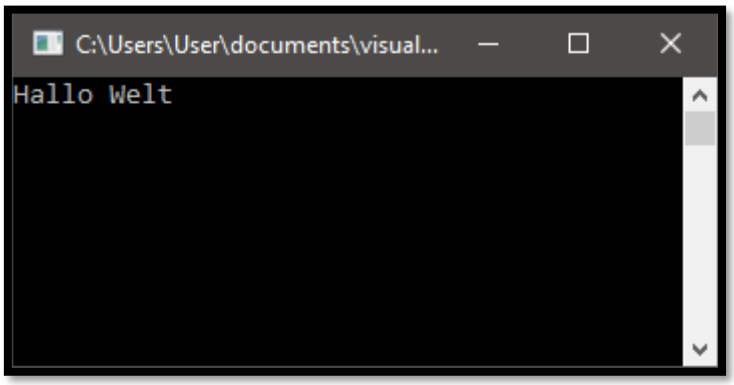
Die zweite Anweisung ist der Aufruf der Methode **ReadKey()**. Auch die ReadKey()-Methode befindet sich in der Klasse Console. Sie hat die Funktion, das Programm solange anzuhalten, bis eine Taste auf der Tastatur gedrückt wurde. Diese Taste wird gelesen und kann bei Bedarf weiterverwertet werden. In unserem kleinen „Hallo Welt“ Programm hat diese Methode die Funktion unser Programm so lange am Leben zu erhalten, bis ein Benutzer eine Taste drückt. Würden wir diese Methode nicht aufrufen, dann würde unser „Hallo Welt“ zwar in der Konsole ausgegeben werden, wir hätten allerdings keine Zeit diesen Text zu lesen, da sich eine Konsolenanwendung sofort beendet, wenn alle Anweisungen abgearbeitet wurden. Die ReadKey()-Methode gilt erst als abgeschlossen wenn eine

Taste gedrückt wurde und eignet sich dementsprechend sehr gut dazu, dass Programm am Leben zu erhalten.

Führen wir das Programm doch einfach mal aus. Klicken Sie dazu einfach auf den Grünen Play-Button mit der Beschriftung „Starten“.



Es sollte sich die Konsole mit folgendem Inhalt öffnen:



Wenn Sie eine Taste auf der Tastatur drücken, schließt sich das Programm sofort.

Löschen Sie einmal die ReadKey()-Anweisung. Sie werden sehen, dass das Programm sich zwar ganz kurz öffnet, aber sofort wieder geschlossen wird.

Sie haben nun also gelernt wie man Anweisungen schreibt. Merken Sie sich, dass Anweisungen immer mit einem Semikolon „;“ beendet werden müssen!

Kommentare im Code

Unser bisheriger Code war ziemlich überschaubar. Stellen Sie sich aber mal vor, Sie haben ein umfangreiches Programm

geschrieben mit tausenden von Anweisungen, hunderten Methoden und vielen verschiedenen Klassen. Da kann es schon mal passieren, dass man die Übersicht verliert und nicht immer genau weiß, was der Sinn von so manchem Code-Abschnitt war. Stellen Sie sich nun vor, dass es in einem so großen Programm einen Fehler gibt, den Sie unbedingt beheben müssen. Ohne einer guten Dokumentation des eigenen Programms kann solch ein Szenario schnell zum Albtraum werden. Genau deswegen schreiben wir Kommentare. Ein Kommentar ist ein Text in unserem Code, welcher nicht vom Computer verarbeitet wird. Er dient einzig und alleine dem Verständnis des Programmierers über die Funktionsweise eines Code-Abschnitts. Kommentare in unserem „Hello World“-Programm könnten zum Beispiel folgendermaßen aussehen:

```
static void Main(string[] args)
{
    //Gebe den Text "Hallo Welt" aus
    Console.WriteLine("Hallo Welt");

    //Halte das Programm an
    Console.ReadKey();
}
```

Was Sie hier sehen sind einzeilige Kommentare. Sie werden eingeleitet mit einem Doppel-Slash „//“ und ziehen sich ab diesem Zeichen durch die ganze restliche Zeile. Kommentare werden im Code grün hervorgehoben.

Wie Sie sehen, versteht man die Funktionsweise von Code-Abschnitten viel besser, wenn Kommentare gesetzt werden. Somit wird die Suche nach einem eventuell auftretenden Fehler um einiges erleichtert.

Neben den einzeiligen Kommentaren gibt es auch die mehrzeiligen Kommentare. Diese strecken sich über mehrere Zeilen.

```
static void Main(string[] args)
{
    /* Das hier ist ein merhzeiliger
     * Kommentar.
     */
    Console.WriteLine("Hallo Welt");
    Console.ReadKey();
}
```

Wie Sie sehen wird ein mehrzeiliger Kommentar mit einem Slash und einem Stern eingeleitet „/*“ und mit einem Stern und Slash beendet „*/“.

Sie haben nun also Kommentare kennengelernt. Ich kann die umfangreiche Nutzung von Kommentaren nur empfehlen, da diese einem sehr viel zukünftigen Ärger ersparen können.

Variablen und Datentypen

In diesem Kapitel werden Sie das wichtigste Konzept der Programmierung kennenlernen. Ohne dem Wissen aus diesem Kapitel werden Sie nicht in der Lage sein, ein sinnvolles Programm zu schreiben. Die Rede ist vom Konzept der

Variablen. Stellen wir uns also zuerst einmal die Frage, was eine Variable überhaupt ist.

Was sind Variablen?

Jedes Programm basiert auf Daten. Nicht umsonst nennt man den Umgang mit Computern auch EDV, was für „Elektronische Datenverarbeitung“ steht. Egal ob man jetzt eine einfache kleine Rechnung im Programm durchführen muss, einen Benutzer mit seinem Namen begrüßen möchte oder die Anzahl von Goldmünzen eines Spielers in einem Videospiel speichern will. Jedes Programm basiert auf Daten die entweder von einem Benutzer eingegeben, oder zur Laufzeit errechnet werden. Diese Daten muss man in der Regel auch speichern, weil man sie immer wieder benötigt. Und genau hier kommen Variablen ins Spiel.

Variablen sind Speicherplätze in unserem Programmcode. Man kann Werte in diese Speicherplätze hineinschreiben und die darin enthaltenen Wert lesen. Man hat also einen lesenden und schreibenden Zugriff auf Variablen. Hier mal ein Beispiel:

Stellen Sie sich vor, Sie entwickeln eine Konsolenanwendung, die den Namen des Benutzers abfragt. Der Benutzer kann seinen Namen in die Konsole eingeben und dann soll er mit diesem begrüßt werden. Hierfür brauchen wir eine Variable, da wir den Text „Hallo {NAME}, willkommen zurück!“ ausgeben möchten. Der Name ist zur Zeit der Programmierung noch nicht bekannt und deshalb können wir auch keinen vorgegebenen Text ausgeben lassen. Wir müssen eine Variable erstellen die „name“ heißt und deren Inhalt dann in den

Begrüßungstext eingefügt werden kann. Die Variable dient also als Platzhalter.

Das Programm würde in der Theorie wie folgt aussehen:

- Frage den Namen des Benutzers ab und schreibe diesen in die Variable „name“. **(Schreibender Zugriff)**
- Gebe den Namen zusammen mit dem Begrüßungstext aus indem er aus der Variable gelesen wird. **(Lesender Zugriff)**

Wie man dieses Programm umsetzt zeige ich Ihnen sobald Sie verstanden haben wie man Variablen im Code erstellen, überschreiben und lesen kann. Genau diese Dinge schauen wir uns jetzt mal an!

Variablen erstellen, überschreiben und lesen

Bevor man eine Variable verwenden kann, muss man diese im Code erst bekannt machen. Den Vorgang des Bekanntmachens einer Variable nennt man „Deklarieren“. Eine Variable muss also zuerst deklariert werden, bevor man sie benutzt.

Die Deklaration einer Variable sieht folgendermaßen aus:

```
static void Main(string[] args)
{
    int alter; //Deklaration
}
```

In diesem Code-Beispiel sehen Sie, wie eine Variable namens „alter“ deklariert wird. Schauen wir uns die Deklaration mal im Detail an.

Das Muster für die Deklaration von Variablen sieht immer folgendermaßen aus: <Datentyp> <Bezeichner>;

Zuerst müssen wir den gewünschten Datentyp für die Variable schreiben. Im Beispielcode wählen wir einen „int“ (Integer), welcher eine Ganzzahl darstellt. Ein Datentyp bestimmt im Grund einfach nur die Art von Wert, die man in eine Variable hineinschreiben kann. Auf Datentypen komme ich aber noch genauer zu sprechen!

Danach müssen wir den Bezeichner für unsere Variable auswählen. Der Bezeichner ist der Name einer Variable, über den man sie zukünftig ansprechen kann. Im Beispielcode nennen wir die Variable „alter“.

Wir haben im Beispiel also eine Variable namens „alter“ erstellt, die in der Lage ist eine Ganzzahl aufzunehmen. Sie speichert also das Lebensalter eines Menschen. Jetzt schreiben wir mal einen Wert in diese Variable.

```
static void Main(string[] args)
{
    int alter; //Deklaration
    alter = 18; //Initialisierung
}
```

In diesem Codeschnipse sehen Sie, wie der zuvor erstellten Variable „alter“ der erste Wert zugewiesen wird. Die erste Zuweisung nennt man „Initialisierung“. Um einer Variable einen Wert zuzuweisen, schreiben wir den Bezeichner gefolgt vom Zuweisungsoperator „=“. Hinter dem „=“ schreiben wir

den Wert den wir zuweisen möchten. Auf diese Art und weise kann man jederzeit eine Zuweisung durchführen. Merken Sie sich jedoch, dass man nur die aller erste Zuweisung „Initialisierung“ nennt.

Deklaration und Initialisierung kann man auch in einer Zeile Code zusammenfassen:

```
static void Main(string[] args)
{
    int alter = 18; //Deklaration und Initialiserung
}
```

Wie Sie in diesem Codeschnipsel sehen können, kann man die Initialisierung (also die erste Wertzuweisung) einfach an die Deklaration anhängen.

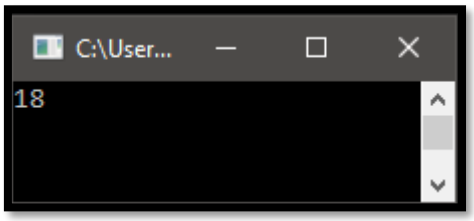
Merke: Eine Variable in einem Programm zu erstellen (bekanntzumachen) nennt man „Deklaration“. Einer Variable einen ersten Wert zuzuweisen nennt man „Initialisierung“.

Eine Variable nützt einem natürlich nichts, wenn man deren Inhalt nicht lesen kann. Schauen wir uns nun an, wie man den Inhalt einer Variable lesen kann.

Eine Variable zu lesen ist im Prinzip ziemlich einfach. Dazu muss man nur den Bezeichner an der Stelle im Code schreiben, an der man den Wert der Variable benötigt. Hier mal ein Beispiel:

```
static void Main(string[] args)
{
    int alter = 18; //Deklaration und Initialiserung
    Console.WriteLine(alter); //Ausgabe von "alter"
    Console.ReadKey(); //Programm anhalten
}
```

Im Beispiel wird zuerst die Variable „alter“ initialisiert. Deren Wert (18) wollen wir anschließend mit „**Console.WriteLine()**“ in der Konsole ausgeben lassen. **WriteLine()** haben Sie bereits kennengerlernt. Man kann dieser Methode zwischen den Klammern einen Wert mitgeben, der in der Konsole ausgegeben wird. Der Wert den wir ausgeben wollen ist der Inhalt der Variable „alter“ und deswegen müssen wir einfach nur den Bezeichner „alter“ als Parameter für **WriteLine()** mitgeben. Anschließend wird noch „**Console.ReadKey()**“ aufgerufen, damit das Programm sich nicht einfach schließt. Die Ausgabe sieht folgendermaßen aus:



Wie Sie sehen, kann man eine Variable also einfach als Platzhalter für einen Wert verwenden, indem man an gewünschter Stelle ihren Bezeichner schreibt.

Datentypen

Sie haben nun gelernt wie man Variablen anlegt, verändert und liest. Es fehlt allerdings noch eine sehr wichtige Grundlage,

die die richtige Arbeit mit Variablen erst wirklich möglich macht. Die Rede ist von den sogenannten Datentypen.

Der Datentyp bestimmt, welche Art von Wert in einer Variable gespeichert werden kann. So kann man beispielsweise bestimmen ob eine Variable Textwerte, Ganzzahlen, Zahlen mit Nachkommastellen, Wahrheitswerte oder andere Arten von Werten beinhalten kann. C# ist eine sogenannte stark typisierte Programmiersprache und deshalb muss schon bei der Erstellung einer Variable ein dazugehörender Datentyp gewählt werden.

Einen Datentyp haben wir bereits kennengelernt. Unserer Variable „alter“ aus dem vorherigen Unterkapitel haben wir den Datentyp „int“ zugewiesen. „int“ steht für Integer und stellt eine einfache Ganzzahl dar. Den Typ haben wir bei Deklaration der Variable angeben müssen:

```
static void Main(string[] args)
{
    int alter; //Deklaration
}
```

Die Variable „alter“ kann also nur Ganzzahl-Werte beinhalten. Die Zuweisung eines Textwertes wäre unzulässig wie man auf dem folgenden Codeschnipsel sehen kann:

```
static void Main(string[] args)
{
    int alter = "Senf";
}
```

Der Textwert „Senf“ wurde rot unterkringelt, weil ein Fehler aufgetreten ist. Man kann nur Integer-Werte (Ganzzahlen) in eine Integer-Variable speichern. Möchte man einen Textwert speichern, muss man den Datentyp „string“ verwenden. Hier mal ein Beispiel dazu:

```
static void Main(string[] args)
{
    string produkt = "Senf";
}
```

Der Variable „produkt“ kann man einen Textwert zuweisen, weil der Datentyp „string“ verwendet wird.

