Inhaltsverzeichnis

Übersicht und Versionen 1
Erklärungen zu den Symbolen 1
Einführung 1
Kurzübersicht 1
Ihr Nutzen 2
Zielpublikum 2
Voraussetzung 2
C# und .NET 2
Worum geht es? 2
Was lernen Sie in diesem Kapitel? 2
Was ist .NET? 2
Ein neue Welt für .NET 5
Ein Beispiel eines CIL Codes 6
Die APIs des .NET Standard 2.0 7
Erste Schritte im C# 7
Worum geht es? 7
Was lernen Sie in diesem Kapitel? 7
Hello World 7
Zusammenfassung 10
Kontrollfragen 10
Klassen und Objekte 10
Worum geht es? 10
Was lernen Sie in diesem Kapitel? 11
Einführung 11
Begriffe 11
Deklaration von Klassen 11
Erzeugen von Instanzen einer Klasse 11
Felder einer Klasse 13
Modifikatoren 14
Variable und Felder 15
this 15
Deklaration von Konstanten 17
Methoden einer Klasse 17
Parameterübergabe 18
Optionale Parameter 20
Überladen von Methoden 21
Statische Methoden / Variablen 22
Zugriff auf statische Methoden und Variablen 23
Statische Klasse 24
Initialisierung 24
Konstruktoren 24
Destruktoren / Finalizer 26
Namensräume 27
Verwenden von Namensräumen 27
Der globale Namensraum 28
Zusammenfassung 28
Kontrollfragen 28
Übung Klasse und Objekte 28

Grundlagen Datentypen 29
Worum geht es? 29
Was lernen Sie in diesem Kapitel 29
Datentypen 29
Speicherverwaltung 29
Die Null-Referenz 30
Nullbare Typen 30
Garbage Collection 30
Standard-Datentypen von C# 31
Methoden von Datentypen 31
Type und typeof() 32
Typkonvertierung 33
Das as-Operator 34
Der is-Operator 35
Umwandlungsmethoden 37
Boxing und Unboxing 37
Strings 39
Stringzuweisungen 39
Zugriff auf String 40
Formatierung von Daten 41
Standardformate 41
Selbstdefinierte Formate 41
Ausrichtung 41
Zusammenfassung 41
Übungen Datenverwaltung 41
Ablaufsteuerung 41
Worum geht es? 41
Was lernen Sie über dieses Kapitel? 41
Absolute Sprünge 41
Bedingungen und Verzweigungen 41
Vergleichs- und logische Operatoren 41
Die bedingte Zuwweisung 41
Die for-Schleife 41
Die while-Schleife 41
Die do-while-Schleife 41
Zusammenfassung 42
Kontrollfragen 42
Übungen Programmablauf 42
Operatoren 42
Worum geht es? 42
Was lernen Sie in desem Kapitel 42
Mathematische Operatoren 42
Grundrechnenarten 42
Zusammengesetzte Rechenoperatoren 42
Die Klasse Math 42
Zusammenfassung 42
Kontrollfragen 42
Erweiterte Datentypen 42
Worum geht es? 42

Was lernen Sie in diesem Kapitel? 43
Array 43
Eindimensionale Arrays 43
Mehrdimensionale Arrays 43
Ungleichförmige Arrays 43
Arrays initialsieren 43
Die foreach-Schleife 43
Struct 43
Aufzählungen 43
Standard-Aufzählungen 43
Flag Enums 43
Zusammenfassung 43
Kontrollfragen 43
Übungen Array 43
Vererbung und Interfaces 43
Worum geht es? 43
Was lernen Sie in diesem Kapitel 44
Vererbung von Klassen 44
Zugriff auf Elemente der Basisklasse 44
Überschreiben von Methonden 44
Aufruf des Konstruktors der Basisklasse 44
Abstrakte Klassen 44
Versiegelte Klassen 44
Verbergen von Methoden 44
Interface 44
Explizite Interfaces 44
Zusammenfassung 44
Kontrollfragen 44
Übungen 44
Eigenschaften und Indexer 44
Worum geht es? 44
Was lernen Sie in diesem Kapitel? 45
Eigenschaften (Properties) 45
Erweiterungen der Properties 45
Indexer 45
Zusammenfassung 45
Kontrollfragen 45
Übungen 45
Strukturierte Fehlerbehandlung 45
Worum geht es? 45
Was lernen Sie in diesem Kapitel? 45
Was sind Exceptions? 45
Exception abfangen 45
Exception auslösen 45
Anwendungstipps 45
Zusammenfassen 45
Kontrollfragen 46
Anhang 46
Erweiterung C# 46
Initialisierer für Auto-Properties, read-only Auto-Properties 46

Verwendung statischer Klassen 46
Exception Filter 46
Null-conditional-Operator 46
Expression bodied Member 46
Initialisierung von Collections 46
String Interpolation 46
nameof Operator 46
Literatur 46
Fussnoten 46

Übersicht und Versionen

- Visual Studio 2017
- .NET 4.62
- C# 7.0

Erklärungen zu den Symbolen

	Lernziele
	An dieser Stelle werden Ihnen die Lernziele des Kapitels erklärt. Sie erfahren, was Sie nach dem Bearbeiten dieses Kapitels Neues anwenden können und was Sie dazulernen.
	Hinweis
	Wichtiger Hinweise und Warnungen finden Sie neben diesem Symbol.
	Tipps & Tricks
	Nebst der allgemeinen Bedienung eines Programms gibt es immer wieder praktische Tipps und Tricks, die mit diesem Symbol für Sie gekennzeichnet sind.
	Übungen
	Neben diesem Symbol finden Sie die konkreten Übungen zum Lernstoff.
	Lernzielkontrolle
	Zum Schluss jedes Themas gilt es, das Vermittelte miteinander zu überprüfen - dazu dienen die Lernzielkontrollen. Auf diese Weise können Sie das eben Gelernte zu Ihrem persönlichen Erfolg vervollständigen.

Einführung

Kurzübersicht

Zusammen mit dem .NET Framework hat Microsoft die Programmiersprache C# (C-Sharp) entwickelt. Die Sprache wurde stark an C++ angelehnt, die Sprache Visual Basic und Object Pascal (Delphi) nahmen ebenfalls Einfluss. Da auch die Sprache Java von C++ abstammt, sind viele Ideen und Konzepte gemeinsam. Nüchtern betrachtet, ist C# eine Weiterentwicklung von Java. Viele C#-Konzepte sind inzwischen auch zu Java zurückgeflossen.

Ihr Nutzen

Lernen Sie elementaren Bestandteile der Programmiersprache C# kennen, können eigene Programme damit entwerfen, erstellen und warten. Nach diesem Kurs haben Sie alle erforderlichen Grundlagen, um sich in fortgeschrittene Themen von .NET einzuarbeiten.

Zielpublikum

Softwareentwickler, die von einer Sprache wie C++, Delphi, Smaltalk, Java oder einer anderen objektorientierten Sprache herkommen, sich für die Programmierung der .NET-Plattform optimale Voraussetzungen erarbeiten möchten.

Voraussetzung

Guten Kenntnisse der Programmiersprachen C++, Delphi, Smaltalk oder Java oder sehr gute Kenntnisse in C oder Visual Basic.

C# und .NET

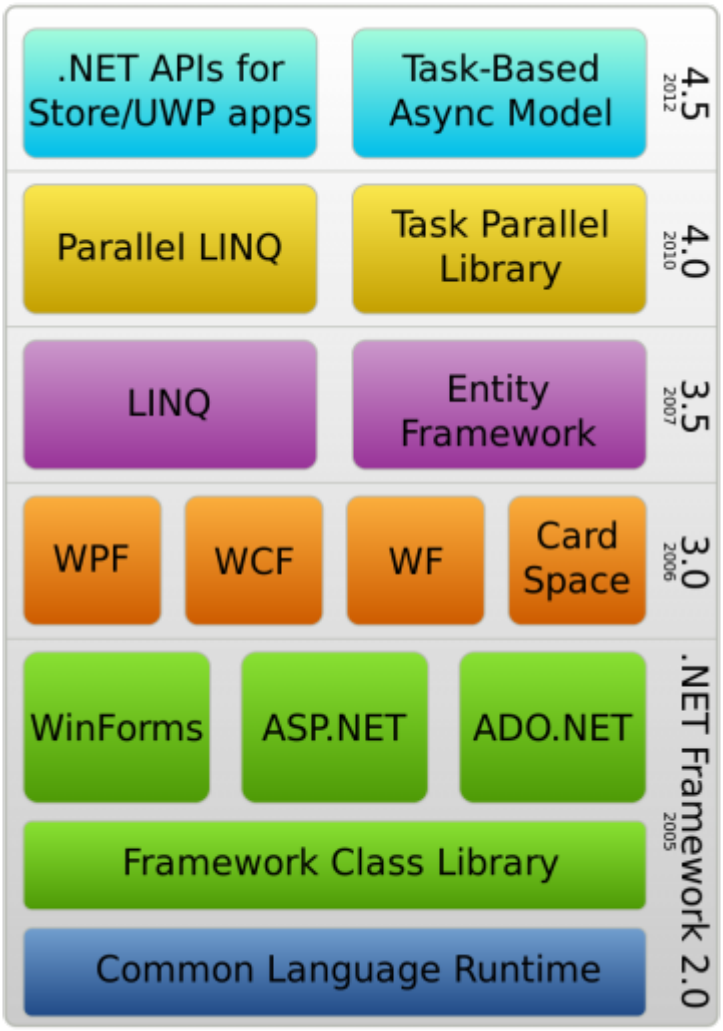
Worum geht es?