Im Folgenden möchte ich Ihnen die wichtigsten Datentypen vorstellen:

Ganzzahlen:

Datentyp	Wertebereich
byte	0 bis 255
short	-32,768 bis 32,767
int	-2,147,483,648 bis 2,147,483,647
long	9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807

Ganzzahlige Datentypen können wie der Name es schon vermuten lässt nur ganze Zahlen beinhalten. Nachkommastellen sind also nicht möglich. Es gibt 4 verschiedene Auswahlmöglichkeiten, wenn man einen Ganzzahl-Typen verwenden möchte. Das hat den Grund, dass

jeder Datentyp einen bestimmten Wertebereich hat, welcher nicht über- oder unterschritten werden kann. Die Auswahl der jeweiligen Datentypen sollte immer davon abhängen wie viel Platz man in einer Variable braucht, denn größere Datentypen mit einem höheren Wertebereich beanspruchen mehr Speicher als kleinere.

Zahlen mit Nachkommastellen:

Datentyp	Wertebereich
float	3.402823e38 bis 3.402823e38
double	-1.79769313486232e308 bis 1.79769313486232e308
decimal	79228162514264337593543950335 bis 79228162514264337593543950335

Für Zahlen mit Nachkommastellen gibt es nur 3 verschiedene Datentypen. Der „float“ kommt am häufigsten zum Einsatz und stellt ganz einfache Zahlen mit Nachkommastellen dar. Der Wertebereich dieser Zahlen ist enorm und sollte im Normalfall nicht überschritten werden. Der Grund dafür, dass es neben dem „float“ noch weitere Datentypen dieser Art gibt, ist nicht der Wertebereich, sondern die Genauigkeit bei der Berechnung der Nachkommastellen. Der „double“ ist genauer als der „float“ und wird vor allem dann eingesetzt, wenn es um die Arbeit mit Geldwerten geht. Hier können schon kleine

Rundungsfehler ein katastrophales Ausmaß haben und dementsprechend ist die Wahl des richtigen Datentyps sehr wichtig. Der „decimal“ wird aufgrund des hohen Ressourcenanspruchs nicht so oft verwendet. Er ist noch genauer als der „double“ und wird vor allem in der Wissenschaft eingesetzt.

Andere Datentypen:

Datentyp	Werte
bool	True und False
string	Text (z.B. „Hallo Welt“)
char	Zeichen/Buchstabe (z.B. „B“)

Zu guter Letzt kommen noch die „besonderen“ Datentypen. Sie sind besonders, weil jeder dieser Datentypen eine komplett eigene Art von Wert aufnehmen kann. Der „bool“ (Boolean) ist ein Wahrheitswert. Dieser Datentyp kann nur die Werte „True“ (Wahr) und „False“ (Falsch) beinhalten.

Der bereits kennengelernte „string“ kann eine Zeichenkette, also Text beinhalten. Er ist eine Aneinanderreihung von „chars“.

„char“ ist ein Datentyp welcher nur ein Zeichen/Buchstaben beinhalten kann.

Sie haben nun die wichtigsten Datentypen kennengelernt. Natürlich müssen Sie diese jetzt nicht auswendig lernen (vor allem nicht die genauen Wertebereiche der Zahlen-Typen), dennoch ist es wichtig sie mal alle gesehen zu haben.

Mathematische Operatoren

Werte müssen nicht nur gespeichert und gelesen, sondern auch verarbeitet werden. Die Mathematik spielt dabei natürlich eine große Rolle und deswegen werden Sie in diesem Kapitel lernen wie man im Code rechnet. Sie werden lernen wie man die 4 Grundrechenarten verwendet (Addition, Subtraktion, Multiplikation und Division) und wie man den Divisionsrest mit einem sehr einfachen Operator ermitteln kann.

Der mathematische Ausdruck

Bevor ich Ihnen zeige wie man die 4 Grundrechenarten im Code anwendet, möchte ich Ihnen erklären was ein mathematischer Ausdruck ist. Der Begriff „Ausdruck“ wird im Verlauf dieses Buches nämlich noch öfter vorkommen und dementsprechend empfinde ich es als wichtig, diesen Begriff hier schnell zu erklären.

Ein mathematischer Ausdruck ist eine Rechnung die ein Ergebnis liefert. Das Ergebnis selbst gehört nicht zum Ausdruck, sondern nur die formulierte Rechnung. Hier mal ein Beispiel:

10 + 2 = 12

Der Ausdruck ist in diesem Fall *„10 + 2“*. Das Ergebnis *„12“* ist das Resultat des Ausdrucks und gehört nicht zu diesem dazu. Wenn ich im weiteren Verlauf des Buches also von einem Ausdruck spreche, dann meine ich die Formulierung einer Rechnung.

Ein Ausdruck kann auch aus mehreren Ausdrücken zusammengesetzt werden:

((2 x 5) + (6 : 2)) = 13

Der Gesamt-Ausdruck lautet *„((2 x 5) + (6 : 2))“*. Dieser setzt sich zusammen aus den zwei Ausdrücken *„(2 x 5)“* und *„(6 : 2)“*.

Achtung: Ein Ausdruck ist eine Rechnung die ein Ergebnis liefert.

Die arithmetischen Operatoren (Grundrechenarten)

Jetzt wo sie wissen was ein mathematischer Ausdruck ist, können wir direkt mit dem Rechnen anfangen. Schauen wir uns also zuerst einmal die vier Grundrechenarten an!

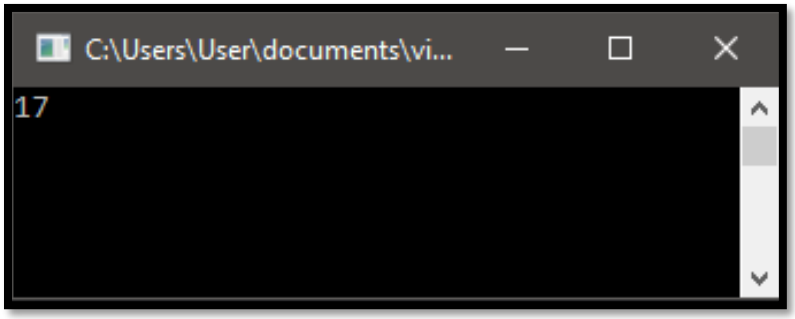
Die vier Grundrechenarten sind die einfachsten und grundlegendsten mathematischen Operationen die es gibt. Auf ihnen basiert alles und dementsprechend sind sie bei der Programmierung von sehr hoher Wichtigkeit. Um diese Operationen durchführen zu können braucht man die sogenannten „Operatoren“. Ein Operator ist ein Zeichen, dass für eine mathematische Operation steht. Ein Beispiel hierfür wäre das „+“-Zeichen für die Addition. In der folgenden Tabelle sehen Sie die wichtigsten Operatoren in C#:

Operator	Name
+	Addition
-	Subtraktion
*****	Multiplikation
/	Division

Diese Operatoren verwendet man, wenn man im Code mathematische Ausdrücke bilden möchte. Hier sind mal ein paar Beispiele:

```
static void Main(string[] args)
{
    int zahl1 = 10 + 5; //15
    int zahl2 = (10 * 3) / (3 * 5); //2
    int ergebnis = zahl1 + zahl2; // 15 + 2 = 17

    Console.WriteLine(ergebnis);
    Console.ReadKey();
}
```

Mit diesem Code erhält man folgende Ausgabe:



Wie Sie sehen, kann man auch Variablen in einen Ausdruck setzen. Die Variablen dienen dann als Platzhalter für den darin enthaltenen Wert.

Die vier Grundrechenarten im Code anzuwenden ist wie Sie sehen ziemlich einfach. Machen wir also weiter mit einem etwas spannenderen Operator.

Der Restwert-Operator (Modulo)

Der Modulo-Operator wird dann verwendet, wenn man den Rest einer Division herausfinden möchte. Der Rest ist immer das, was bei einer Division übrigbleibt. Hier mal ein kleines Beispiel (ich verwende ab jetzt die C#-Operatoren):

10 / 3 = 3 Rest 1

Die 3 passt 3 mal in die 10 hinein weil „3 * 3 = 9“. Die Rechnung geht also nicht ganz auf, denn von der 10 bleibt noch 1 über. Diese 1 ist der Divisionsrest. Hier noch ein Beispiel:

15 / 4 = 3 Rest 2

Die 4 passt 3 mal in die 15 hinein weil „3*4 = 12“. „15 – 12 = 2“. Es bleibt ein Divisionsrest von 2.

Diese Rechnung müssen wir beim Programmieren natürlich nicht so kompliziert ausschreiben. In C# (und so gut wie allen anderen Programmiersprachen auch) gibt es nämlich den Modulo-Operator, welcher den Rest einer Division zurückgibt.

Eine normale Division führen wir mit dem „/“-Zeichen durch. Um den Rest einer Division zu erfahren nutzen wir das „%“-Zeichen, welches den Modulo-Operator darstellt.

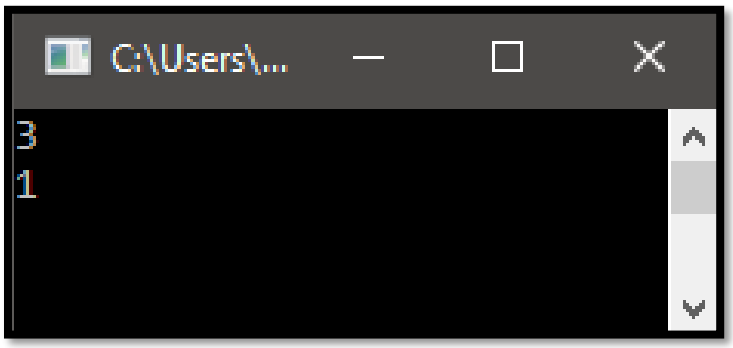
Im folgenden Code-Beispiel sehen Sie einmal eine einfache Division und eine Modulo-Operation. Die Rechnung ist dieselbe wie im ersten Rechenbeispiel:

```
static void Main(string[] args)
{
    int division = 10 / 3; //Einfache Division
    int rest = 10 % 3; //Rest herausfinden

    Console.WriteLine(division);
    Console.WriteLine(rest);
    Console.ReadKey();
}
```

Die Variable „division“ enthält den Quotienten aus 10 geteilt durch 3. Das Ergebnis ist der Wert 3.

Die Variable „rest“ enthält den Rest der exakt gleichen Division. Der Rest beträgt 1.

Die Ausgabe beider Variablen-Werte sieht folgendermaßen aus (oben Division, unten Restwert):



Die Ausgabe stimmt mit unseren per Hand gerechneten Ergebnissen überein.

Sie wissen nun also wie man mathematische Ausdrücke bildet, wie man die Grundrechenarten im Code anwendet und wie man den Rest einer Division ermittelt. Das sind alles sehr wichtige Grundlagen!

Methoden

Über das ganze Buch hinweg haben wir sie schon aufgerufen und jetzt wird es Zeit, dass Klarheit darüber geschaffen wird. Es geht in diesem Kapitel um Methoden. Diese stellen eine weitere unabdingbare Grundlage dar, da C# Programme aus Methoden zusammengebaut werden.

Eine Methode ist eine Reihe von zusammengehörenden Anweisungen, die nacheinander ausgeführt werden. Damit diese Anweisungen abgearbeitet werden, muss man nur die Methode aufrufen, indem man ihren Namen schreibt. Man kann also sagen, dass eine Methode ein kleines „Unterprogramm“ darstellt, welches beliebig oft ausgeführt werden kann. Methoden sorgen dafür, dass man bestimmte Code-Abschnitte wiederverwenden kann.

Methoden aufrufen

Wie man Methoden aufruft haben wir bereits gelernt. Man schreibt dazu einfach den Methoden-Namen gefolgt von einem Klammerpaar.

```
Console.ReadKey();
```

In dieses Klammerpaar kann man (wenn es in der Methode so definiert wurde) auch Parameter schreiben. Die **WriteLine()** Methode enthält beispielsweise einen Parameter der Textwerte annimmt, welche dann in der Konsole ausgegeben werden.

```
Console.WriteLine("Hallo Welt");
```

Methoden definieren

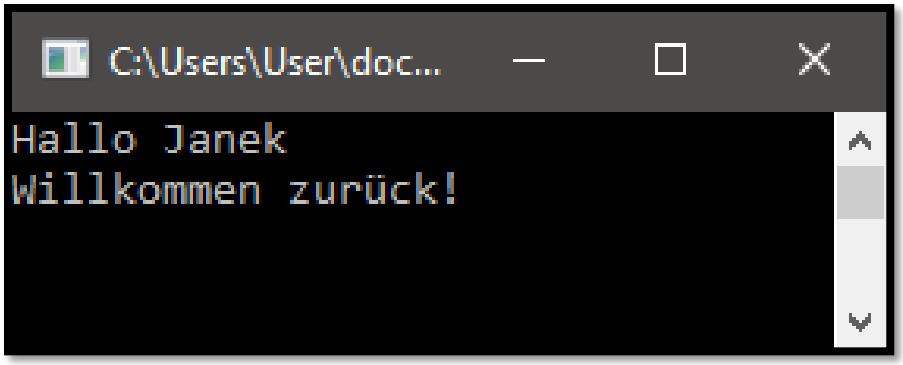
Bevor eine Methode aufgerufen werden kann, muss sie natürlich irgendwo definiert werden. Die Methodendefinition enthält alle Anweisungen, die beim Aufruf ausgeführt werden. Sehen wir uns zuerst einmal den Grundaufbau einer Methode an:

```
static void SageHallo(string name)
{
    Console.WriteLine("Hallo " + name);
    Console.WriteLine("Willkommen zurück!");
}
```

Was Sie auf dem Codeschnipsel sehen ist eine einfache Methodendefinition. Die Definition dieser Methode müssen Sie in eine Klasse schreiben (z.B. die Klasse „Program“). Diese Methode kann man aufrufen:

```
static void Main(string[] args)
{
    SageHallo("Janek");
    Console.ReadKey();
}
```

Wenn Sie das Programm starten, erhalten Sie folgende Ausgabe:



Anstelle von „Janek“ können Sie natürlich jeden beliebigen Namen eingeben. Die Methode ist wiederverwendbar und kann immer mit den benötigten Parametern ausgeführt werden. Schauen wir uns jetzt mal die Definition genauer an:

```
static void SageHallo(string name)
{
    Console.WriteLine("Hallo " + name);
    Console.WriteLine("Willkommen zurück!");
}
```

Zuerst gucken wir uns den Methodenkopf an. Der Methodenkopf enthält 3 bis 4 Elemente.

static void SageHallo(string name)

Fangen wir an mit dem Schlüsselwort **„static“**. Diese Methode ist statisch, was bedeutet, dass sie nicht an ein Objekt gebunden ist. Das müssen Sie zu diesem Zeitpunkt noch nicht verstehen, denn das lernen wir noch im Kapitel über Objektorientierung. Da ihre ersten Gehversuche mit Methoden in der Klasse „Program“ stattfinden werden, machen Sie diese Methoden mit dem „static“-Schlüsselwort statisch.

Als nächstes folgt das Schlüsselwort **„void“**. Void bedeutet, dass eine Methode keinen Rückgabewert hat. Ein Rückgabewert ist ein Ergebnis, dass eine Methode am Ende der Ausführung an den Aufrufer zurückgibt. Wenn Sie einen Rückgabewert für Ihre Methode haben wollen, dann schreiben Sie statt „void“, den zurückzugebenden Datentyp. Wollen Sie als Ergebnis eine Ganzzahl zurückgeben schreiben Sie statt „void“ also „int“. Dazu kommt aber noch ein anschauliches Beispiel!

Nach dem Rückgabetyp folgt der Methodenbezeichner. Der Bezeichner ist der Name der Methode (im Codebeispiel „SageHallo“). Man braucht diesen, um die Methode aufrufen zu können.

Auf den Bezeichner folgt das Klammerpaar, in welchen man die Parameter-Variablen deklariert. Diese Variablen können später im Methodenkörper verwendet werden. Die Initialisierung der Variablen erfolgt beim Aufrufen der Methoden, wenn ein Wert mitgegeben wird. Im Beispielcode deklarieren wir einen Parameter vom Typ „string“, mit dem Bezeichner „name“.

Zusammengefasst haben wir also eine statische (nicht objektgebundene) Methode, welche keinen Rückgabewert hat, über den Bezeichner „SageHallo“ aufrufbar ist und eine Parameter-Variable vom Typ „string“ besitzt, der man einen Namen zuweisen kann.

Kommen wir nun zum Methodenkörper. Der Körper einer Methode ist ein einfacher Codeblock, in welchen wir

Anweisungen hineinschreiben. Im Codebeispiel sieht dieser folgendermaßen aus:

```
{
    Console.WriteLine("Hallo " + name);
    Console.WriteLine("Willkommen zurück!");
}
```

Wir haben hier lediglich 2 Anweisungen hineingeschrieben. Einmal geben wir den Text „Hallo {name}“ und einmal „Willkommen zurück“ aus. Wie Sie sehen, kann man die Parameter-Variable „name“ ganz normal im Methodenkörper verwenden. In diesem Beispiel ist sie der Platzhalter für den auszugebenden Namen.

Sie haben nun also gelernt wie man eine einfache Methode ohne Rückgabewert definiert. Sehen wir uns nun an, wie man einen Wert an den Aufrufer zurückgeben kann!

Rückgabewerte von Methoden

Wie ich bereits erwähnt habe, können Methoden auch einen Rückgabewert haben. Der Rückgabewert ist ein Wert, der als Ergebnis einer Methode an den Aufrufer zurückgegeben wird. Um einer Methode einen Rückgabewert zu geben, müssen wir zunächst den Datentyp dafür festlegen. Das machen wir im Methodenkopf:

```
static int Addition(int zahl1, int zahl2)
{
    int ergebnis = zahl1 + zahl2;
    return ergebnis;
}
```

Konzentrieren wir uns zuerst nur auf den Kopf der Methode. Wie Sie sehen, haben wir da wo vorher „void“ stand den Datentyp „int“ angegeben. Das bedeutet, dass der Rückgabewert dieser Methode vom Typ „Integer“ ist.

Die Methode heißt „Addition“ und nimmt 2 Parameter an. Die Parameter heißen „zahl1“ und „zahl2“ und sind ebenfalls vom Typ „int“.

So viel also zum Methodenkopf. Kommen wir nun zum Methodenkörper. Der Codeblock enthält 2 Anweisungen. Die erste Anweisung ist die Initialisierung einer Variable namens „ergebnis“ vom Typ „int“. Ihr wird die Summe aus „zahl1“ und „zahl2“ zugewiesen. In der zweiten Zeile wird das Ergebnis als Rückgabewert zurückgegeben. Dazu schreibt man das Schlüsselwort **„return“** gefolgt vom zurückzugebenden Wert.

Gucken wir uns nun an, wie man den Rückgabewert abfangen und verarbeiten kann. Im folgenden Beispiel speichern wir den Rückgabewert in der Integer-Variable namens „summe“:

```
static void Main(string[] args)
{
    int summe = Addition(10, 5);
    Console.WriteLine(summe);
    Console.ReadKey();
}
```

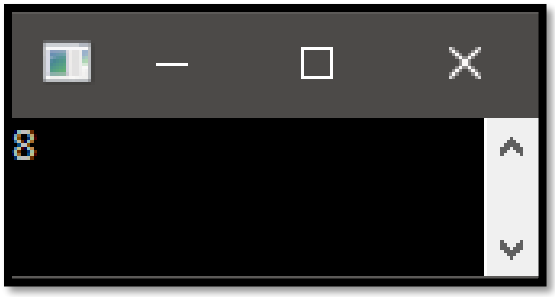
Der Methodenaufruf ist sowas wie ein Platzhalter für den Wert der zurückgegeben wird. Man kann den Aufruf also mit dem „=“-Operator zuweisen.

Nach der Zuweisung des Rückgabewerts in die Variable „summe“, lassen wir ihren Wert in der Konsole ausgeben. In der Konsole steht das Ergebnis „15“, da die Werte 10 und 5 als Parameter an die Methode „Addition“ übergeben wurden.

Um zu beweisen, dass der Methodenaufruf wie ein Platzhalter für den Rückgabewert benutzt werden kann, übergebe ich als Parameter für „WriteLine()“ einfach mal den Aufruf:

```
static void Main(string[] args)
{
    Console.WriteLine(Addition(5, 3));
    Console.ReadKey();
}
```

Das Ergebnis lautet 8, da als Parameter für „Addition“ die Werte 5 und 3 verwendet wurden:



Fallunterscheidungen

In diesem Kapitel lernen Sie, wie Sie Ihren Code mit den sogenannten Kontrollstrukturen steuern können. Sie werden die Fallunterscheidungen kennenlernen, welche die Ausführung von Codeblöcken zu bestimmten Bedingungen ermöglichen. In C# ist die bedingte Ausführung auf zwei verschiedene Arten realisierbar. Es gibt die If-Anweisungen und die Switch-Blöcke. Sie werden in diesem Kapitel beide Arten kennenlernen.

If-Anweisungen

Eine If-Anweisung ist ein Code-Block, welcher nur dann ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird. Der Aufbau einer If-Anweisung sieht folgendermaßen aus:

```
static void Main(string[] args)
{
    int alter = 10;

    if (alter >= 18)
    {
        Console.WriteLine("Du bist volljährig!");
    }

    Console.ReadKey();
}
```

In diesem Codebeispiel sehen Sie eine sehr einfache If-Anweisung. Wie sie Sehen wird der If-Block mit dem Schlüsselwort **„if“** (deutsch „wenn“) eingeleitet. Neben dem „if“ wird eine Bedingung gebildet. Die Bedingung im Beispiel lautet:

(alter >= 18)

Diese Bedingung muss erfüllt werden, damit der darauffolgende Codeblock ausgeführt wird.

Der Kopf dieses If-Statements lässt sich wie folgt übersetzen:

Wenn die Variable „alter“ größer oder gleich 18 ist, dann führe den Codeblock aus!

Sie sehen also, dass eine If-Anweisung aus einer Bedingung und einem dazugehörenden Codeblock besteht. In dem Codeblock werden die Anweisungen hineingeschrieben, die dann ausgeführt werden, wenn die Bedingung erfüllt wird.

Bedingungen bilden (Boolesche Ausdrücke)

Um If-Anweisungen benutzen zu können müssen wir erst lernen wie man die dazugehörenden Bedingungen bildet. Schauen wir uns das also mal an!

Bedingungen werden mit booleschen Ausdrücken gebildet. Ausdrücke haben wir im Kontext der einfachen Mathematik bereits kennengelernt. Ein Ausdruck war einfach nur eine Rechnung die ein Ergebnis liefert. Ein boolescher Ausdruck ist im Grunde dasselbe, nur dass er als Ergebnis lediglich einen booleschen Wert liefern kann. Ein boolescher Wert ist ein Wahrheitswert und dementsprechend kann ein solcher Ausdruck nur ein „true“ (wahr) oder „false“ (falsch) zurückgeben. Das ist auch ziemlich logisch, denn eine Bedingung kann entweder erfüllt werden (true) oder nicht erfüllt werden (false).

Auch boolesche Ausdrücke bildet man mit Operatoren und es gibt eine reihe von verschiedenen booleschen Operatoren, welche die Formulierung von Bedingungen ermöglichen. Diese werden in zwei verschiedene Arten unterteilt:

- **Vergleichsoperatoren**
- **Logische Operatoren**

Schauen wir uns diese doch mal jeweils im Detail an!

Vergleichsoperatoren

Mit Vergleichsoperatoren kann man zwei Werte miteinander vergleichen. Man kann prüfen ob zwei Werte gleich bzw. ungleich sind und ob der eine Wert größer oder kleiner als der

andere ist. In der folgenden Tabelle sehen Sie alle Vergleichsoperatoren:

Operator	Name	Beispiel
<	Kleiner als…	1 < 2 → true
>	Größer als…	5 > 3 → true
<=	Kleiner oder gleich…	5 >= 5 → true
>=	Größer oder gleich…	4 <= 5 → true
==	Gleich…	3 == 3 → true
!=	Ungleich…	2 != 4 → true

Die in der Tabelle gezeigten Beispiele ergeben alle das Ergebnis „true“. Die Beispiele stellen bereits richtig geformte Bedingungen dar und könnten so in einer If-Anweisung verwendet werden. Sehen wir uns nochmal das If-Beispiel an:

```
static void Main(string[] args)
{
    int alter = 10;

    if (alter >= 18)
    {
        Console.WriteLine("Du bist volljährig!");
    }

    Console.ReadKey();
}
```

Diese If-Anweisung hat als Bedingung den booleschen Ausdruck „alter >= 18“. Hier wurde der „Größer oder gleich“ Operator verwendet und dementsprechend wird dieser

Codeblock immer dann ausgeführt, wenn der Wert von „alter“ größer oder gleich „18“ ist. Wird die Bedingung nicht erfüllt, dann wird der Codeblock einfach übersprungen.

Logische Operatoren

Die logischen Operatoren werden dazu verwendet um mehrere Bedingungen zu verknüpfen. Somit kann man mehrere kleine Bedingungen zu einer einzigen komplexen zusammenfassen. Das braucht man zum Beispiel dann, wenn eine If-Anweisung nur ausgeführt werden soll, wenn mehrere Bedingungen gleichzeitig erfüllt werden.

In der folgenden Tabelle sehen Sie die logischen Operatoren aufgelistet:

Operator	Name	Funktionsweise
&&	Und-Operator	(true && true) → true
||	Oder-Operator	(true || false) → true
!	Nicht-Operator	(!true) → false

Diese Operatoren funktionieren folgendermaßen:

Der Und-Operator (&&)

Der Und-Operator verknüpft zwei Teilbedingungen zu einer Bedingung. Die große Bedingung die daraus resultiert wird nur dann erfüllt, wenn beide Teilbedingungen erfüllt werden. Man kann sich das ganz einfach merken indem man den Operator wörtlich nimmt:

„Bedingung1 **UND** Bedingung2 müssen erfüllt werden, damit der Ausdruck ein „true“ liefert“.

Hier ein Beispiel:

```
int alter = 10;
bool ausweisDabei = true;

if (alter >= 18 && ausweisDabei == true)
{
    Console.WriteLine("Du darfst das Spiel ab 18 kaufen!");
}

Console.ReadKey();
```

Im Beispiel wird der If-Block nur dann ausgeführt, wenn die Variable „alter“ größer oder gleich dem Wert „18“ ist **UND** wenn die Variable „ausweisDabei“ den Wert „true“ enthält. Man darf ein Spiel ab 18 also nur dann kaufen, wenn man volljährig ist und dies mit einem Ausweis beweisen kann.