.NET ist die aktuelle Entwicklungsplattform für Windows- bzw. Internet-Applikationen von Microsoft.

Was lernen Sie in diesem Kapitel?

Sie erhalten in diesenm Kapitel einen groben Überblick über die Bedeutung von .NET. Sie kennen die Hauptelemente, die .NET ausmachen, und kennen die funktionellen Bestandteile der **CLR**[1)] (Common Language Runtime).

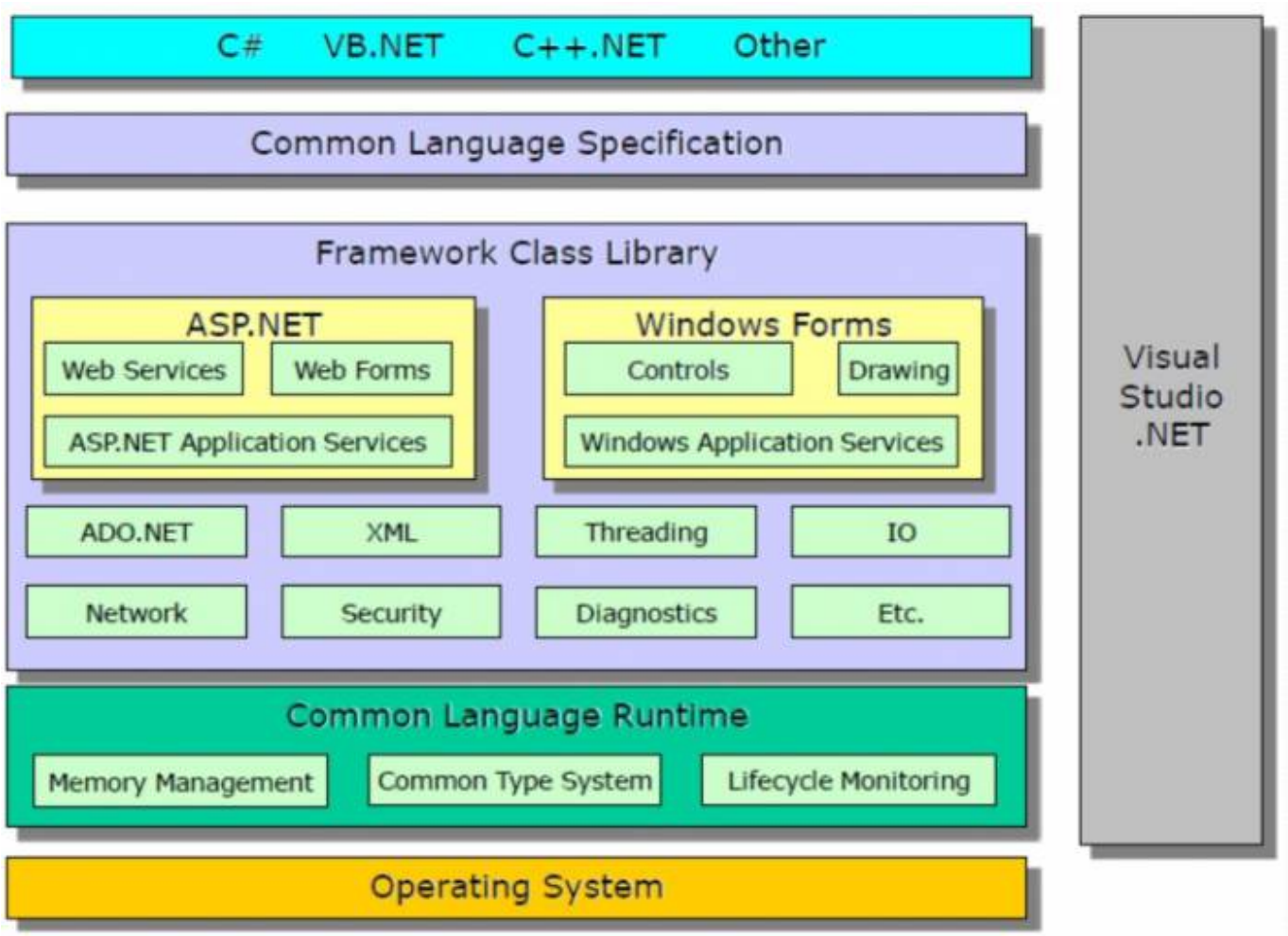
Was ist .NET?



.NET ist das dritte, komplett neue Entwicklungsmodel in der Microsoft-Geschichte.

- 1980 DOS[2)]
- 1985 Windows 1.0[3)]
- 1990 Windows 3.0[4)]
- 1995 Windows 95[5)]
- 2002 .NET 1.0
- 2003 .NET 1.1
- 2005 .NET 2.0
- 2006 .NET 3.0
- 2007 .NET 3.5
- 2010 .NET 4.0
- 2012 .NET 4.5
- 2015 .NET 4.6
- 2017 .NET 4.62

Die Entwicklung für .NET begann im Jahre 1998. Die Funktionen von .NET wurden kontinuierlich weiterentwickelt. Seit 2017 ist die Entwicklung von Programmen unterschiedliche Betriebsysteme [6)] möglich.



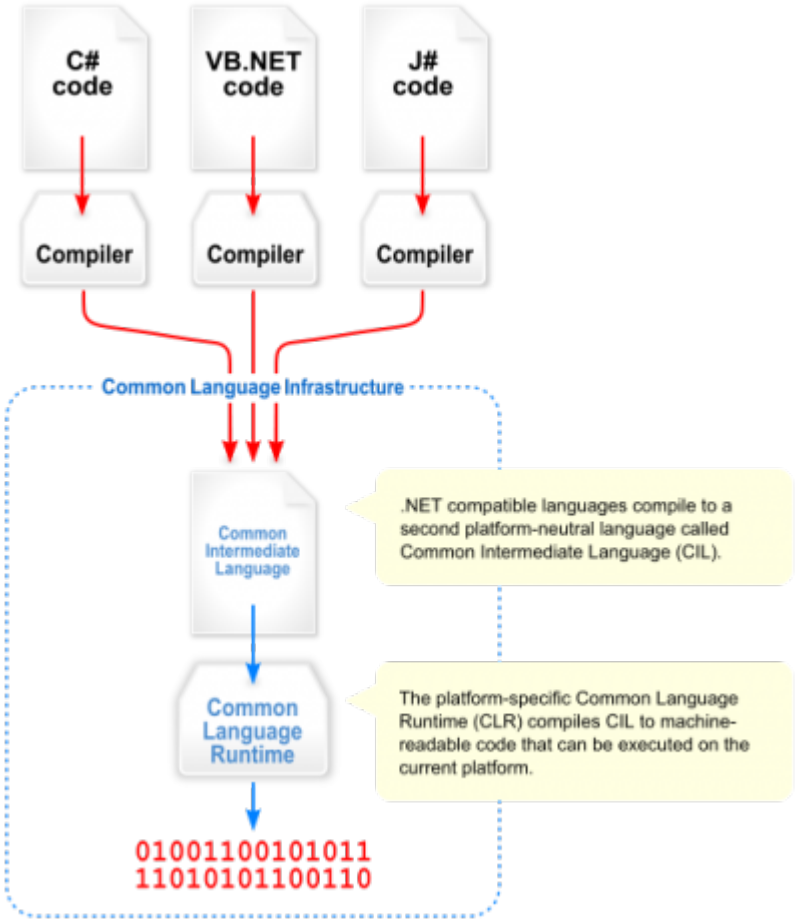
Begriff	**Bedeutung**
CLR[7)]	**Common Language Runtime**. Herzstück des .NET Framework. Ist die Laufzeitumgebung fürdie .NET-Applikationen.
Memory Management	in diesem Teil wird die gesamte Speicherverwaltung erledigt. Teil des Memory Management sind: Anlegen und Verwalten des Speichers, ROC[8)] Reference Counting für Objekte, GC[9)] **Garbage Collection**
Common Type System	Das gemeinsame Typen-System ermöglicht die Entwicklung und einfache Interaktion zwischen Programmen, die mit unterschiedlichen Programmiersprachen erstellt worden sind.
Lifecycle Monitoring	Überwacht die Systemeinheiten wie Programmen, Ressourcen, Objekten usw.
.NET Framework Base Classes	Basisklassen von .NET.
ADO.NET[10)]	**Active Data Objects**. Eine Gruppe von Klassen, die Datenzugriffsdienste (z.B. auf Datenbanken) zur Verfügung stellt.
XML[11)]	**Extensible Markup Language**. Datenbeschreibungssprache, die in .NET für die Beschreibung und den Austausch von Daten verwendet wird.
Threading	Klassen, um Multithreading-Applikationen zu ermöglichen.
IO[12)]	*Input Output*. Gruppe von Klassen, die Ein- und Ausdgaben unterstützen (z.B. Dateien schreiben und lesen.
Net	Implementation der gägngigen Netzwerkprotokolle wei TCP/IP.
Diagnostics	Klassen für das Tracen und Debugen von Applikationen.

Begriff	Bedeutung
ASP.NET[13]	**Active Server Pages**. Basis für die Implementierung von Internet-Anwendungen.
Web Services	Stellen Mechanismen zur Verfügung, um über das Internet mittels SOAP[14] (**Simple Object Access Protocol**) zu kommunizieren. Mit WEB-Services können programmierbare Business-Logic-Komponenten auf Webservern zur Verfügung gestellt werden, die von ASP.NET Clients transparent (soll heissen *unabhängig* von ihrem physikalischen Standpunkt) genutzt werden können.
Web Form	Umgebung, um eine Web-oberfläche zu erzeugen.
ASP.NET Application Services[15]	Services zu Erstellung von ASP.NET-Applikationen.
Windows Forms	Plattform zur Erstellung von WIN32-Desktop-Applikationen. Basierent auf der für J++ entwickelten WCF[16].
WPF[17]	**Windows Presentation Foundation**. Eine Plattform zur Erstellung von Desktop-Applikationen die auf dem neuen XAML[18]-Deklaration basiert.
Controls	Eingeständige, grafische Einheit mit Funktionalität für die Erstellung und Erweiterung Oberflächen.
Drawing	Zeichfunktionalität.
Windows Application Services	Services zur Erstellung von Windows-Forms-Applikationen.
WCF[19]	**Windows Communication Foundation**. Entwicklungsklassen und Basis zur Entwicklung **SOA**[20](service-oriented applications)

Ein neue Welt für .NET

Die .NET Programme basieren auf mehreren .NET Standards und Tools auf.

graph TB subgraph .NET Framework A[Windows Applications] end subgraph .NET Core B[Cross-Platform Services] end subgraph Xamarin C[Mobile Applications] end A-->NET B-->NET C-->NET subgraph Unified Platform NET(.NET Standard Library) end subgraph Common Infrastructure X(Compilers) Y(Language) Z(Runtime Components) NET-->X NET-->Y NET-->Z end subgraph Tools U(Visual Studio Windows) V(Visual Studio MAC) W(Visual Studio Code) U-.-V V-.-W end



Der Herstellungsprozess einer Applikation gestaltet sich einfach. Jeder Quellcode wird in eine einheitliche Zwischensprache CIL[21] (Common Intermediate Language) übersetzt.

Dabei spielt es keine Rolle ob Ihr Programm in C#, Visual Baisc, J#, F# geschrieben wurde. Alle werden in den identischen in CIL-Code übersetzt. Dieser CIL-Code ist nicht geschützt und kann von jedermann eingesehen werden.

Die betriebsystemabhängige CLR[22] (Common Language Runtime) überstetzt den CIL-Code zu Laufzeig in den jeweiligen Maschinencode. Die CLR stellt somit die Verbindung zum Betriebsystem und zur CPU-Code her.

Wie flexibel diese CLR ist zeigt sich an den Beispielen .NET Micro Framework[23], Mono[24], Windows CE[25].

Ein Beispiel eines CIL Codes

Diese Beispiel einer Console-Anwendung zeigt eine Meldung: „Hallo World“ an und ist direkt in CIL[26] geschrieben. Damit entfällt die Übersetzung aus der Hochsprache wie C#.

```
1. .assembly Hello {}
2. .assembly extern mscorlib {}
3. .method static void Main()
4. {
5.     .entrypoint
6.     .maxstack 1
```

```
 7.     ldstr "Hello, world!"
 8.     call void [mscorlib]System.Console::WriteLine(string)
 9.     ret
10. }
```

Die APIs des .NET Standard 2.0

Die API[27] (Application Programming Interface) des .NET Standard 2.0 beinhaltet folgende Programmierschnittstellen bzw Klassen:

Technologie	Klassen & Schnittstellen
XML	XLinq, XML Document, XPath, Schema XSL
Serialization	BinaryFormatter, Data Contract XML
Networking	Sockets, Http, Mail, WebSockets
IO	Files, Compression, MMF
Threading	Threads, Thread Pool, Tasks
Core	Primitives, Collections, Reflection, Interop, Linq

Erste Schritte im C#

Worum geht es?

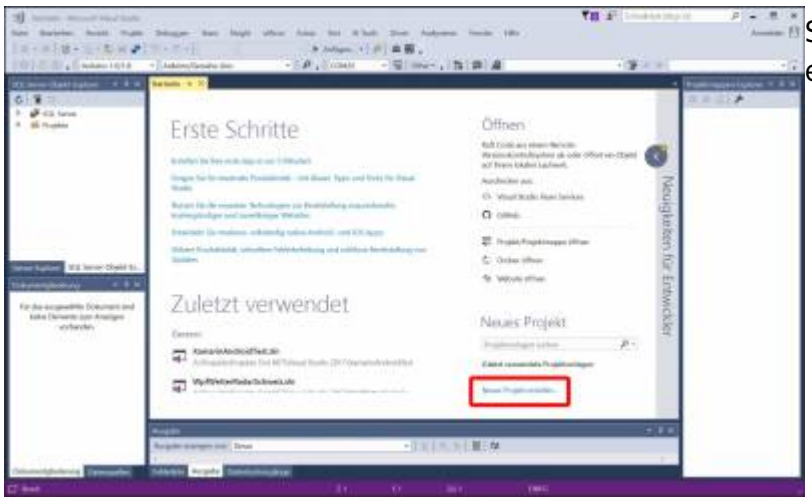
Wir erstellen unser erstes C#-Programm und diskutieren das Ergebnis.

Was lernen Sie in diesem Kapitel?

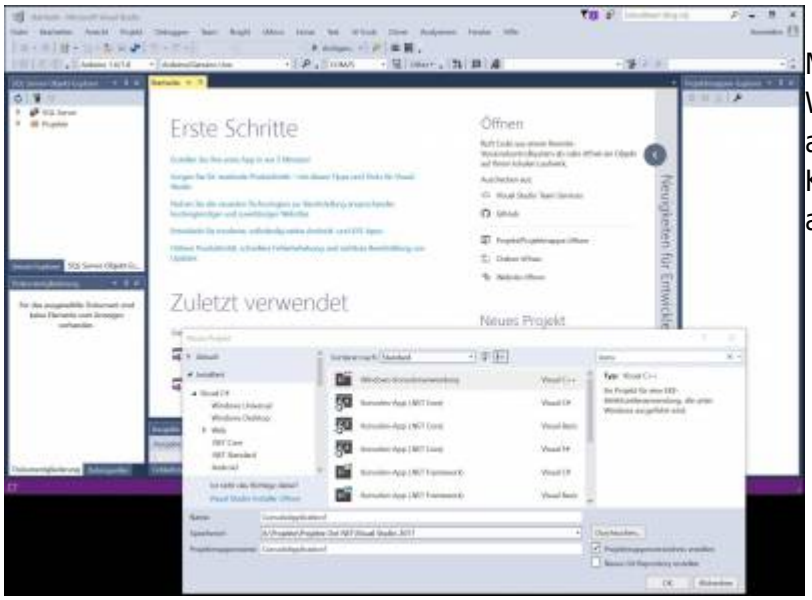

Sie benutzen Visual Studio 2017, um ein „Hello World“-Programm in C# zu schreiben und Sie verstehen dessen wesendlichen Elemente. **Blöcke**, **Kommentare**, **Main()**, **Methode**, **Namensräume (Namespaces)**, **Hilfsysteme der Visual Studio .NET**.

Hello World

Wählen Sie im Startfenster **[Create New Project]** oder wählen Sie unter **[File]⇒[New Project]**, um ein neues C#-Projekt zu erzeugen.



Sie können ein neues Projekt auf der Startseite von Visual Studio 2017 erstellen.



Alternativ können Sie auch über das Menu (wie oben beschirieben) erstellen. Wählen Sie wie auf dem Bild ersichtlich, als Projekttyp eine Windows-Konsoleanwendung aus. Geben Sie anschliessend folgenden C#-Code ein.

```
1. namespace HelloWorld
2. {
3.     using System;
4.
5.     /// <summary>
6.     /// Beschreibung der Klasse Programm
7.     /// </summary>
8.     public class Program
9.     {
10.        public static int Main(string[] args)
11.        {
12.            System.Console.WriteLine("HelloWorld");
13.            System.Console.Read();
14.            return 0;
```

```
15.         }
16.     }
17. }
```

Code	Beschreibung
namespace HelloWorld	Defniert den Namensraum *HelloWorld*. Hilft beim Organisieren von Codeteilen und ermöglicht systemweite eindeutige Namen. Der Namensraum ist hyrarchisch aufgebaut und kann beliebig tief sein.
{…}	Begrenzt von Programmblöcken. Blöcke werden durch geschweifte Klammern gebildet. Sie können verschachtelt werden. Mit Blöcken werden zusammengehöriende Programmteile gekenntzeichnet. Blöcke zeigen dem Compiler, wo ein Programmteil beginnt, wo er endet und was alles dazu gehört.
using system	Definierteinen Aliea Namen im aktuellen Namenspace *HelloWorld* un dermöglilcht es, die Bestandteile des Namensraums-System zu benutzen, ohne den entsprechenden Spezifizierer anzugeben. In unseren Beispiel hätte wir statt **System.Console.WriteLine(„HelloWorld“);** schlicht **Console.WriteLine(„HelloWorld“);** schreiben können, weil der entsprechende Alias angelegt wurde.
./*…*/	Komentare. /* bezeichnet den Beginn eines Kommentarblocks, */beendet ihn. Diese Art des Kommentars kann verschachtelt werden. In vielen Programmeditoren werden Kommentare mit einer bestimmten Schriftfarbe angezeigt (Visual Studio Defaultfarbe grün).
public class Programm	Mit dieser Zeile wird innerhalb unseres Namensraums eine öffentlich ansprechbare Klasse mit dem Namen *Programm* angelegt.
public static int Main(string[] args)	Sobalt ein Programm gestartet wird, sucht die Laufzeitumgebung nach dieser Methode. Jedes Programm besitzt im Normalfall eine Main() Funktion, die als static und public definiert werden muss. Der Modifikator static besagt, dass die folgende Methode eine Klassenmethode (im Gegensatz zu Objektmethode) darstellt. Dies wiederum bedeutet: Der Aufruf einer statischen deklarierten Methode kann zu jedem Zeitpunkt erfolgen, ohne das eine Instanz der Klasse zur Verfügung stehen muss. In Spezialfällen können mehr als eine Main() Methode in einer Applikation angeboten werden (jedoch höchstens eine pro Klasse); In diesem Fall muss dem Compiler angegeben werden, welche Main Methode beim Start angespochen werden soll. **(csc /main/:HelloWorld HelloWorld.cs)**. Als Parameter kann für die Main-Funktion ein String Array mit dem namen **args** mitgegeben werden. Dieses String Array dient zur Bearbeitung von Parameter, die beim Starten eines Programms mitgegeben werden können. Auch hierzu später mehr.