Der Oder-Operator (||)

Der Oder-Operator verknüpft wie der Und-Operator auch zwei Teilbedingungen. Bei diesem ist es aber so, dass nur eine der Teilbedingungen erfüllt werden muss, damit der gesamte Ausdruck ein „true“ ergibt. Auch diesen Operator kann man in einfache Worte fassen:

„Wenn entweder Bedingung1 ***ODER*** *Bedingung2 erfüllt wird, dann ergibt der Ausdruck „true“. Die gesamte Bedingung wird auch dann erfüllt, wenn beide Teilbedingungen „true“ zurückgeben.“*

Hier ein Beispiel:

```
bool ausweisDabei = false;
bool führerscheinDabei = true;

if (ausweisDabei == true || führerscheinDabei == true)
{
    Console.WriteLine("Du darfst in den Club!");
}

Console.ReadKey();
```

In diesem Beispiel muss man sich vor einem Club-Eingang ausweisen. Man kann das entweder mit seinem Ausweis machen oder mit seinem Führerschein. Der Codeblock wird nur dann ausgeführt, wenn entweder „ausweisDabei“ oder „führerscheinDabei“ den Wert „true“ hat. Beides geht natürlich auch.

An dieser Stelle ein kleiner Tipp: Um eine Variable auf den Wert „true“ zu prüfen kann man auch die verkürzte Schreibweise nutzen. Das „== true“ ist bei einer prüfung auf „true“ nicht nötig, wie Sie auf folgendem Codeschnipsel sehen können:

```
if (ausweisDabei || führerscheinDabei)
{
    Console.WriteLine("Du darfst in den Club!");
}
```

Das einfache Schreiben des Variablenbezeichners reicht aus!

Der Nicht-Operator (!)

Der Nicht-Operator wird nicht zum Verknüpfen, sondern zum Negieren verwendet. Das bedeutet, dass man damit einen booleschen Wert „umdrehen“ kann. Somit kann man mit dem Nicht-Operator prüfen, ob eine Bedingung **nicht** erfüllt wird. Man setzt diesen Operator vor einen booleschen Wert oder Ausdruck, wie man im folgenden Beispiel sehen kann:

```
bool betrunken = false;

if (!betrunken)
{
    Console.WriteLine("Du darfst Auto fahren!");
}

Console.ReadKey();
```

Diese If-Anweisung wird nur dann ausgeführt, wenn man **NICHT** betrunken ist, also wenn die Variable „betrunken“ den Wert „false hat. Nur dann darf man Auto fahren.

In einer Bedingung **„!betrunken“** zu schreiben ist das exakt selbe wie **„betrunken == false“** abzufragen.

Else und Else-If

Oft kommt es vor, dass man einen alternativen Codeblock ausführen möchte, wenn die Bedingung der If-Anweisung nicht erfüllt wird. So kann man zum Beispiel bei der bereits in einem Beispiel gezeigten Altersabfrage eine alternative Ausgabe programmieren, wenn der Benutzer nicht volljährig ist. Genau dafür braucht man sogenannte „Else“-Blöcke. Else

bedeutet auf Deutsch „andernfalls“ und eine ganze If-Anweisung mit einem Else-Block lässt sich wörtlich folgendermaßen übersetzen:

*„**Wenn**(if) die Bedingung erfüllt wird, führe folgenden Codeblock aus. **Andernfalls**(else) führe diesen alternativen Codeblock aus!“*

Hier ist sehen Sie unsere erweiterte Altersabfrage:

```
int alter = 10;

if (alter >= 18)
{
    Console.WriteLine("Du bist volljährig!");
}
else
{
    Console.WriteLine("Du bist nicht volljährig!");
}
```

Der Else-Block wird mit dem Schlüsselwort **„else“** eingeleitet und muss direkt unter den If-Block geschrieben werden, zu dem er gehört.

Ein Else-Block kann auch an eine alternative Bedingung geknüpft sein. So könnte man die Altersabfrage auch dahingehend erweitern, dass man in Begleitung von einem Elternteil trotzdem ein Spiel ab 18 kaufen kann. Eine Alternative Bedingung wird mit einem „Else-If“ gebildet und sieht folgendermaßen aus:

```
int alter = 10;
bool elternDabei = true;

if (alter >= 18)
{
    Console.WriteLine("Du darfst das Spiel kaufen!");
}
else if (elternDabei == true)
{
    Console.WriteLine("Deine Eltern kaufen das Spiel!");
}
else
{
    Console.WriteLine("Du darfst das Spiel nicht kaufen!");
}
```

Zuerst wird im eigentlichen If-Block geprüft ob wir volljährig sind. Wird diese Bedingung nicht erfüllt, wird im Else-If-Block geprüft ob wir als alternative vielleicht unsere Eltern dabeihaben. Wird diese Bedingung erfüllt, kaufen uns einfach unsere Eltern das Spiel und die Sache ist erledigt. Wenn wir allerdings auch diese Bedingung nicht erfüllen können, dann wird der letzte Else-Block ausgeführt, der die Ausgabe „Du darfst das Spiel nicht kaufen!“ enthält.

Switch-Blöcke

Eine weitere Art der Fallunterscheidung stellen die sogenannten „Switch-Blöcke“ dar. Ein Switch-Block prüft eine Variable auf verschiedene Werte. Diese Werte sind die sogenannten „Cases“ (Fälle) unter welchen sich entsprechende Anweisungen befinden, die dann ausgeführt werden, wenn die zu prüfende Variable denselben Wert beinhaltet wie ein „Case“.

Hier ist ein einfaches Beispiel:

```
int monat = 3;

switch(monat)
{
    case 1:
        Console.WriteLine("Januar");
        break;

    case 2:
        Console.WriteLine("Februar");
        break;

    case 3:
        Console.WriteLine("März");
        break;
}
```

Auf diesem Codeschnipsel sehen Sie einen einfachen Switch-Block. Zuerst wird eine Integer-Variable namens „monat“ mit dem Wert „3“ initialisiert. Danach folgt der Switch-Block.

Der Block wird eingeleitet mit dem Schlüsselwort **„switch“** gefolgt von einem Klammerpaar, in welchen die zu überprüfende Variable geschrieben wird. Dann kommt der Codeblock und in diesem werden die verschiedenen Fälle, also die „cases“ definiert. Ein Case ist einfach nur ein konstanter (unveränderbarer) Wert, auf den die gegebene Variable geprüft wird. Er wird definiert mit dem Schlüsselwort **„case“** gefolgt von einem Wert. Nach dem Wert schreibt man einen

Doppelpunkt „:“ und unter dieser Definition stehen die zum Case gehörenden Anweisungen.

Die Anweisungen nach dem Doppelpunkt werden nur dann ausgeführt, wenn die zu prüfende Variable denselben Wert wie der Case hat. Im Beispiel hat die Variable „monat“ den Wert „3“. Dementsprechend wird der 3te Case ausgeführt, welcher in der Konsole den Text „März“ ausgibt. Um einen Case abzuschließen schreibt man das Schlüsselwort **„break;“**.

Die Funktionsweise von einem Switch-Block ist also ziemlich einfach. Man hat eine Variable, die auf verschiedene festgelegte Werte überprüft wird und wenn deren Wert mit einem der definierten Cases übereinstimmt, dann wird der entsprechende Code ausgeführt. Nach dem **„break;“** wird wieder aus dem Switch-Block rausgesprungen. Wenn keiner der Cases erfüllt wird, dann wird einfach wieder aus dem Switch-Block gesprungen, es sei denn man hat einen sogenannten **„Default“**-Case definiert. Der „Default“-Case wird immer als letzte Alternative ausgeführt. Man schreibt ihn mit dem Schlüsselwort **„default“**. Auf dem folgenden Code-Beispiel sieht man einen Switch-Block mit einem „Default“-Case.

```
switch(monat)
{
    case 1:
        Console.WriteLine("Januar");
        break;

    case 2:
        Console.WriteLine("Februar");
        break;

    case 3:
        Console.WriteLine("März");
        break;

    default:
        Console.WriteLine("Kein gültiger Monat!");
        break;
}
```

Sie haben nun also alle Arten der Fallunterscheidung in C# kennengelernt. Um das Ganze nochmal zusammenzufassen sei gesagt, dass man die „If-Anweisungen“ dann verwende, wenn man eine richtige Bedingung formulieren möchte.

Die Switch-Blöcke verwendet man dann, wenn eine Variable auf mehrere Werte geprüft werden soll.

Mit diesen zwei Werkzeugen werden sind Sie nun in der Lage, vollfunktionsfähige Anwendungen zu programmieren, die auf verschiedene Situationen entsprechend reagieren.

Arrays

In diesem Kapitel lernen Sie die sogenannten Arrays kennen. Arrays ermöglichen, wie Variablen auch, die Speicherung von Daten zur Laufzeit. Der Unterschied zwischen normalen Variablen und Arrays ist allerdings der, dass ein Array mehrere Werte gleichzeitig beinhalten kann.

Es gibt zwei verschiedene Arten von Arrays:

- **1 Dimensionale Arrays (1D)**
- **2 Dimensionale Arrays (2D)**

Die 1D-Arrays kann man sich wie eine Liste von Daten vorstellen, während ein 2D-Array wie eine Tabelle zu betrachten ist.

Hier ist ein anschauliches Beispiel für ein **1D-Array**

namen[]
„Horst“
„Günther“
„Peter“
„Jassens“

Diese Liste stellt einen 1D-Array dar. Er hat den Bezeichner „namen“ und wird dazu verwendet, Namen in Form von Strings abzuspeichern.

Ein 2D-Array sieht folgendermaßen aus:

Vorname	Nachname
„Peter“	„Müller“
„Fritz“	„Bauer“
„Anna“	„Gruber“

Dieser ist wie eine Tabelle mit Zeilen und Spalten darstellbar.

Arrays erstellen

Sie wissen nun also, dass ein Array eine Variable ist, in die man mehrere Werte gleichzeitig hineinspeichern kann. Schauen wir uns jetzt mal an wie man einen Array im Code erstellt.

Einen einfachen 1D-Array legt man folgendermaßen an:

```
static void Main(string[] args)
{
    string[] namen = new string[5];
}
```

Zuerst gibt man den gewünschten Datentyp für die Werte des Arrays an. Bei der Angabe des Datentyps müssen zwei eckige Klammern „[]“ angefügt werden, damit diese Variable als ein Array erkannt wird. Im Beispiel legen wir einen „string“-Array an.

Nach der Angabe des Datentyps muss der Bezeichner für den Array geschrieben werden. Im Beispiel heißt unser Array „namen“.

Haben wir einen Bezeichner gewählt, müssen wir das Array-Objekt erstellen. Eine Instanz von einem Objekt erstellt man mit dem Schlüsselwort **„new“**. Dieses Schlüsselwort werden wir im Kapitel über Objektorientierte Programmierung noch genauer ansehen, finden wir uns an dieser Stelle aber einfach mal damit ab, dass wir es beim Erzeugen eines Arrays brauchen. Nach dem „new“ müssen wir noch einmal den Datentyp des Arrays schreiben, gefolgt von einem eckigen Klammerpaar, welches die Größe unseres Arrays beinhaltet.

Das Objekt das wir mit „new string[5]“ erzeugt haben, weisen wir per Zuweisungsoperator „=“ unserer Array-Variable zu. Und schon ist unser Array erstellt. Wir können nun also direkt Werte hinzufügen.

Array-Werte lesen/überschreiben

Schauen wir uns nun an, wie man die Werte von einem Array lesen, bzw. bearbeiten kann.

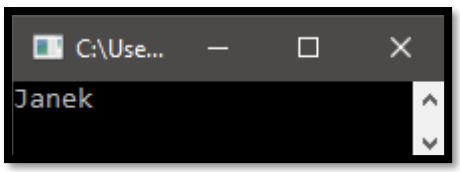
Jeder Wert eines Arrays liegt in einem sogenannten Index. Mit dem Index können Sie bestimmen, welchen Wert im Array Sie ansprechen möchten. Wenn Sie einen Index lesen bzw. überschreiben möchten, müssen Sie diesen zwischen den eckigen Klammern angeben, die man nach dem Bezeichner des Arrays schreibt. Im folgenden Beispiel weisen wir unserem „namen“-Array Werte hinzu:

```
static void Main(string[] args)
{
    string[] namen = new string[5]; //Array erstellen

    namen[0] = "Janek";
    namen[1] = "Peter";
    namen[2] = "Franz";
    namen[3] = "Dieter";
    namen[4] = "Rolf";
}
```

Wie Sie sehen beginnt der Array bei Index 0, welchen wir den Namen „Janek“ zuweisen. Das liegt daran, dass Arrays immer bei Index 0 beginnen. Das heißt, dass auch wenn unser Array 5 Indizes besitzt, der letzte Index die Nummer 4 ist.

Der schreibende und lesende Zugriff auf Array-Indizes funktioniert, mal abgesehen von der Index-Angabe, exakt gleich wie der von einer normalen Variable. Sie können einen der Werte auch in der Konsole ausgeben:

```
Console.WriteLine(namen[0]);
Console.ReadKey();
```



2D-Arrays

Sie haben nun gelernt, wie man einen 1D-Array erstellt und dessen Werte lesen bzw. überschreiben kann. Schauen wir uns nun an, wie man einen 2D-Array erstellt.