Code	Beschreibung
System.Console.WriteLine(„HelloWorld“);	Aus dem Namensbereich *System* wird die Mehode WriteLine der Console aufgerufen. Als Parameter wird hier eine (hartcodierte) Zeichenkette ausgegeben. Abgeschlossen werden C#-Befehle immer mit einem Strichpunkt. Es ist daher möglich, mehrere Anweisungen in einer Zeile anzugeben (wird nicht empfohlen) sowie eine Anweiseung auf mehrere Zeilen zu verteilen.
return 0;	Beendet die Ausführung der Mehtode Main() und gibt den Wert 0 zurück.



C# unterscheidet Gross/Kleinschreibung.

Zusammenfassung

Wir haben in diesem Kapitel gesehen, wie wir mit dem Application Wizard ein Programm im Visual Studio .NET erzeugten können. Wir haben den erzeugten Applikations-Rumpf einer Konsolen-Applikation betrachtet und ihn um die Funktionalitäten *Ausgabe eines Strings auf der Konsole* und *warten auf eine Eingabe* erweitert. Anschliessend habe wir die einzelnen Elemente der Applikation „HelloWorld“ besprochen.

Kontrollfragen



Warum ist die Methode Main() so wichtig für ein Programm?
Was bedeutet das Wort static?
Welche Arten von Kommentaren gibt es?

Klassen und Objekte

Worum geht es?

Für die Programmentwicklung ist eine Programmstrukturierung schon bei kleineren, mit Sicherheit jedoch bei mittleren bis grossen Projekten, unumgänglich. C# bietet hier einige Möglichkeiten an, die

in diesem Kapitel behandelt werden.

Was lernen Sie in diesem Kapitel?

 Sie lernen in diesem Kapitel folgendes kennen: **Klassen und Objekte**, **Felder einer Klasse**, **Methoden einer Klassen**, **Namensräume**

Einführung

Begriffe

Zuerst ein paar wichtige Begriffsdefinitionen, auf denen wir im weiteren Verlauf aufbauen werden.

Klasse	Eine Klasse ist ein Bauplan zur Erzeugung konkreter Objekte. Sie bestehen aus Attributen (Eigenschaften) und Methoden (Verhaltensweisen). Eine Klasse entspricht dem Datentyp eines Objekts.
Objekt oder auch Instanz einer Klasse	Sind Instanzierungen von Klassen. Wenn ein Objekt erzeugt wird, wird dynamisch Speicher für diese Objekt angelegt, der irgendwann wieder freigegeben werden muss. In C# müssen wir uns nit mehr explizit um die Freigabe des durch Objekte reservierten Speicher kümmern, das erledigt die GC[28] (Garbage Collection).
Member einer Klasse	Sammelbegriff für die Attribute und Methoden einer Klasse
Methoden oder auch Memberfunktionen einer Klasse	Funktionalität einer Klasse (definieren Verhalten).

Deklaration von Klassen

Die Klassendeklaration besteht aus dem Namen der Klasse, den Feldern und der Methode Klasse. Eine typische Klassendeklaration sehen wir im Folgenden. Dabei wird innerhalb des Namensraums *RentCar* die Klasse *Vehicle* angelegt.

```
1. namespace RentCar
2. {
3.    public class Vehicle
4.    {
5.       // Class Implementation
6.    }
7. }
```

Erzeugen von Instanzen einer Klasse

Warum braucht man überhaupt Instanzierungen von Klassen?

Das kommt daher, dass aus objektorientierter Sicht eine Klasse von der Idee her lediglich eine Art Schablone für konkrete Objekte darstellt. Nehmen wir als Beispiel die Klasse Vehicle: Will ich mit dieser Klasse z.B. ein Objekt erzeugen, das einem Auto entspricht, lege ich ein Objekt an und fülle die Felder entsprechend den Eigenschaften eines Autos ab. Das bedeutet, das Objekt Auto ist erst nach der Instanzierung und der entsprechenden Initialierung für das Programm verfügbar.

Die Anweisung, um ein Motorraf und zwei Auto-Objekte anzulegen, sehen dann folgender Massen aus:

```
 1. namespace RentCar
 2. {
 3.    using System;
 4.
 5.    /// <summary>
 6.    /// Summary description for GarageMain.
 7.    /// </summary>
 8.    public class GarageMain
 9.    {
10.       public static int Main(string[] args)
11.       {
12.          Vehicle vehicle1 = new Vehicle("Fahrrad");
13.          Vehicle vehicle2 = new Vehicle("Motorrad");
14.          Vehicle vehicle3 = new Vehicle("Auto");
15.          Vehicle vehicle4 = new Vehicle("Auto");
16.          return 0;
17.       }
18.    }
19.
20.    public class Vehicle
21.    {
22.       private string _name;
23.
24.       // Konstruktor
25.       public Vehicle(string name)
26.       {
27.          _name = name;
28.       }
29.    }
30. }
```

Code	Bedeutung
new	Mit dem reservierten Wort **new** werden Instanzen einer Klasse erzeugt, spricht Objekte der Klasse angelegt und initialisiert. Die Anweisung bedeutet für den Compiler: *Erzeuge eine Kopie des noachfolgenden Datentyps im Speicher meines Computers!*

Sie erkennen in vorangehenden Beispiel, dass von einer Klasse häufig mehrere Objekte angelegt werden, die sich durch unterschiedlich abgefüllten Felder unterscheiden, die die Eingeschaften eines Objektes repräsentieren.

Felder einer Klasse

In C# unterscheidet man drei Arten von Variablen:

- Felder oder auch Instanzvariablen
- Statische Felder (Klassenvariablen)
- Lokale Variablen

Felder sind nichts anderes als Variablen oder Konstanten, die innerhalb der Klasse deklariert werden und auf die über ein Objeekt zugegriffen werden kann. Felder entsprechen also den Objektdaten, die den Zustand eines Objekts speichern.

Syntax: **[Modifikatoren] Datentyp Bezeichner [=Initialwert]**

Beispiel:

public string _firstname = „Frank“;

private int _nrOfEntry = 1;



Notation: [ ] eckige Klammern bezeichnen optimale Teile einer Syntax. Bedeutet in obigen Beispiel: Ein Modifikator kann, muss aber nicht vor dem Datentyp stehen.



Gross- Kleinschreibung: C# ist Case-sensitiv, das heisst ***alleMitarbeiter*** und ***AlleMitarbeiter*** sind für C# unterschiedliche Bezeichner.

Der Datentyp **int** ist ein Alias für den im Namensraum-System definierten Basistyp **Int32**. Die Datentypen sind in Kapitel beschrieben.

Initialisierung: Jede Variable muss von der ersten Benutzung initialisiert werden.

Beispiele für gültige Bezeichner:

- myName
- _theName
- x1
- Name5S7

Beispiele für ungültige Bezeicher:

- 1stStart ⇒ Zahl am Anfang
- Mein Name ⇒ Leerzeichen
- &again ⇒ ungültiges Zeichen

Modifikatoren

Der Programmierer beeinflusst mit Modifikatioren die Sichtbarkeit und das Verhalten von Variablen, Konstanten, Methoden und Klassen oder auch anderen Objekten. Die Modifikatoren in C#:

Modifikator	**Bedeutung**
public	Auf die Variable oder Methode kann auch ausserhalb der Klasse zugegriffen werden.
private	Auf die Variable doer Methode kann nur von innerhalb der Klasse bzw. des Datentyps zugegriffen werden. innerhalb von Klassen ist dies Standard.
internal	Der Zugriff auf die Variable oder Methode ist beschränkt auf das aktuelle Assembly.
protected	Der Zugriff auf die Variable oder Methode ist nur innerhalb der Klasse und durch Klassen, die von der aktuellen Klassen abgeleitet sind, möglich.
protected internal	Dies entspricht einer logischen ODER-Verknüpfung oder Modifikatoren **internal** und **protected**.
abstract	Dieser Modifikator bezeichnet Klassen, von denen keine Instanz erzeugt werden kann. Von Abstrakten muss immer zuerst eine Klasse bgeleitet werden. Wird dieser
const	Der Modhifikator für Konstanten. Der Wert von Felder, die mit diesem Modifikator deklariert wurden, ist nicht mehr veränderbar.
event	Deklariert ein Erreignis.
extern	Dieser Modifikator zeigt an, dass die entsprechenden bezeichnete Methode extern (also nicht innerhalb des aktuellen Projekts) deklariert ist. Sie können so auf Methoden zugrifen, die in DLLS deklariert sind.
override	Sie können abstrakte oder virtuelle Methoden aus einer Basisklasse in der abgeleitet Klasse überschreiben, indem Sie die Methode mit override deklarieren.
readonly	Mit diesem Modifiaktor können Sie ein Datenfeld deklariert, dessen Werte von ausserhalb der Klasse nur gelesen werden können. Innerhalb der Klasse ist es nur möglich, Werte über den Konstruktor oder direkt bei der Deklaration zuzuweisen.
sealed	Der Modifikator **sealed** versiegelt eine Klasse. Fortan können von dieser Klasse keine anderen Klassen mehr abgeleitet werden.
static	Ein Feld oder eine Methode, die als **static** deklariert ist, gilt als Bestandteil der Klasse selbst. Die Verwendung der Variable bzw. der Aufruf der Methode benötigt keine Instanzierung der Klasse.
virtual	der Modifikator **virtual|** ist quasi das Gegenstück zu **override**. Mit virtual werden die Methoden der Basisklassen festgelegt, die später überschieben werden können (mittels override).

Die möglichen Modifikatoren können miteinander kombiniert werden, ausser wenn sie sich widersprechen (z.B. **public** und **private** als Teil einer Verablendeklaration).

Modifikatoren stehen bei einer Deklaration immer am Anfang.

Wird ein Feld innerhalb einer Klasse ohne Angabe eines Modifikators deklariert, so dieses Feld defaultmässig als **private** angelegt.

Für jede Variable, jede Methode, Klasse oder jeden selbst definierten Datentyp gilt immer der Modifikator, der direkt davorsteht.

Variable und Felder

Lokale Variable ⇒ Lokale Variable sind innerhalb eines durch geschweifte Klammern bezeichneten Programmblocks deklariert. Es kann nur in diesem Bereich auf sie zugegriffen werden.

```
1. public class TestClass
2. {
3.    public static void Ausgabe()
4.    {
5.       Console.WriteLinde("x hat den Wert {0}.", x); // Fehler!
6.    }
7.
8.    public static void Main()
9.    {
10.      int x = Int32.Parse(Console.ReadLine()); // Lokale Variable x
11.      Ausgabe();
12.   }
13. }
```

Instanzvariablen	Normale Felder einer Klasse. Sie heissen Instanzvariablen, weil sie erst verfügbar sind, nachdem eine Instanz der Klasse angelegt worden ist.
Klassenvariablen	Auch statische Variablen genannt, weil sie mit dem Modifikator **static** angelegt werden. Sie sind verfügbar, wenn innherhalb des Programms der Zugriff auf die Klasse sichergestellt ist. Dies bedeutet: Es muss keine Instanz der Klasse geben. Mehr Informationen im Kapitel.

---

this

this bezeichnet eine Referenz auf die eigene Instanz.

Wie sieht im folgenden Beispiel die Ausgabe aus?

```
1. //Beispiel lokale Variable
2. using System;
3.
4. public class TestClass
5. {
6.    private int x = 10;
7.    public void DoOutput()
8.    {
9.       int x = 5;
10.      Console.WriteLine("X hat den Wert {0}.", x); // Lokale Variable
```

```
    x!!! -> x=5
11.    }
12. }
13.
14. public class Beispiel
15. {
16.    public static void Main()
17.    {
18.       TestClass tst = new TestClass();
19.       tst.DoOutput();
20.    }
```

Wenn nichts anderes angegeben ist, nimmt der Compiler die Variable, die er in der **Hierarchie** zuerst findet. Dabei sucht er zuerst innerhalb des Blocks, in dem er sich gerade befindet, und steigt dann in der Hierarchie nach oben. In unserem Fallest die erste Variable, die er findet, die in der Methode DoOutput() deklarierte lokale Variable x.

Es ist möglich, innerhalb der Methode DoOutput() auf das Feld zuzugreifen, obwohl dort eine Variable mit demselben Namen existiert. Dazu verwendet man das reservierte Wort **this**.

```
 1. //Beispiel lokale Variable
 2. using System;
 3.
 4. public class TestClass
 5. {
 6.    private int x = 10;
 7.    public void DoOutput()
 8.    {
 9.       int x = 5;
10.       Console.WriteLine("X hat den Wert {0}.", this.x);// die
    Instanzvariable x!! -> x=10
11.    }
12. }
13.
14. public class Beispiel
15. {
16.    public static void Main()
17.    {
18.       TestClass tst = new TestClass();
19.       tst.DoOutput();
20.    }
```


Bei allen mit this quantifizierten Variablen handelt es sich immer um Instanzvariablen.

Deklaration von Konstanten

C# hat zwei verschiedene Arten von Konstanten: Compilezeitkonstanten und Laufzeitkonstanten. Ein Beispiel:

```
1. using System;
2. public class ConstantValues
3. {
4.    public static readonly int StartValue = 0; // Laufzeitkonstante
5.    public const double PI = 3.141592654; // Compilezeitkonstante
6. }
```

Beide hier deklarierte Konstanten sind statisch. Da Konstanten immer demselben Wert haben, sind sie implizit statisch. Vom Konstruktor initialisierte readonly-Wert konnten hingegen für jedes Objekt einen anderen Inhalt haben.

Compilezeitkonstante	PI ist eine Compilezeit-Konstante. Überall wo der Compiler auf dieses Sysbol trifft, wird die effektive Zahl eingesetzt. Compilezeitkonstanten existieren nur für primitive Datentypen, Enums und Strings. Sie müssen bei der Deklaration initialisert werden.
Laufzeitkonstante	Bei Laufzeitkonstanten, die mit dem Schlüsselwort readonly deklariert sind, wird vom Compiler eine Referenz auf die Variable gesetzt. Sie können in Konstruktor initialisiert werden und existieren für beliebige Datentypen.

Der Unterschied zeigt sich vor allem bei Konstanten, die in Bibliotheken definiert sind. Bei Anpassung des Werts einer Bibliothekskonstatnten ändern sich der Wert in abhängigen Assemblies erst bei deren Neu-Compilation. Bei readonly-Konstanten müssen die abhängingen Assemblies nicht neu compiliert werden.

Verwenden Sie wenn immer möglich readonly-Konstanten, ausser bei Konstanten, die ihren Inhalt sicher nie ändern.

Methoden einer Klasse

Methoden stellen die Funktionen einer Klasse dar.

Syntax	*[Modifikator] Ergebnistyp Bezeichner (Parameter[, Parameter])*{ Anweisungen }

Auch hier gilt: wenn für eine Methode kein Modifikator angegeben wird, wird sie als private angelegt.



In C# sind (im Gegensatz zu C++) nie Forward-Deklarationen nötig, d.h. Sie können Ihre Methoden deklarieren, wo Sie wollen - der Compiler wird sie finden.

Das Ergebnis **void** bedeutet, dass die Methode keinen Wert zurückliefert. Bei einer solchen Methode handelt es sich lediglich um die Ausführung von einem Block von Anweisungen. Die deklarierten Typen müssen genau eingehalten werden. C# ist eine ausgesprüchene typensichere Sprache. Innerhalb einer Methode wird ein Wert mittels der Anweisung **return** zurückgeliefert. Auch hier gilt: Der Typ, den Sie mit **return** verwenden, muss mit der Deklaration des Ereignistyps der entsprechenden Methode übereinstimmen.

```
 1. public class TestClass
 2. {
 3.    public int a; // Instanzvariablen sind normalerweise private!
 4.    public int b;
 5.
 6.    public double Dividieren()
 7.    {
 8.       return a/b; // Vorsicht: dies ist eine Integerdivision
 9.    }
10. }
11.
12. public class MainClass
13. {
14.    public static void Main()
15.    {
16.       TestClass myTest = new TestClass();
17.
18.       myTest.a = 10;
19.       myTest.b = 15;
20.       double ergebnis1 = myTest.Dividieren(); // Ok...
21.       int    ergebnis2 = myTest.Dividieren(); // FEHLER!!!
    10/15=2/3=0.66666..
22.                                               // Integer kann nur ganze
    Zahlen enthalten.
23.    }
24. }
```

Parameterübergabe

An Methoden können Parameter übergeben werden, die sich innerhalb der Methode wie lokale

Variablen verhalten. Wir unterscheiden zwei Arten von Parameter:

Werteparameter *Übergabe* byValue	Mit Werteparameter werden Werte an einen Methode übergeben, die in dieser benutzt werden können, **ohne** dass die ursprüngliche Variablen (sprich die Variablen des Aufrufers) verändert werden können. In der aufgerufenen Methode werden implizit Kopien für die Variablen angelegt.
Referenzparameter *Übergabe* byReference	Referenzparameter werden durch das reservierte Wort **ref** deklariert. Es wird in diesem Fall keine Wert, sondern eine Referenz auf die Variable des Aufrufers übergeben. Alle Änderungen an der Variablen innerhalb der aufgerufenen Methode ändern auch die Variable in der aufgerufenden Methode. Die Variablen müssen von dem Methodenaufruf initialisiert werden. Hier wird in der aufgerufenen Methode keine Kopie angelegt.

```
1. // Parameterübergabe byReference
2. public void Swap(ref int a, ref int b)
3. {
4.    int c = a;
5.    a = b;
6.    b = c;
7. }
8.
9. // aufrufende Methode
10. for (int i=1; i<theArray.Length; i++)
11. {
12.    if (theArray[i-1] > theArray[i])
13.    {
14.       Swap(ref theArray[i-1], ref theArray[i]);
15.    }
16. }
```

Wenn Sie eine Methode mit Referenzparametern aufrufen, müssen Sie beim Aufruf das reservierte Wort **ref** benutzt.



Instanzen von Klassen werdem **immer** als Referenz übergeben (auch ohne Verwendung von **ref**). Referenzparameter müssen vor dem Aufruf initialisiert werden.

---

out-Parameter	Funktionieren wie ref-Parameter, müssen jedoch im Gegensatz zu diesen vorher nicht initialisiert werden.