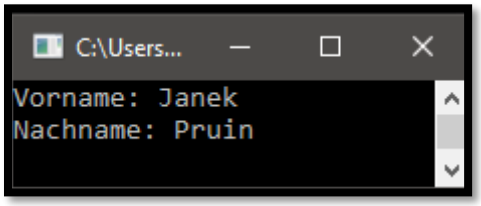
Die Erzeugung eines 2D-Arrays sieht sehr ähnlich aus wie die eines 1-Dimensionalen. Unserer Array-Variable müssen wir auch hier wieder ein eckiges Klammerpaar geben. Dieses Mal wollen wir aber sagen, dass der Array 2 Dimensionen besitzen soll, was wir mit einem einfachen Komma „,“ zwischen den eckigen Klammern bewerkstelligen. Bei der Erzeugung des Array-Objekts müssen wir dann nur noch die Größen für die jeweiligen Dimensionen angeben:

```
string[,] namen = new string[5,2]; //2D-Array erstellen
```

Im Beispiel erzeugen wir einen 2D-Array, bestehend aus 5 Zeilen und jeweils 2 Spalten. In die erste Spalte können wir einen Vornamen und in die zweite Spalte den Nachnamen schreiben. Der Zugriff auf einen Index innerhalb eines 2D-Arrays funktioniert wie beim 1D-Array, nur das wir hier 2 Werte angeben müssen. Die Zeilen-Nummer und die Spalten-Nummer.

Im folgenden Beispiel weisen wir zwei Zeilen, jeweils einen Vor- und einen Nachnamen zu. Außerdem lassen wir die erste Zeile des Arrays in der Konsole Ausgeben:

```
static void Main(string[] args)
{
    string[,] namen = new string[5,2]; //2D-Array erstellen

    //Zeile 1
    namen[0, 0] = "Janek";
    namen[0, 1] = "Pruin";

    //Zeile 2
    namen[1, 0] = "Peter";
    namen[1, 1] = "Müller";

    //Ausgabe
    Console.WriteLine("Vorname: " + namen[0, 0]);
    Console.WriteLine("Nachname: " + namen[0, 1]);
    Console.ReadKey();
}
```



Schleifen

Sehr oft kommt es vor, dass man bestimmte Teile des Programm-Codes mehr als nur einmal ausführen möchte. Stellen Sie sich mal vor, sie wollen einen Array mit 1000 Werten befüllen und dass Sie all diese Anweisungen per Hand schreiben müssen. Unser Code würde so um 1000 Zeilen wachsen, welche alle im Grunde dasselbe machen. Mit Schleifen kann man ein solches Szenario verhindern, da diese eine wiederholte Ausführung von Codeblöcken ermöglichen.

In C# gibt es 4 verschiedene Arten von Schleifen, welche wir uns in diesem Kapitel jeweils im Detail anschauen werden.

Zur Verfügung stehen uns folgende Schleifen-Arten:

- **While**
- **Do-While**
- **For**
- **Foreach**

Gehen wir diese der Reihe nach durch.

While-Schleifen

Die While-Schleife ist die einfachster Schleifen-Art. Sie ermöglicht es uns, einen Codeblock solange auszuführen, wie eine bestimmte Bedingung erfüllt wird. Die While-Schleife ist sozusagen eine If-Anweisung, die sich von selbst immer wieder wiederholt.

Der Aufbau einer While-Schleife sieht folgendermaßen aus:

```
static void Main(string[] args)
{
    int zahl = 0;

    while(zahl < 5)
    {

    }
}
```

Die While-Schleife wird eingeleitet mit dem Schlüsselwort **„while“**, gefolgt von einer Bedingung wie wir sie schon aus den

If-Anweisungen kennen. In den Codeblock können wir den Code schreiben, der immer wieder wiederholt werden soll, solange die Bedingung aus dem Schleifenkopf erfüllt wird.

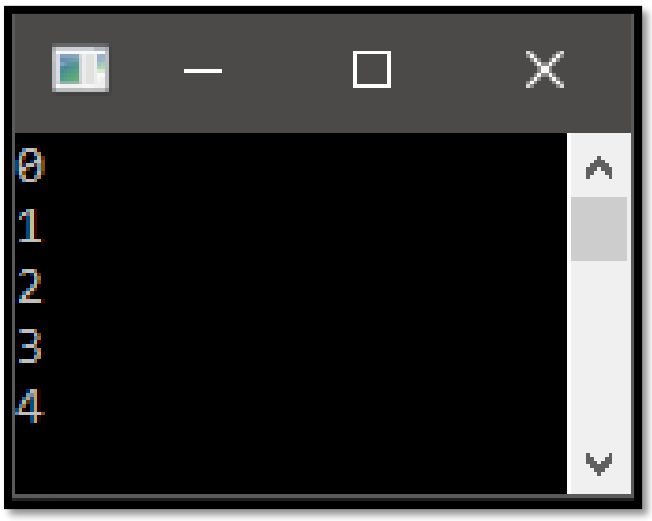
Im Code-Beispiel haben wir eine While-Schleife erstellt, deren Bedingung besagt, dass die Integer-Variable namens „zahl“ kleiner sein muss als 5. Jedes Mal, wenn alle Anweisungen innerhalb des Codeblocks ausgeführt worden sind, wird geprüft ob die Bedingung noch erfüllt wird. Ergibt diese Prüfung ein „true“, wird der Codeblock ein weiteres Mal durchlaufen.

Das Codebeispiel lässt sich folgendermaßen modifizieren:

```
static void Main(string[] args)
{
    int zahl = 0;

    while(zahl < 5)
    {
        Console.WriteLine(zahl);
        zahl += 1;
    }

    Console.ReadKey();
}
```

Diese Schleife wird 5-mal durchlaufen, da die Variable „zahl“ den Startwert 0 hat und die Bedingung besagt, dass „zahl“ kleiner sein muss als 5. Im Codeblock der Schleife wird der Wert der Variable „zahl“ per „WriteLine()“ ausgegeben und

dann wird dieser um 1 erhöht. Die Ausgabe des Beispielprogramms sieht wie folgt aus:



Sie wissen nun also, dass die While-Schleife sozusagen eine sich wiederholt ausführende If-Anweisung ist.

Do-While-Schleifen

Do-While-Schleifen sind auch ziemlich einfach zu verstehen, da sie im Grunde dasselbe wie die zuvor vorgestellten While-Schleifen sind. Der einzige Unterschied besteht darin, dass Do-While-Schleifen ihre Bedingungs-Überprüfung am Ende der Schleife durchführen. Aus dieser Besonderheit resultiert der Vorteil, dass eine Do-While-Schleife immer mindestens einmal ausgeführt wird. Der Grundaufbau der Do-While-Schleifen sieht folgendermaßen aus:

```
static void Main(string[] args)
{
    int zahl = 0;

    do
    {

    } while (zahl < 5);
}
```

Die Schleife wird eingeleitet mit dem Schlüsselwort **„do“** gefolgt vom Schleifenkörper in Form eines Codeblocks. Am Ende des Blocks folgt das Schlüsselwort **„while“** gefolgt von der Bedingung für die Wiederholung. Wichtig ist hier, dass am Ende der Bedingung ein Semikolon gesetzt wird.

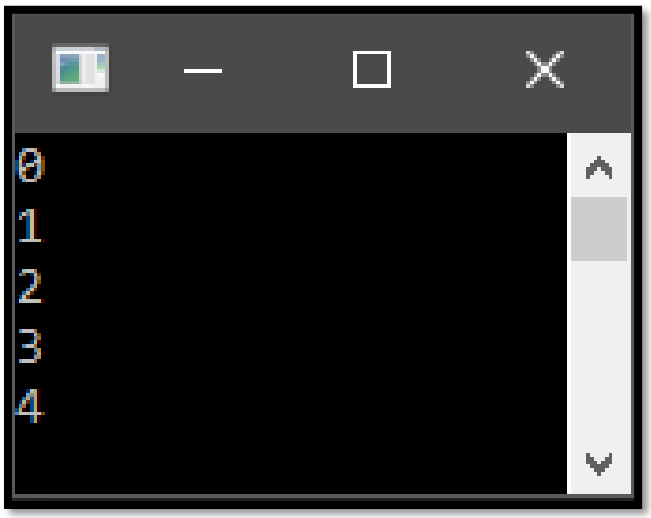
Die Besonderheit an der Do-While-Schleife ist wie bereits erwähnt die, dass sie immer mindestens einmal ausgeführt werden, egal ob die Bedingung erfüllt wird oder nicht. Das liegt daran, dass die Bedingung erst am Ende der Ausführung geprüft wird. Es ist klar, dass man die While-Schleife öfter verwendet als die Do-While-Schleife, dennoch sollte man sie wirklich nicht vergessen, da man diese Schleifenart doch sehr oft Situationsbedingt einsetzen kann.

Hier ist ein Beispiel für eine Do-While-Schleife:

```
static void Main(string[] args)
{
    int zahl = 0;

    do
    {
        Console.WriteLine(zahl);
        zahl += 1;
    } while (zahl < 5);

    Console.ReadKey();
}
```

Das Beispiel ist dasselbe wie das, dass wir schon bei der einfachen While-Schleife gesehen haben. Wir haben eine Zahl, die jeden Schleifendurchlauf um 1 erhöht und ausgegeben wird. Die Schleife läuft solange durch, bis die Zahl nichtmehr kleiner ist als 5.

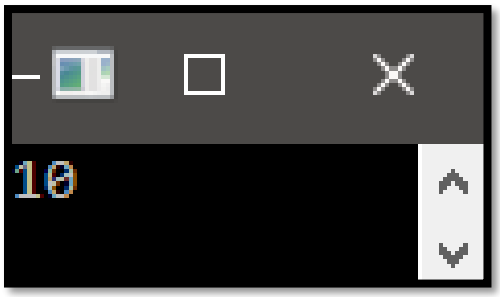
Die Ausgabe von diesem Programm ist dementsprechend die folgende:



Hier ist noch ein Beispiel in dem ganz gut demonstriert wird, dass die Schleife trotz einer nicht erfüllten Bedingung mindestens einmal ausgeführt wird:

```
int zahl = 10;

do
{
    Console.WriteLine(zahl);
    zahl += 1;
} while (zahl < 5);

Console.ReadKey();
```

Der Wert der Variable „zahl“ hat hier von Anfang an den Wert 10 und ist dementsprechend größer als 5. Die Bedingung der Do-While-Schleife wird also von Anfang an nicht erfüllt. Wenn wir das Programm ausführen sehen wir allerdings, dass die Zahl trotzdem ausgegeben wird:



For-Schleifen

Kommen wir nun zur komplexesten Schleifenart von allen. Der For-Schleife. Die For-Schleife ist die wohl am häufigsten benötigte Schleife von allen, da sie dank ihrer eingebauten Zählervariable sehr flexibel einsetzbar ist. Mit einer For-Schleife kann man Datenstrukturen wie z.B. Arrays durchlaufen, systematisch Mathematische Operationen durchführen und vieles mehr.

Auf dem ersten Blick sieht diese Schleifenart sehr kompliziert aus, wenn man den Aufbau aber einmal verstanden hat, dann merkt man schnell, dass der Schein trügt. Schauen wir uns mal eine einfache For-Schleife an:

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(i);
    }

    Console.ReadKey();
}
```

Eingeleitet wird die Schleife mit dem Schlüsselwort **„For“**, gefolgt von einem Klammerpaar welches aus 3 Elementen besteht. Nehmen wir das Klammerpaar einmal auseinander:

int i = 0; → **Initialisierung der Zählervariable**

i < 5; → **Bedingung zum weiteren Ausführen**

i++ → **Erhöhung der Zählervariable nach jedem Durchlauf**

Diese 3 Elemente innerhalb des Schleifenkopfes werden jeweils mit einem Semikolon voneinander getrennt (Das letzte Element (i++) braucht natürlich kein Semikolon zur Trennung).

Element 1. Initialisierung der Zählervariable

Die Zählervariable ist der Kern unserer For-Schleife. Sie enthält immer die Zahl des aktuellen Durchlaufs und zählt sozusagen, wie oft die Schleife sich schon wiederholt hat.

Man kann die Zählervariable auch im Schleifenkörper lesen um damit bestimmte Operationen durchzuführen. Somit ist es beispielsweise möglich, einen Array Index für Index zu durchlaufen.

Im Normalfall nennt man die Zählervariable einfach „i“. Hat man eine weitere verschachtelte For-Schleife, dann nennt man deren Zählervariable „j“.

Element 2. Bedingung zum weiteren Ausführen

Dieses Element ist ziemlich einfach zu verstehen. Es handelt sich dabei einfach um einen booleschen Ausdruck, welcher bestimmt ob ein weiterer durchlauf der Schleife stattfinden wird oder nicht. Wenn der Ausdruck „true“ ergibt, dann wird die Schleife noch einmal ausgeführt.

Element 3. Erhöhung der Zählervariable nach jedem Durchlauf

Am Ende folgt eine sehr einfache mathematische Operation. Und zwar die Erhöhung (manchmal auch Verringerung) der

Zählervariable. Diese Operation wird immer am Ende eines Durchlaufs durchgeführt.

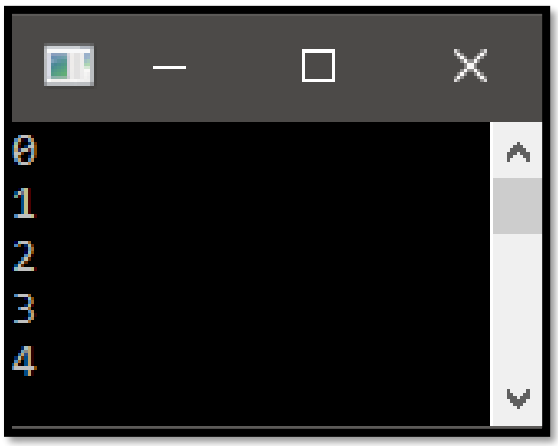
Häufig verwendet man für diese Erhöhung bzw. Verringerung des Zählervariablen-Wertes die vereinfachte Schreibweise von „+= 1“. Diese sieht folgendermaßen aus: „i++“. Mit einem einfachen „++“ am Ende des Variablennamens sagen wir dem Programm, dass wir den Wert einer Variable um 1 erhöhen wollen. Das Ganze geht natürlich auch beim Subtrahieren und wird wie folgt geschrieben: „i--“.

Jetzt wo wir den Aufbau einer For-Schleife kennengelernt haben, schauen wir uns mal ein praktisches Beispiel an!

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(i);
    }

    Console.ReadKey();
}
```