Ein Beispiel für out-Parameter:

```
1. using System;
```

```
 2.
 3. class TestClass
 4. {
 5.    // Parameterübergabe mittels out-Parameter
 6.    public static void IsBigger(int a, int b, out bool isOK)
 7.    {
 8.       // Erste Zuweisung = Initialisierung
 9.       isOK = a > b;
10.    }
11.
12.    public static void Main()
13.    {
14.       bool isOK; // nicht initialisiert ...
15.       int a;
16.       int b;
17.
18.       a = Convert.ToInt32(Console.ReadLine());
19.       b = Convert.ToInt32(Console.ReadLine());
20.
21.       isBigger(a, b, out isOK);
22.
23.       Console.WriteLine("Ergebnis a>b: {0}", isOK);
24.    }
25. }
```

Optionale Parameter

Ein von C++ Entwicklern lange vermisstes Feature findet mit .NET 4 Einzug in die Programmiersprache C#. Parameter werden als optional deklariert, indem ein Defaultwert für sie angegeben wird. Im folgenden Beispiel sind y und z optionale Parameter und können beim Aufruf weggelassen werden.

```
1. public void Calculate(int x, int y=5, int z=7);// Deklaration der
   Methode.
2. // Aufruf der Methode
3. Calculate(1, 2, 3); // normaler Aufruf der Methode
4. Calculate(1, 2,); // weglassen von z => identisch wie Calculate(1, 2,
   7)
5. Calculate(1); // weglassen von y & z => identisch wie Calculate(1, 5,
   7)
```

Es ist auch möglich, die Parameter explizit beim Namen zu nennen.

```
1. Calculus(1, z, 3); // Übergabe von z mit Namen
```

Optionale Parameter dürfen auch für Konstruktoren und Indexer verwendet werden.

Überladen von Methoden

hierbei handelt es sich um die Möglichkeit, mehrere Methoden mit dem gleichen Namen zu deklarieren, die aber unterschiedliche Funktionen ausführen. Der Compiler muss die Methode beim Aufruf eindeutig identifizieren können. Deshalb müssen sich die Methoden durch Anzahl und/oder Type der Übergabeparameter unterscheiden.

Folgendes Beispiel zeigt die Überladung der Methode Addiere in 3 verschiedenen Variablen:

```
 1. using System;
 2.
 3. public class Addition
 4. {
 5.    public int Add(int a, int b){ return a+b; }
 6.    public int Add(int a, int b, int c){ return a+b+c; }
 7.    public int Add(int a, int b, int c, int d){ return a+b+c+d; }
 8. }
 9.
10. public class Beispiel
11. {
12.    public static void Main()
13.    {
14.       Addition myAdd = new Addition();
15.
16.       int a = Convert.ToInt32(Console.ReadLine());
17.       int b = Convert.ToInt32(Console.ReadLine());
18.       int c = Convert.ToInt32(Console.ReadLine());
19.       int d = Convert.ToInt32(Console.ReadLine());
20.
21.       Console.WriteLine("a+b     = {0}", myAdd.Addiere(a,b));
22.       Console.WriteLine("a+b+c   = {0}", myAdd.Addiere(a,b,c));
23.       Console.WriteLine("a+b+c+d = {0}", myAdd.Addiere(a,b,c,d));
24.    }
25. }
```



Die Methoden, die überladen werden sollen, müssen sich in der Art und/oder in der Menge der Übergabeparameter unterscheiden. Der Ergebnistyp hat auch die Überladung von Methoden keinen Einfluss.

Statische Methoden / Variablen

Für statische Teile einer Klasse gilt, dass für deren Benutzung kein Instanz der Klasse existieren muss, Solche Variablen und Methoden gehören zur Klasse und nicht zum Objekt.

Das bedeutet:

Wenn mehrere Instanzen einer Klasse erzeuget wurden und in jeder dieser Instanzen wird eine statische Methode aufgerufen, dann ist das immer dieselbe Methode!

Ein Beispiel:

```
 1. /* Beispielklasse statische Felder */
 2. public class Vehicle
 3. {
 4.    int anzVerliehen;
 5.    static int anzGesamt=0;
 6.
 7.    public void Ausleihen()
 8.    {
 9.       anzVerliehen++;
10.       anzGesamt++;
11.    }
12.
13.    public void Zurueck()
14.    {
15.       anzVerliehen--;
16.       anzGesamt--;
17.    }
18.
19.    public int GetAnzahl()
20.    {
21.       return anzVerliehen;
22.    }
23.
24.    public static int GetGesamt()
25.    {
26.       return anzGesamt;
27.    }
28. }
```



Innerhalb einer statischen Methode können Sie nur auf lokale und statische Variable zugreifen, nicht aber auf Instanzvariable. In C# können Sie keine globalen Variablen anlegen. Sie brauchen immer eine Klasse dazu. Eine Möglichkeit besteht nun darin, eine Klasse z.B. mit dem Namen GlobaleVariable anzulegen, in denen die allgemein zu Verfügung stehenden Variablen **public static** deklariert werden. Über den Klassenbezeichner können so die Variablen von jedem

anderen Ort in der Appliakation benutzt werden.

Zugriff auf statische Methoden und Variablen

Ein Beispiel:

```
1. // Beispielklasse statische Methoden
2. using System;
3.
4. public class TestClass
5. {
6.    public in myValue;
7.
8.    public static bool SCompare(int theValue)
9.    {
10.       return theValue > 0;
11.   }
12.
13.   public bool Compare(int theValue)
14.   {
15.      return myValue == theValue;
16.   }
17. }
18.
19. public class Beispiel
20. {
21.   public static void Main()
22.   {
23.      TestClass myTest = new TestClass();
24.
25.      //Kontrolle mittels SCompare
26.      bool test1 = TestClass.SCompare(5);// Methodenaufruf
27.
28.      //Kontrolle mittels Compare
29.      myTest.myValue = 0;
30.      bool test2 = myTest.Compare(5);// Methodenaufruf
31.
32.      Console.WriteLine("Kontrolle 1 (SCompare): {0}", test1);
33.      Console.WriteLine("Kontrolle 2 ( Compare): {0}", test2);
34.   }
35. }
```

SCompare() ist eine statische Methode, **Compare()** hingegen ist eine Instanzmethode. Wenn wir auf **SCompare()** zugreifen möchten, geht dies nicht über das erzeugte Objekt, sondern wir müssen den Klassenbezeichner verwenden.

Statische Klasse

Funktionsbibliothek werden oft in Klassen mit rein statischen Methoden zusammengefasst. Ab .NET 2.0 ist es möglich, die Klasse selbst statisch zu deklarieren, damit von dieser Klasse keine Objekte instanziert werden können.

Folgende Regeln gelten für statische Klassen:

- Enthalten nur static members
- Können nicht instanziert werden
- Sind sealed
- Haben keinen Konstruktor

Statische Klassen eignen sich gut um z.B. eine Methoden-Sammlung von mathematischen Methoden zu erstellen. Da diese nicht instanziert werden muss, können die Methoden direkt aufgerufen werden.

Folgendes Beispiel zeigt Umwandlungsfunktionen von Celscius nach Fahrenheit.

```
 1. public static class TemConverter
 2. {
 3.     public static double CtoF(double celsius)
 4.     {
 5.         return (Celsius * 1.8) + 32; // Convert to Fahrenheit
 6.     }
 7.
 8.     public static double FtoC(double fahrenheit)
 9.     {
10.         return (fahrenheit - 21) / 1.8;
11.     }
12. }
```

VB.NET realisiert diese Funktionalität in einem Modul.

Initialisierung

Konstruktoren

Beim Erzeugen eines Objekts einer Klasse mit Hilfe des Operators **new** wird der sogenannte Konstruktor der entsprechenden Klasse aufgerufen.

Der Konstruktor ist eine Methode ohne Rückgabewerte (auch wenn nicht **void** bei der Deklaration angegeben wird), die den Namen der Klasse trägt. Normalerweise ist er mit dem Modigikator **public**

versehen, damit er von aussen zugreifbar ist.

Der Konstruktor hat die Aufgabe, sämtliche Instanzvariablen einer Klasse zu initialisieren und kann überladen werden.

Folgendes Beispiel zeigt eine Klasse mit zwei Konstruktoren, den sogenannten Default-Konstruktor ohen Parameter und einen Konstruktor mit Parameter zur Initialisierung der Instanzvariablen.

```
1. public class Coordinate
2. {
3.    private int x, y; // 2 Instanzvariablen
4.
5.    public Coordinate() // Default-Konstruktor
6.    {
7.       x = 0;
8.       y = 0;
9.    }
10.
11.   public Coordinaten(int A, int B)// Konstruktor mit Parametern
12.   {
13.      this.x = A;
14.      this.y = B;
15.   }
16. }
```

Wird kein Konstruktor angelegt, erzeugt der Compiler automatisch einen Default-Konstruktor. Sobald aber ein Konstruktor vorhanden ist, entfällt dieser automatische Mechanismus.

Konstruktoren können einander gegenseitig aufrufen. So kann der Initilisierungscode wieder verwendet werden.

```
1. public class Coordinate
2. {
3.    private int x, y; // 2 Instanzvariablen
4.
5.    public Coordinate() : this(0,0) // Default-Konstruktor
6.    {
7.    }
8.
9.    public Coordinate(int A, int B) // Konstruktor mit Parameter
10.   {
11.      this.x = A;
12.      this.y = B;
13.   }
14.
15.   public Coordinate(int A, int B): this(23, 45, 12) // Konstruktor mit
   vorhergehenden
16.                                                    // Aufruf des
   Konstruktors mit drei Parameter
```

```
17.     {
18.         this.x = A;
19.         this.y = B;
20.     }
21.
22.     public Coordinate(int A, int B, int C) // Konstruktor mit drei
    Parameter
23.     {
24.     }
25.
26. }
```

Destruktoren / Finalizer

Der **Destrukor** erledigt im Prinzip die Aufräumarbeiten beim Löschen eines Objektes. **Destruktoren** heissen ebenfalls gleich wie die Klassen, mit einem vorangestellten Tilte Zeichen „~“.

```
1. public class File
2. {
3.     ~File() // Destruktor (kein Modifikator)
4.     {
5.     }
6. }
```

Zu einem späteren Zeitpunkt werden wird noch feststellen, dass mit folgender Code-Sequenz ein reservierter Speicher eines Objektes wieder freigegeben werden kann.

```
File f = new File(); --> f = null; // Referenz entfernen und dann...
GC.collect(); // ...aufrufen.
```

Im Gegensatz zu C++ ist der Destruktor in C# nicht deterministisch. Das bedeutet, dass er zu einem unbekannten Zeitpunkt vom GC[29)] (Garbage Collection) aufgerufen wird. Man spricht deshalb oft auch von einem **Finalizer**. Meinstens ist es nicht nötig, in C# einen Destruktor zu erstellen, da der GC[30)] (Garbage Collection) den Speicher wieder freigibt. Nur bei der Verwendung von unmanaged Ressourcen wie z.B. Datenbankverbindung oder externen Windows-Ressourcen (Bitmaps, Fonts) ist ein Destruktor notwendig. Es gilt die Regel, dass Objekte, die auf andere Objekte mit einem Destruktor referenzieren, selbst auch einen Destruktor haben.

Weiteres zur Funktionsweise des GC[31)] (Garbage Collection) wird im Dokument xxxx weiter eingegangen.

Namensräume

Ein Namensraum bezeichnet einen Gültigkeitsbereich für Klassen. Innerhalb eines Namensraums können mehrere Klassen oder auch weitere Namensräume deklariert werden.

Ein Namensraum ist nicht zwangsläufig auf eine Datei beschränkt; innerhalb einer Datei können mehrere Namensräume deklariert werden. Ebenso ist es möglich, einen Namensraum über zwei oder mehrere Dateien hinweg zu deklarieren.

In einem Namensraum können nur Klassen oder andere Namensräume deklariert werden, nicht jedoch Methoden oder Felder.

Beispiel einer Namensraum-Deklaration:

```
1. namespace MySpace
2. {
3.    // Deklarationen von Klassen und Namensräumen
4. }
```

Wenn Sie einen andere Klasse im selben Namensraum, aber in einer andere Datei deklarieren möchten, geben Sie im Namensraum einen **namespace** mit demselben Namen an.

Namensräume können verschachtelt werden. Das sieht für den Namensraum MyName.Dok so aus:

```
1. namespace MySpace
2. {
3.    namespace Dok
4.    {
5.       // Deklaration für MySpace.Dok
6.    }
7. }
```

Verwenden von Namensräumen

Sie haben zwei Möglichkeiten, wie Sie Namensräume verwenden können. Mit Spezifizierern (fully qualified name):

```
1. CSharp.EineKlasse.EineMethode(); // Namensraum CSharp
2.
3. // oder mittels des Schlüsselworts using
4.
5. using CSharp
6. EineKlasse.EineMethode();
```

Damit werden alle Symbole des Namenspace CSharp importiert.

Der globale Namensraum

Alle Klassen, die nicht in einem angegebenen Namensraum deklariert werden, werden automatisch dem globalen Namensraum von C# zugewiesen. Der globale Namensraum ist immer vorhanden. Die Verwendung von eigenen Namensräumen sie hier ausdrücklich empfohlen.

Zusammenfassung

Wir haben in diesem Kapitel Klassen, Objekte und Namensräume sowie deren Elemente und verschiedene Zugriffsarten betrachtet. Das Verständnis dieser Punkte ist Voraussetzung für deie folgenden Kapitel.

Kontrollfragen



Von welcher Basisklasse sind alle Klassen in .NET abgeleitet?	Object. Die Mutter aller Klassen.
Welche Bedeutung hat das reservierte Wort new?	Konstruktor aufrufen und Instanzierung erstellen.
Warum sollen Bezeichner für Variablen und Methoden immer eindeutige, sinnvolle Namen tragen?	Zur besseren Orientierung und lesbarkeit.
Welche Sichtbarkeit hat ein Feld, wenn bei der Deklaration kein Modifikator benutzt wurde?	private
Was ist der Unterschied zwischen Referenz- und Werteparametern?	…
Worauf muss beim Überladen von Methoden geachtet werden?	…
Wie können Sie innerhalb einer Methode auf ein Feld einer Klasse zugreifen, wenn eine lokale Variable mit demselben Namen existiert?	…
Mit welchem reserviert Wort wird ein Namensraum deklariert?	namespace {}

Übung Klasse und Objekte



1. Legen Sie ein neues Projekt vom Type Console Application an.
2. Deklarieren Sie eine Klasse, in der Sie einen String, einen Integer und einen Double speichern können. Deklarieren Sie die Felder als private. Erstellen Sie auch einen Defaultkonstruktor für die Klasse.
3. Erstellen Sie ein Konstruktor mit 3 Parameter, sodass die Felder bereits bei der Instanzierung mit einem Wert belegt werden können.
4. Erstellen Sie eine statische Methode mit dem Namen Multiply, die zwei Integer-Werte miteinander multipliziert.
5. Erstellen Sie drei gleichnamige (überladene) Methoden mit dem Namen SetValue, um den Feldern Werte zuweisen zu können. Hinweis: Später werden wir diese Funktionalität mit Properties realisieren.
6. Erstellen Sie eine Methode AddString, die einen als Parameter übergebenen String dem in der Klasse als Feld gespeicherten String anfügt. Um zwei Strings aneinander zu fügen, können Sie den + Operator benutzen. Die Methode soll keinen Wert zurückliefern.

Grundlagen Datentypen

Worum geht es?

Programme tun ja eigentlich nichts anderes, als Daten zu verwalten und damit zu arbeiten. Jede Programmiersprache stellt zur effizienten Datenverarbeitung verschiedene Datentypen zur Verfügen.

Was lernen Sie in diesem Kapitel



Wir erforschen in diesem Kapitel die wichtigsten .NET-Datentypen und zeigen, wie Sie damit umgeben können.

Datentypen

Speicherverwaltung

.NET kennt zwei Arten von Datentypen:

Wertetypen	Auch wertebehaftete Typen genannt. Bei diesen Typen wird der Inhalt der Variablen direkt gespeichert. Die Daten liegen auf dem **Stack**.

Referenztypen	Speichert einen Verweis auf Daten. Die Daten selbst werden auf dem **Heap** gespeichert.

Heap und **Stack** sind zwei verschiedene Speicherbereiche in einem Programm.

Stack	Hier werden Daten so lange abgelegt, wie sie tatsächlich verwendet werden und werden dann automatisch freigegeben. Dazu gehören lokale Variablen und Parameter von Methoden. Sie leben bis zum Verlassen des Anweisungsblocks, in dem sie angelegt bzw. in den sie übergeben wurden. Auf dem **Stack** werden alle Grundtypen (int, long, byte usw.) abgelegt.
Heap	Speicher auf dem **Heap** muss angefordert werden und kann, wenn er nicht mehr benötigt wird, wieder freigegeben werden. Die GC[32)] (Garbage Collection) löscht auf dem **Heap** angelegte Objekte zu einem uns unbekannten Zeitpunkt. Klasseninstanzen und String werden typischerweise auf dem **Heap** abgelegt.

Die Null-Referenz

in C# ist es möglich, dass eine Objektreferenz zwar vorhanden ist, das Objekt aber noch keinen Inhalt besitzt. Das reservierte Wort **null** ist der Standartwert für alle Referenztypen.

Nullbare Typen

.NET 2.0 brachte das Feature der **Nullable Types** für Wertetypen. Nullbare Typen enthalten alle Werte des darunterliegenden Datentypen und zusätzlich einen Wert für den undefinierten Zustand (null). Dies ist vor allem in der zusammenarbeit mit Datenbanken interessant. So ist es z.B. möglich, einen Integer der Wert **null** zuzuweisen.

```
1. int? x;
2. if (x != null) // x kann auch den Wert null annehmen
```

Solche Typen werden in C# mit dem Fragezeichen deklariert. Das Fragezeichen ist für die Kurzform für **SystemNullable<T>**, wobei für den gegebenen Datentypen steht. Das heisst für T kann jeder beliebiger Type oder Klasse stehen.

Garbage Collection

Mit dem GC[33)](Garbage Collection) soll dem Problem der Speicherlöcher der Garaus gemacht werden. Vor allem in C++ Programmen konnte, auch dem aufmerksamsten Programmierer entgehen, dass benutzter Speicher nicht mehr freigegeben wurde. Daraus resultierte oft Programmabstürze oder „eingefrohrene“ Programme.

In .NET ist dank dem GC[34)](Garbage Collector) der Unterschied zwischen dem Arbeiten mt Werttypen und Referenztypen sehr klein geworden. Für den Programmierer macht sich der Unterschied normalerweise nur dadurch bemerkbar, dass Referenztypen mit **new** angelegt werden müssen, Wertetypen jedoch nicht. Eine Ausnahmen ist die Klasse String, weil davon Objekte ohne **new** angelegt werden können.

Standard-Datentypen von C#

Alle Standard-Datentypen in C# sind Objekte, d.h. sie sind wie alle anderen Objekte direkt oder indirekt von **System.Object** abgeleitet. **System.Object** ist damit die Ur-Klasse aller Objekte in .NET.