In diesem Beispiel haben wir eine For-Schleife, die 5-mal aufgeführt wird. Wir haben der Zählervariable „i“ einen Startwert von 0 gegeben und die Bedingung für einen weiteren Durchlauf ist immer die, dass i kleiner ist als 5. Im inneren der Schleife wird per „Console.WriteLine(i)“ immer der aktuelle Wert der Zählervariable ausgegeben und zum Schluss wird

diese per „i++“ um 1 erhöht. Die Ausgabe dieses Programms sieht also so aus:



Dieses Beispiel war ziemlich einfach. Da die For-Schleife aber sehr häufig in Verbindung mit Arrays eingesetzt wird, möchte ich auch noch ein Beispiel machen, in welchen ein Array Schritt für Schritt durchlaufen wird:

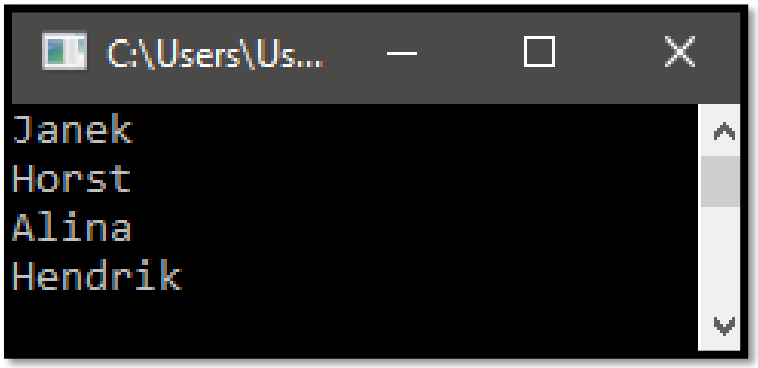
```
string[] namen = new string[4];
namen[0] = "Janek";
namen[1] = "Horst";
namen[2] = "Alina";
namen[3] = "Hendrik";

for(int i = 0; i < namen.Length; i++)
{
    Console.WriteLine(namen[i]);
}

Console.ReadKey();
```

Zuerst wird ein String-Array mit 4 Indizes erstellt und mit Namen befüllt.

Danach wird eine For-Loop erstellt welche bei 0 anfängt (weil Arrays bei Index 0 beginnen) und solange ausgeführt wird, wie „i“ einen kleineren Wert hat als „namen.Length“. „Length“ ist eine Eigenschaft von Arrays, welche immer die Größe des Arrays beinhaltet.

Im Schleifenkörper wird dann immer der aktuelle Index vom Array „namen“ per „Console.WriteLine()“ ausgegeben. „namen[i]“ enthält immer den aktuell durchlaufenen Index-Wert.

Die Ausgabe sieht folgendermaßen aus:



Foreach-Schleifen

Foreach-Schleifen sind die letzte Schleifen-Art von C#. Sie sind dazu da um Datenstrukturen wie Arrays, sehr einfach zu durchlaufen und deren Werte zu lesen. Wichtig ist dabei zu beachten, dass man mit einer Foreach-Schleife nur die Werte der einzelnen Indizes lesen, aber nicht überschreiben kann, da nur Kopien der Werte zum Lesen bereitgestellt werden.

Schauen wir uns mal ein einfaches Beispiel für eine Foreach-Schleife an:

```
string[] namenListe = new string[4];
namenListe[0] = "Janek";
namenListe[1] = "Horst";
namenListe[2] = "Alina";
namenListe[3] = "Hendrik";

foreach(string name in namenListe)
{
    Console.WriteLine(name);
}

Console.ReadKey();
```

Zuerst erstellen und befüllen wir einen String-Array mit 4 Indizes. Diese Indizes beinhalten jeweils einen Namen.

Danach kommt die Foreach-Schleife in Spiel und sorgt dafür, dass jeder einzelne Index durchlaufen und in der Konsole ausgegeben wird. Der Aufbau einer Foreach-Schleife ist dabei ziemlich einfach.

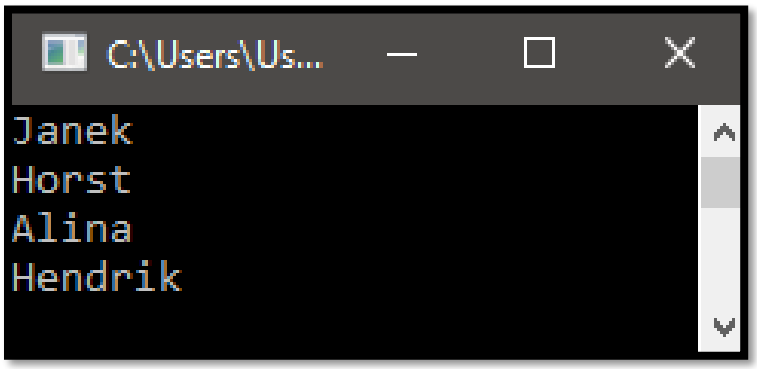
Die Foreach-Schleife besteht aus 2 Variablen. Einmal hat man eine Variable die immer den Wert des aktuell durchlaufenen Index beinhaltet und dann hat man noch die Variable die die Datenstruktur also den Array enthält. Werfen wir mal einen Blick auf den Aufbau des Schleifenkopfes:

```
foreach(string name in namenListe)
{
    Console.WriteLine(name);
}
```

Die Schleife wird eingeleitet mit dem Schlüsselwort **„foreach“**, gefolgt von einem Klammerpaar. Der Inhalt des Klammerpaars ist sehr einfach aufgebaut. Zuerst deklarieren wir eine Variable von dem Typ, den die einzelnen Wert der zu durchlaufenden Datenstruktur hat. In unserem Beispiel ist dies der Typ „string“. Danach geben geben wir mit dem Schlüsselwort **„in“** an, welche Datenstruktur wir durchlaufen wollen. Nach dem „in“ schreiben wir also einfach den Bezeichner des Arrays oder der Datenstruktur (im Beispiel „namensListe“).

Im Körper der Schleife können wir die einzelnen Indizes dann ganz einfach lesen. Im Beispiel geben wir immer den Wert des aktuell durchlaufenen Index in der Konsole aus.

Die Ausgabe sieht dann so aus:



Die Foreach-Schleife ist also ziemlich einfach. Um den Aufbau des Schleifenkopfes besser zu verstehen, kann man diesen

leserlich betrachten. Der folgende Schleifenkopf liest sich auf Deutsch wie folgt:

```
foreach(string name in namenListe)
```

*„Mit jedem **namen** innerhalb der **namensListe** mache folgendes…“*

Objektorientierte Programmierung

Jetzt wo Sie alle Grundlagen der Programmiersprache C# kennen, wagen wir uns an die Objektorientierte Programmierung. Dieses Kapitel dient Ihnen als eine kleine Einführung in das Programmierparadigma und die Grundidee von C#.

Was ist Objektorientierung?

Wie der Name es schon vermuten lässt dreht sich in der Objektorientierten Programmierung alles um Objekte. Ein Objekt kann man sich wie ein physisches „etwas“ vorstellen, dass irgendeine Aufgabe in unserem Programm übernimmt. So ist beispielsweise ein Array ein Objekt, dass eine Sammlung von Daten darstellt. Oder ein Auto in einem Videospiel ist ein Objekt, dass es dem Spieler ermöglich herumzufahren. Jedes C# Programm besteht aus Objekten, die alle für eine bestimmte Aufgabe zuständig sind. Man schreibt also nicht einfach nur einen Quellcode der von Oben nach Unten durchgeführt wird, sondern entwickelt verschiedene einzelne Objekte, die dann mit all den anderen Objekten des Programms zusammenarbeiten.

Eigene Objekte entwickelt man indem man sogenannte „Klassen“ programmiert. Eine Klasse ist ein Bauplan für ein Objekt, welche dessen Erstellung (die sogenannte Instanziierung) erst ermöglicht.

Bei der Objektorientierten Programmierung programmiert man also Klassen (Baupläne) welche dann die eigentlichen Objekt-Instanzen erstellen, die dann eine Funktionalität für das Programm zur Verfügung stellen.

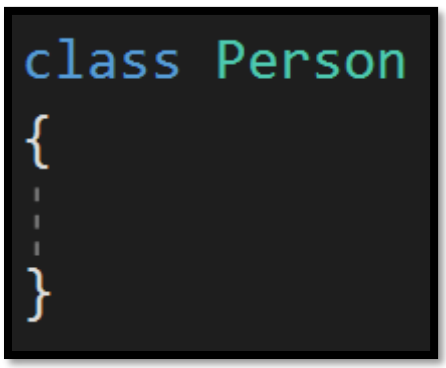
Auf Objekte und Klassen gehe ich im Verlauf dieses Kapitels aber natürlich noch genauer ein. Das sollte jetzt erstmal nur als eine sehr einfache Erklärung dienen.

Klassen

Wie wir bereits erfahren haben, sind Klassen Baupläne für Objekte. Das heißt, dass man in Klassen die Funktionen programmiert, die die später instanziierten Objekte dann haben sollen.

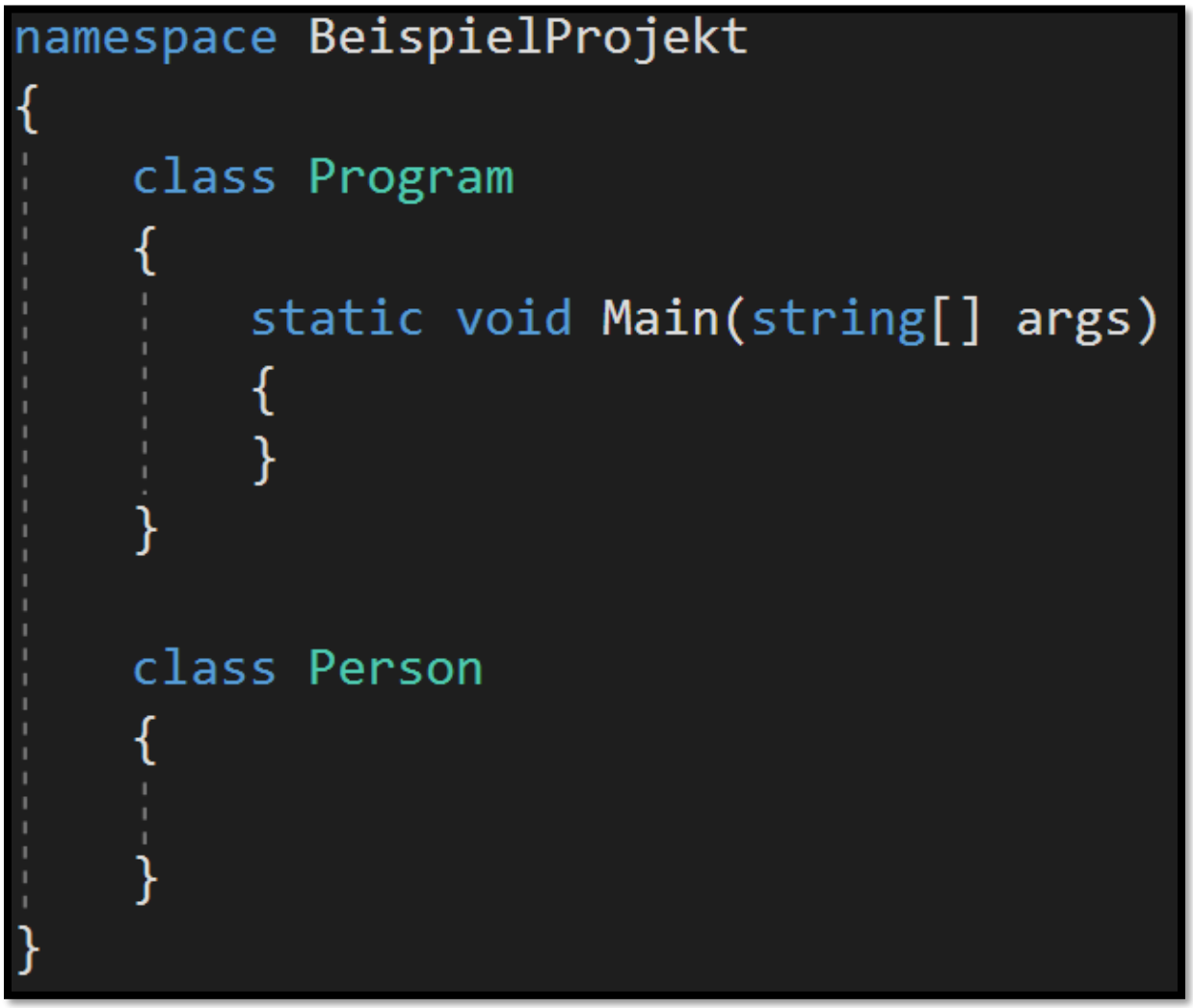
Klassen definieren

Eine Klasse definiert man mit dem Schlüsselwort **„class“** gefolgt vom gewünschten Klassenbezeichner. Hier ist mal eine leere Klasse als Beispiel:



Diese Klasse wird später eine Person in unserem Programm beschreiben!

Eine Klasse definiert man innerhalb eines Namespaces.



Auf dem Bild sehen Sie 2 Klassen. Einmal die schon vorgenerierte Klasse „Program“, welche auch die Main-Methode beinhaltet und die von uns gerade erstellte Klasse „Person“.

Klassen-Member

Eine Klasse besteht aus verschiedenen Elementen welche man Klassen-Member nennt. Ein Member kann eine Eigenschaft, Methode und vieles mehr sein! Auf die verschiedenen Arten

von Membern wird noch im Verlauf des Kapitels eingegangen. Ich möchte den Begriff an dieser Stelle dennoch kurz erklären, da er im weiteren Verlauf noch öfter fallen wird.

Ein Member ist ein Teil einer Klasse. Er gibt einer Klasse eine gewisse Funktionalität, an der sich von der Klasse abgeleitete Objekte bedienen können.

Eigenschaften von Klassen

Gucken wir uns nun die erste Art von Klassen-Membern an. Es geht um Eigenschaften. Eigenschaften sind so etwas wie Variablen. Sie können Werte beinhalten die man verändern und lesen kann. Eigenschaften werden dann verwendet, wenn man dem Objekt einen Wert geben möchte der es in gewissermaßen beschreibt. Eine Person könnte zum Beispiel die Eigenschaft „Alter“ haben, die das Alter der Person enthält. „Alter“ ist deshalb eine Eigenschaft, da das Lebensalter eines Menschen auch in der echten Welt eine sehr beschreibende Eigenschaft für ihn ist. Ein weiteres Beispiel wäre eine Eigenschaft namens „Name“. Jeder Mensch hat einen Namen und dementsprechend beschreibt dieser Wert einen Menschen auch.

In der Praxis sieht die Definition einer Eigenschaft wie folgt aus:

```
class Person
{
    //Eigenschaften
    public int Alter { get; set; }
    public string Name { get; set; }
}
```

Zuerst schreibt man den Zugriffsmodifizierer „Public“. Damit sagt man dem Programm, dass diese Eigenschaft auch außerhalb des Klassenblocks bekannt ist und gelesen bzw. überschrieben werden kann. Dann kommt die einfache Angabe des Datentyps gefolgt vom Eigenschafts-Bezeichner. Der Bezeichner von Eigenschaften sollte anders als der von normalen Variablen immer großgeschrieben werden.

Die Eigenschaft die Sie auf dem Beispiel sehen ist eine sogenannte „automatisch implementierte Eigenschaft“. Das erkennt man an dem **„{ get; set; }“**.

Das **Get** und **Set** sind die Methoden, mit welchen auf den Wert der Eigenschaft zugegriffen wird. Diese 2 Methoden sind das, was eine normale Variable von einer Eigenschaft unterscheidet. Sie kapseln den eigentlichen Wert der hinter der Eigenschaft steckt und sorgen dafür, dass auf diesen nur kontrolliert zugegriffen wird. Man kann die Get- und Set-Methoden auch selber definieren und bestimmte Abfragen mit einbauen, wie das geht zeige ich aber erst ein bisschen später!

Wie Sie sehen haben die Eigenschaften die wir definiert haben noch keinen Startwert. Das liegt daran, dass die Klasse nur ein

Bauplan für Objekte ist und jedes einzelne Objekt später seine eigenen Werte in diesen Eigenschaften haben wird. Denn jedes Objekt unterscheidet sich von den anderen, auch wenn Sie aus derselben Klasse instanziiert wurden. Eigenschaften sind also die Member, die jedes Objekt später einzigartig machen.

Objekte instanziieren

Bevor wir uns den Aufbau von Klassen weiter anschauen, möchte ich Ihnen zeigen wie man ein Objekt von einer Klasse instanziiert. Der Grund dafür ist der, dass wir andere Themen später besser nachvollziehen können, wenn wir Beispiele an bereits existierenden Objekten machen. Die Instanziierung von einem Objekt ist eigentlich ziemlich einfach. Nehmen wir als Beispiel unsere zuvor erstellte Klasse „Person“:

```
namespace BeispielProjekt
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }

    class Person
    {
        //Eigenschaften
        public int Alter { get; set; }
        public string Name { get; set; }
    }
}
```

Wir haben die Klasse „Person“ mit den Eigenschaften „Alter“ und „Name“. Von dieser Klasse möchten wir jetzt ein Objekt instanziieren. Das funktioniert im Grunde fast genauso wie die Erstellung einer normalen Variable:

```
static void Main(string[] args)
{
    Person horst = new Person();
}
```

In der Main-Methode erstellen wir eine Variable namens „horst“. Diese Variable ist vom Datentyp „Person“. Sie sehen also, dass jede Klasse einen eigenen Datentyp darstellt, von dem wir Variablen erstellen können. Diese Variable initialisieren wir auch. Wir weisen der Variable eine neue Objektinstanz der Klasse „Person“ zu, indem wir nach dem Zuweisungsoperator **„new“** schreiben, gefolgt vom Datentyp mit einem Klammerpaar. „new“ erstellt eine neue Objektinstanz und „Person()“ ruft den sogenannten Konstruktor auf, welchen wir uns später noch genauer anschauen werden.

In der Variable „horst“ steckt nun also ein Objekt vom Typ „Person“.

Auf Klassen-Member in Objekten zugreifen

Die Klasse „Person“ hatte auch Eigenschaften, auf welche wir direkt mal zugreifen wollen. Der Zugriff auf Eigenschaften funktioniert ähnlich wie der auf normale Variablen. Wir

müssen lediglich mit dem „Punkt-Operator“ angeben, welchen Member des Objekts in „horst“ wir ansprechen wollen:

```
static void Main(string[] args)
{
    Person horst = new Person();
    horst.Alter = 20;
    horst.Name = "Horst";
}
```

In diesem Beispiel weisen wir den Eigenschaften „Alter“ und „Name“ vom Objekt „horst“ Werte zu. Diese Eigenschaften können wir natürlich auch lesen und beispielsweise in der Konsole ausgeben:

```
static void Main(string[] args)
{
    Person horst = new Person();
    horst.Alter = 20;
    horst.Name = "Horst";

    Console.WriteLine(horst.Alter);
    Console.WriteLine(horst.Name);
    Console.ReadKey();
}
```

Eigenschaften lassen sich also lesen und überschreiben wie normale Variablen auch.

Ein genauerer Blick auf Eigenschaften

Gehen wir nun nochmal einen Schritt zurück zu den Eigenschaften von Klassen. Wir haben bereits gelernt, dass eine Eigenschaft ein objektbeschreibender Wertebehälter ist, welcher dafür sorgt, dass sich später instanziierte Objekte auch voneinander unterscheiden können. In diesem Unterkapitel möchte ich die Funktionsweise von Eigenschaften genauer erklären.

Gucken wir uns zuerst einmal den Grundaufbau einer Eigenschaft an. Eine Eigenschaft ist ein Konstrukt aus einer privaten Variable (einer Variable auf die man nicht von außerhalb des Objektes zugreifen kann) und 2 Methoden für den Zugriff auf diese Variable. Die 2 Methoden sind die **Get-** und **Set-Methoden**.

Die Get-Methode

Die Get-Methode wird immer dann aufgerufen, wenn wir den Wert einer Eigenschaft lesen wollen. In diesem Beispiel wird also im Hintergrund die Get-Methode aufgerufen:

```
Console.WriteLine(horst.Alter);
```

Die Get-Methode gibt als Rückgabewert ganz einfach den Wert der privaten Variable der Eigenschaft zurück.

Die Set-Methode

Die Set-Methode wird dann aufgerufen, wenn wir den Wert der Eigenschaft „setzen“, also überschreiben wollen. Im folgenden Beispiel wird die Set-Methode im Hintergrund ausgeführt:

```
horst.Alter = 20;
```

Schauen wir uns nun mal an wie man eine Eigenschaft ganz ausschreiben kann und die Get- und Set-Methoden selber schreibt:

```
class Person
{
    //Eigenschaften
    private int _alter;
    public int Alter
    {
        get
        {
            return _alter;
        }
        set
        {
            _alter = value;
        }
    }
}
```

Zuerst deklarieren wir innerhalb der Klasse „Person“ eine private Integer-Variable namens „_alter“. Diese Variable wird immer den Wert unserer Eigenschaft beinhalten und von der Eigenschaft gekapselt werden.

Danach erstellen wir die Eigenschaft. Wir schreiben zuerst wie bereits gelernt „public <Datentyp> <Bezeichner>“ also „public int Alter“ in unserem Beispiel und packen darunter einen Codeblock.

Innerhalb des Codeblocks können wir die Get- und Set-Methoden unserer Eigenschaft implementieren.

Dazu schreiben wir einfach die Schlüsselwörter **„get“** und **„set“** mit jeweils einem dazugehörenden Codeblock welcher den entsprechenden Methodenkörper darstellt.

Die Get-Methode ist wie bereits gelernt die Methode, die beim Lesen des Eigenschaftswertes ausgeführt wird. Deshalb müssen wir per **„return“** den Wert der privaten Variable „_alter“ zurückgeben.

Die Set-Methode wird zum Schreiben verwendet. Aus diesem Grund weisen wir der privaten Variable „_alter“ den Wert von **„value“** zu. „value“ ist ein Schlüsselwort welches immer den Wert darstellt, den wir bei einem schreibenden Zugriff auf die Eigenschaft angeben.

```
horst.Alter = 20;
```

In diesem Beispiel hat „value“ den Wert 20.

Sie haben nun also gelernt wie man eine Methode voll ausschreiben kann. Meistens reicht einem zwar die kürzere Schreibweise für automatisch implementierte Eigenschaften, die volle Schreibweise braucht man aber auch ab und an, wenn man vor einem Zugriff auf die Eigenschaft erst ein paar

Überprüfungen durchführen möchte. Hier sind die Schreibweisen nochmal im Vergleich:

Volle Schreibweise

```
class Person
{
    //Eigenschaften
    private int _alter;
    public int Alter
    {
        get
        {
            return _alter;
        }
        set
        {
            _alter = value;
        }
    }
}
```

Automatische Implementierung

```
class Person
{
    //Eigenschaften
    public int Alter { get; set; }
}
```

Achtung! Beide Beispiele sind von der Logik her das exakt gleiche!

Methoden in Klassen

Unsere Objekte sollen später natürlich nicht nur Werte beinhalten, sondern auch in der Lage sein irgendwelche Dinge zu tun. Und genau hier kommen Methoden ins Spiel. Wir haben Methoden in diesem Buch bereits kennengelernt, ich möchte dir in diesem Unterkapitel aber zeigen, wie man diese in Klassen implementieren kann.

Um eine Methode in einer Klasse zu definieren müssen wir diese nur in den Codeblock der Klasse schreiben. Wenn Sie nicht mehr wissen wie man eine Methode definiert, dann springen Sie am besten kurz zurück in das entsprechende Kapitel auf Seite 35.

In diesem Beispiel haben wir eine Methoden namens „SageHallo()“ innerhalb unserer Klasse „Person“ definiert:

```
class Person
{
    //Eigenschaften
    public int Alter { get; set; }

    //Methoden
    public void SageHallo()
    {
        Console.WriteLine("Hallo!");
    }
}
```

Diese Methode hat keinen Rückgabewert und gibt lediglich den Text „Hallo!“ in der Konsole aus.

Jedes Objekt vom Typ „Person“ kann nun also die Methode für sich aufrufen.

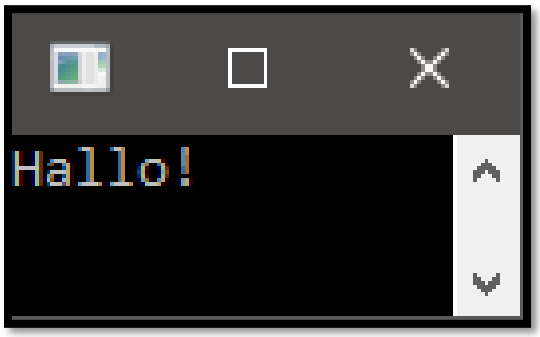
Schauen wir uns mal den Aufbau dieser Methode an, denn eine Sache ist hier anders als bei unseren vorherigen Methoden. Wir haben hier das Schlüsselwort **„pubic“**, damit unsere Methode auch von außerhalb der Klasse ausführbar ist und wir haben das Schlüsselwort **„static“** weggelassen. „static“ verwendet man nur dann, wenn man eine Methode schreibt die nicht an ein Objekt gebunden ist. Da wir von „Person“ aber Objekte ableiten die diese Methode dann auch ausführen sollen, müssen wir diese nicht-statisch machen indem wir „static“ einfach weglassen.

Schauen wir uns nun mal an wie man diese Methode aufruft.

Da eine Methode wie eine Eigenschaft auch ein Klassen-Member ist, können wir auf diese ganz einfach mit dem Punkt-Operator zugreifen um sie auszuführen:

```
static void Main(string[] args)
{
    Person horst = new Person();
    horst.SageHallo();

    Console.ReadKey();
}
```

Wenn wir das Programm ausführen erhalten wir folgende Ausgabe:



Ich möchte zum Schluss noch anmerken, dass eine Methode immer nur eine einzige spezielle Aufgabe haben sollte. Vermeiden Sie zu lange Methoden die viele Aufgaben gleichzeitig übernehmen und halten Sie sich stets kurz. Somit ist garantiert, dass wenn es ein Problem gibt, Sie immer genau wissen wo Sie danach suchen müssen.

Methoden überladen

Die Methodenüberladung ermöglicht es uns, mehrere Versionen von einer Methode zu erstellen, die jeweils unterschiedliche Parameterlisten besitzen. Dazu kann man eine Methode mit demselben Namen mehrmals in einer Klasse definieren und jeder „Version“, also jeder Überladung dieser Methode, eine andere Parameterliste geben.

Hier ist ein Beispiel dafür:

```
class Ausgeber
{
    //Methoden
    public void Ausgabe(string text)
    {
        Console.WriteLine(text);
    }

    public void Ausgabe(int zahl)
    {
        Console.WriteLine(zahl);
    }
}
```

In diesem Beispiel haben wir eine Methode namens „Ausgabe()“ welche 2 verschiedene Überladungen besitzt. Einmal nimmt sie einen Parameter vom Typ „String“ an und gibt diesen dann in der Konsole aus und einmal macht sie dasselbe mit einem Parameter vom Typ „Integer“.

Wir können diese Methode „Ausgabe()“ nun mit beiden Datentypen jeweils aufrufen:

```
static void Main(string[] args)
{
    Ausgeber ausgeber = new Ausgeber();

    ausgeber.Ausgabe("Hallo Welt!");
    ausgeber.Ausgabe(2000);
}
```

Der Konstruktor

Der Konstruktor ist eine Methode die wir im Verlauf dieses Buches schon öfter aufgerufen haben. Erinnern Sie sich zurück an die Instanziierung von Objekten mit dem **„new“**-Schlüsselwort.

```
Person horst = new Person();
```

Wie Sie sehen sieht das „Person();“ nach dem „new“ wie ein Methodenaufruf aus. Das liegt daran, dass es sich dabei wirklich um einen Methodenaufruf handelt. Es wird dabei nämlich der sogenannte Konstruktor aufgerufen!

Der Konstruktor ist eine ganz spezielle Methode die in **jeder** einzelnen Klasse definiert ist, auch wenn Sie diese Definition selber gar nicht vorgenommen haben. Er ist die aller erste Methode die in einem Objekt aufgerufen wird, wenn wir dieses instanziieren.

In einem Konstruktor kann man bestimmte Variablen bzw. Eigenschaften initialisieren und das Objekt in welchen er aufgerufen wird somit auf sein „Leben“ vorbereiten.

Definieren wir keinen eigenen Konstruktor, besitzt unsere Klasse automatisch einen leeren und unsichtbaren Standard-Konstruktor.

Schauen wir uns nun einmal an wie man einen eigenen Konstruktor definieren kann. Den Aufbau eines Konstruktors sehen Sie im folgenden Beispiel:

```
class Person
{
    //Eigenschaften
    public string Name { get; set; }

    //Konstruktor
    public Person(string _name)
    {
        Name = _name;
    }
}
```

In diesem Beispiel haben wir eine Klasse namens „Person“, welche die Eigenschaft „Name“ besitzt.

Unter der Eigenschaft befindet sich unser eigener Konstruktor, mit welchen wir der Eigenschaft „Name“ schon direkt beim

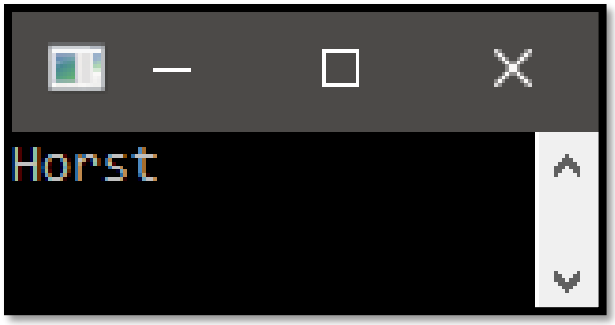
Erstellen eines „Person“-Objekts einen Namen zuweisen können. Betrachten wir diese Konstruktor-Definition mal genauer.

Der Konstruktor muss „public“ sein, damit man ihn beim Instanziieren eines neuen Objekts auch erreichen kann. Der Bezeichner des Konstruktors muss derselbe wie der Klassenname sein. Dementsprechend heißt unser Konstruktor „Person“. Neben dem Bezeichner muss man nun wie bei einer normalen Methode auch eine Parameterliste Schreiben. Mit den Werten dieser Parameterliste befüllt man im Normalfall dann die Eigenschaften innerhalb des Konstruktor-Körpers.

Haben wir einen Konstruktor wie auf dem gezeigten Beispiel definiert, müssen wir den Parameter beim Aufruf natürlich auch mitgeben. Das sieht dann folgendermaßen aus:

```
static void Main(string[] args)
{
    Person horst = new Person("Horst");
    Console.WriteLine(horst.Name);

    Console.ReadKey();
}
```

In diesem Beispiel geben wir dem Konstruktor als Parameter den Namen „Horst“ mit. Da im Konstruktor die Eigenschaft „Name“ initialisiert wird, können wir ihren Wert auch gleich im Anschluss in der Konsole ausgeben lassen. Das Resultat ist diese Konsolenausgabe:



Konstruktoren überladen

Wie normale Methoden auch kann man den Konstruktor überladen. Dementsprechend ist es möglich mehrere Versionen für seinen Konstruktor zu erstellen. Ein Beispiel für einen überladenen Konstruktor sehen Sie hier:

```
class Person
{
    //Eigenschaften
    public string Name { get; set; }
    public int Alter { get; set; }

    //Konstruktor
    public Person(string _name)
    {
        Name = _name;
    }

    public Person(string _name, int _alter)
    {
        Name = _name;
        Alter = _alter;
    }
}
```

In diesem Beispiel haben wir 2 Überladungen von unserem Konstruktor erstellt.

Der erste Konstruktor nimmt nur einen Parameter vom Typ „String“ an und weist diesen dann der Eigenschaft „Name“ zu.

Der zweite Konstruktor nimmt einen „String“ und einen „Integer“ an. Mit dem „String“ wird die Eigenschaft „Name“ initialisiert und der „Integer“ wird „Alter“ zugewiesen.

Vererbung

Das wohl wichtigste Konzept der Objektorientierten Programmierung ist die sogenannte Vererbung. Klassen können von anderen Klassen erben, um so die eigene Funktionalität zu erweitern.

Beim Vererben gibt eine Klasse ihre Member an eine erbende Klasse weiter. Somit können wir mithilfe der Vererbung Klassen erstellen, die auf der Funktionalität von einer anderen Klasse aufbauen.

Um das ganze richtig nachvollziehen zu können, müssen wir uns mal ein einfaches Beispiel ansehen. Nehmen wir mal an wir haben eine Klasse namens „Fahrzeug“, die die Grundfunktionalität für ein x-beliebiges Fahrzeug bereitstellt. In dieser Klasse wird nicht spezifiziert ob es sich dabei um einen Traktor, einen Kleinwagen, ein Motorrad usw. handelt. Es werden lediglich die Grundfunktionen bereitgestellt die wirklich jedes Fahrzeug braucht. Zum Beispiel eine Methode Fahren „Fahren()“ und eine weitere Methode zum Bremsen „Bremsen()“. Die Eigenschaften für diese „Fahrzeug“-Klasse sind „AnzahlRäder“ und „AnzahlPassagiere“, da jedes Fahrzeug diese Eigenschaften hat. Ein Fahrzeug braucht Räder

um fahren zu können und jedes Fahrzeug hat eine maximale Anzahl an Passagieren, die gleichzeitig damit reisen können.

Wir haben mit „Fahrzeug“ also eine Klasse, die wirklich nur die Grundfunktionen für alle möglichen Fahrzeuge zur Verfügung stellt. Auf dieser Klasse können anderen Klassen wie beispielsweise eine Klasse namens „Auto“ aufbauen indem sie von „Fahrzeug“ erben. Somit müssten wir in der spezifischeren „Auto“-Klasse nicht mehr die Grundfunktionen für ein Fahrzeug programmieren, da diese dank der Vererbung von „Fahrzeug“ schon vorhanden sind.

Die Klasse „Auto“ könnte jetzt also einfach noch spezifischere Member beinhalten die ein Auto ausmachen, wie beispielsweise eine Eigenschaft namens „Motor“, die besagt welche Art von Motor sich im Auto befindet. Eine weitere Eigenschaft von Autos wäre „AnzahlTüren“, da jedes Auto eine bestimmte Anzahl an Türen besitzt. Außerdem können wir darin noch 2 Methoden implementieren namens „MotorAn()“ und „MotorAus()“, damit man den Motor auch ein- und ausschalten kann.

Schauen wir uns diese beiden Klassen doch einfach mal im Code an, damit wir uns ein besseres Bild davonmachen können. Beachten Sie bitte, dass die Methoden einfach nur aus kleinen Konsolenausgaben bestehen, damit wir ein logisches und anschaubares Beispiel haben:

Die Klasse „Fahrzeug“

```
class Fahrzeug
{
    //Eigenschaften
    public int AnzahlRäder { get; set; }
    public int AnzahlPassagiere { get; set; }

    //Methoden
    public void Fahren()
    {
        Console.WriteLine("Fahre...");
    }
    public void Bremsen()
    {
        Console.WriteLine("Bremse...");
    }
}
```

Diese Klasse stellt alle Grundfunktionen für ein Fahrzeug zur Verfügung. Sie wird diese Grundfunktionen später an andere Klassen wir zum Beispiel „Auto“ weitergeben, indem diese von „Fahrzeug“ erben.

Die Klasse „Auto“

```
class Auto : Fahrzeug
{
    //Eigenschaften
    public string Motor { get; set; }
    public int AnzahlTüren { get; set; }

    //Methoden
    public void MotorAn()
    {
        Console.WriteLine("Motor wurde eingeschaltet");
    }

    public void MotorAus()
    {
        Console.WriteLine("Motor wurde abgeschaltet");
    }
}
```

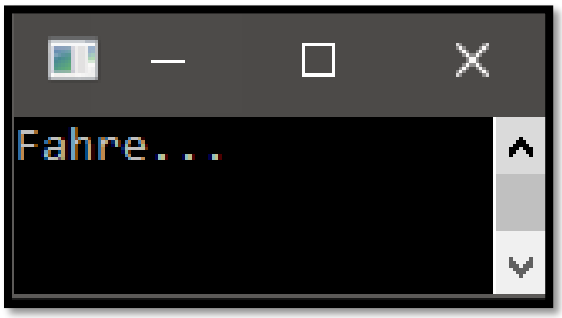
Diese Klasse namens „Auto“ stellt eine ganz bestimmte Art von Fahrzeug dar. Dementsprechend besitzt sie ihre eigenen Member, die die Funktionen für diese Art von Fahrzeug bereitstellen.

Wie Sie sehen können haben wir ganz oben am Kopf der Klasse einen Doppelpunkt gesetzt, gefolgt vom Klassennamen „Fahrzeug“. Damit sorgen wir dafür, dass „Auto“ von „Fahrzeug“ erbt und somit alle Member dieser Klasse übernimmt. Wir müssen also nichts Weiteres mehr schreiben und schon hat „Auto“ alle Member die „Fahrzeug“ auch hat.

Wenn wir nun also ein Objekt aus der Klasse „Auto“ instanziieren, dann kann dieses Objekt beispielsweise die Methode „Fahren()“ aufrufen und das obwohl diese Methode in einer ganz anderen Klasse definiert wurde.

```
static void Main(string[] args)
{
    Auto meinAuto = new Auto();
    meinAuto.Fahren();

    Console.ReadKey();
}
```

Die Ausgabe für dieses Programm würde folgendermaßen aussehen:



Die Member die wir in der „Auto“-Klasse selbst definiert haben sind natürlich auch verfügbar.

Sie haben nun also gelernt, dass man mithilfe der Vererbungsmechanik eine Klasse um die Funktionalität von einer anderen Klasse erweitern kann.

Eine Klasse von der eine andere Klasse erbt nennt man eine **Basisklasse**.

In C# kann eine Klasse immer nur von einer einzigen anderen Klasse erben. Es ist jedoch möglich komplexe Vererbungshierarchien aufzubauen, da eine Klasse die von einer anderen Klasse erbt auch an eine weitere Klasse vererben kann.

Somit könnte die Klasse „Auto“ von „Fahrzeug“ erben, während diese von der Klasse „Maschine“ erbt.

Also: „Maschine“ → „Fahrzeug“ → „Auto“

„Auto“ hätte in diesem Fall auch zugriff auf all die Member von „Maschine“.

Die Vererbung ist wie bereits gesagt das wichtigste Konzept der Objektorientierten Programmierung. Man sollte diese Funktion wirklich nutzen, da die Programme somit viel übersichtlicher, logischer und wiederverwendbarer werden.

Ein kleiner Tipp noch: Programmiere Klassen nicht zu groß. Sorge dafür, dass jede Klasse eine gewisse Aufgabe hat und nicht zu vollgeladen wird. Eine Klasse sollte wie gesagt nur eine bestimmte Aufgabe haben und alles was nicht zu dieser Aufgabe gehört sollte auf andere Klassen verteilt werden.

Sie haben es geschafft!

Herzlichen Glückwunsch. Wenn Sie es bis hier geschafft haben, haben Sie alle nötigen Grundlagen erworben um direkt mit dem Programmieren anzufangen. Natürlich gibt es noch immer viel zu lernen aber wie sagt man so schön: Als Programmierer lernt man nie aus! Machen Sie das Beste aus dem erworbenen Wissen. Wagen Sie sich an ihre ersten Praktischen Projekte und haben Sie vor allem Spaß dabei!

Der nächste logische Schritt wäre es sich unseren „C# Masterkurs“ auf www.programmieren-starten.de zu sichern. In diesem Videokurs lernen sie nochmal alle Themen in Videoform kennen und haben vor allem auch Zugriff auf die fortgeschrittenen Themen. Darin lernen Sie erweiterte Konzepte der Objektorientierung, nützliche Klassen des .NET-Frameworks und viele weitere unglaublich wichtige C#-Features kennen. Der Kurs umfasst über 15 Stunden Videomaterial und ist ein Muss für jeden, der die Programmiersprache C# schnell und effektiv erlernen möchte! Für Sie sind darin vor allem die fortgeschrittenen Themen von großem Nutzen!

Impressum

Erscheinungsjahr 2018

Hendrik Pruin
Sternstraße 15
85080 Gaimersheim

Email: hendrikpruin@programmieren-starten.de
Webseite: www.programmieren-starten.de

Datenschutz:
www.programmierenstarten.de/datenschutzerklaerung/