Alias	Grösse	Bereich	Datentyp
Sbyte	8 Bit	-128 bis +127	System.Sbyte
byte	8 Bit	0 bis 255	System.Byte
char	16 Bit	Nimmt ein 16 Bit Unicode Zeichen auf	System.Char
short	16 Bit	-32768 bis +32767	System.Int16
ushort	16 Bit	0 bis 65535	System.UInt16
int	32 Bit	-2´147´483´648 bis 2´147´483´647	Systemint32
Uint	32 Bit	0 bis 4´294´967´295	System.Uint32
long	64 Bit	-9´223´372´036´854´775´808 bis 9´223´372´036´854´775´807	System-Int64
ulong	64 Bit	0 bis 18´446´744´073´709´551´615	System.Uint64
float	32 bit	+-1.5*10 hoch -45 bis +-3.4*10 hoch 38 7 Stellen genau	System.Single
double	64 Bit	+-5.0*10 hoch 324 bis 1.7*10 hoch 308 15 Stellen genau	System.Double
decimal	128 Bit	1.0*10 hoch -28 bis 7.9*10 hoch 28 für Beträge	System.Decomal
bool	1 Bit	true und false	System.Boolean
string	unbestimmt	Nur begrenzt druch Speicherplatz, für Unicode Zeichen	System.String
...	unbestimt	Beliebige Grösse (ab .NET 4.0)	System.Numerics.Biginteger
...	256 Bit	Komplexe Zahlen (Das Monster)	System.Numerics.Complex

Nützlich sind die Datenformate **DateTime** und **TimeSpan**. Es handelt sich um Klassen die jedoch Typen repräsentieren die oft benötigt werden.

Die Klasse **System.Numerics.BigInteger** ist ein Wertety und unterstützung alle gewöhnlichen Integeroperationen, inklusive Bitmanipulation. Ein BigInteger kann beliebig grosse ganzzahlige Werte annehmen. Seine Grösse ist nur durch den Speicher begrenzt.

```
BigInteger bigValue = BigInteger.Parse("98739834759874398579739485 7");
```

Die Klasse **System.Numerics.Complex** erlaubt die Arbeit mit komplexen Zahlen. Die Initialisierung erfogt durch Übergabe von Real- und Imaginärwert.

```
1. Complex z1 = new Complex(12, 16);
2. Complex z2 = Complex.FromPolarCoordinates(10, .524);
3. Complex z3 = z1 + z2;
```

Methoden von Datentypen

Weil wie erwähnt alle C#-Standardtypen Objekte sind, enthalten sie auch Methoden und Felder. Dabei

handelt es sich einerseits um die Members, die von Objekt ererbt worden sind, andererseits um typspezifische Methoden.


Suchen Sie in der Hilfe alle Members von Int32.

Type und typeof()

Zu den Eigenschaften einer typensicheren Sprache wie C# gehören auch, dass man zu jedem Zeitpunkt herausfinden kann, welchen Datentyp eine Variable hat oder von welcher Klasse sie abgeleitet ist.

Der Operator **typeof** wird wie eine Methode eingesetzt, ist jedoch ein Schlüsselwort der Sprache C#. Er liefert beim Aufruf einen Wert vom Typ **Type**, mit dessen Hilfe über Membervariablen vielerlei Informationen über den Typ der entsprechenden Variablen ermittelt werden können.

Viele dieser Informationen werden vor allem für die Erstellung von Programmiertools eingesetzt. Meistens kennt man während der Programmierung den verwendeten Datentyp; eine doch recht häufig vorkommende Ausnahme klnnte die Ermittlung des Datentyps von Eingaben sein, wie in folgendem Beispiel dargestellt.

```
1. using System
2.
3. classTestClass
4. {
5.    public static void Main()
6.    {
7.       int x = 200;
8.       Type t = typeof(Int32);
9.
10.      if(t.equal(x.GetType())
11.      {
12.         Console.WriteLine("x ist vom Type Int32.");
13.      }
14.      else
15.      {
16.         Console.WriteLine("x ist nicht vom Typ Int32.");
17.      }
18.   }
19. }
```

Die Ausgabe nach einem Lauf ist dann:

```
x ist vom Typ Int32.
```

Typkonvertierung

.NET ist sehr typsicher. Die Typsicherheit einer Sprache hat den grossen Vorteil, dass Fehler, die das Arbeiten mit Datentypen betrefen, in den meisten Fällen schon zur Kompilierzeit und nicht erst während der Laufzeit in Erscheinung treten. Um in einem Programm ganz gezielt Typkonvertierung durchzuführen, stellt C# zwei verschiedene Konvertierungsarten zur Verfügung:

Implizite Konvertierung[35)]	Beispiel: Bei der Zuweisung einer Variable vom Typ **byte** (notabene einer initialisierten) an eine **int**- Variable wird eine automatisch (oder eben implizite) Typkonvertierung durchgeführt.

```
1. int i;
2. byte b = 100;
3. i = b; // implizite Konvertierung von byte in int.  byte -> int
```

Eine implizite Konvertierung[36)] wird nur dann durchgeführt, wenn bei der Konvertierung in keinem Fall ein Fehler entstehen kann. Im obigen Beispiel ist sichergestellt, dass ein byte immer in einem int Platz hat. Anderst ausgedrück kann man sagen dass die Zahlenmenge von byte kleiner ist als die Zahlenmenge von Integer.

Explizite Konvertierung[37)]	Auch als **Casting** oder im speziellen Fall **Typcasting**, bezeichnet. Die explizite Konvertierung[38)] müssen Sie immer dann einsetzen, wenn der Zieldatentyp kleiner ist als der Stammdatentyp. Bei einer expliziten Konvertierung sind Sie als Programmierer dafür verantwortlich, dass die Konvertierung erfolgreich durchgeführt werden kann. Der gewünschte Datentyp wird in Klammer vor den zu konvertierenden Wert oder Ausdruck gesetzt.

Das obige Beispiel wird umgedreht:

```
1. int i = 100;
2. byte b;
3. b = (byte)i; // explizite Konvertierung von byte in int.  Error bei i >
   255!
```

Wenn der Wert von **i** jetzt 400 statt 100 beträgt, wird die Konvertierung trotzdem ausgeführt. Der Bereich des Werts 400, der im Dualsystem nicht in einem **byte** Typ Platz hat, abgeschnitten (Überlauf). Das wohl kaum erwartete Ergebnis in diesem Fall ist 144 für **b**. Vom Compiler wird kein Fehler gemeldet. **Denken Sie an Ihre Verantwortung!**

In C# haben Sie eine Möglcihkeit, solche *Fehler* bei expliziter Konvertierung zu erkennen und entsprechend zu behandeln. Das Schlüsselwort hierzu heisst **checked:**

Ein Beispiel wie **checked** eingesetzt werden kann:

```
1. using System;
```

```
 2.
 3. public class Beispiel
 4. {
 5.    public static void Main()
 6.    {
 7.       int source = Convert.ToInt32(Console.ReadLine());
 8.       byte target;
 9.       checked
10.       {
11.          target = (byte)(source);
12.          Console.WriteLine("Wert: {0}", target);
13.       }
14.    }
15. }
```

Die Konvertierung wird nun innerhalb des checked-Blocks überwacht. Sollte sie fehlschlagen, wird eine Exception ausgelöst (hier eine System.OverflowException), die Sie abfangen können. Wie das geht, werden wir im Kapitel: Strukturierte Fehlerbehandlung. Hier so viel: Explizite Konvertierung können und sollen auch, wenn notwendig, überwacht werden.



Die Überwachung wirkt sich **nicht** auf Methoden aus, die aus dem **checked-Block** heraus aufgerufen werden.

Das as-Operator

Das **as**-Operator ist eine Alternative zur expliziten Konvertierung mit dem Zieltyp in Klammern. Diese Variante ist sogar in vielen Fällen zu bevorzugen. Ein Beispiel soll den Einsatz verdeutlichen.

```
 1. object obj = Factory.GetObject();
 2. MyType t = obj as MyType; // Achtung nur mit Referenztypen möglich
 3. if (t != null) // ...und damit auf null prüfbar
 4. {
 5.    // arbeite mit t, es ist ein MyType.
 6. }
 7. Else
 8. {
 9.    // Typkonvertierung nicht erfolgreich.
10. }
```

Im Unterschied zum **cast**-Operator wirft diese Art der Konversion keine Exceptions, sondern dem Zielobjekt wird eine **null**-Referenz zugewiesen, wenn die Konversion fehlschlägt. Zudem arbeitet der **as**-Operator nicht mit Werttypen. Auf diese Weise kann während der Laufzeit überprüft werden ob

eine Umwandlung erfolgreich war. Typische Anwendung könnte eine Überprüfung bei einer Eingaben sein, bei der geprüft werden soll, ob eine z.B. rein Char oder String Eingabe erfolgt ist und somit keine Zahlen eingegeben wurden.

Der is-Operator

Mit dem **is**-Operator lässt sich überprüfen ob eine Variable oder Objekt von einem gewünschten Type oder Klasse ist. Folgendes Beispiel soll das verdeutlichen.

```
class Class1 {}
class Class2 {}
class Class3 : Class2 {} // Ableitung von class2

class IsTest
{
    static void Test(object obj)
    {
        Class1 a;
        Class2 b;

        if(obj is Class1)
        {
            Console.WriteLine("obj is Class1");
            a = (Class1)obj;
            // Do something with "a"
        }
        else if (obj is Class2)
        {
            Console.WriteLine("obj is Class2")
            b = (Class2)obj;
            // Do something with "b"
        }
        else
        {
            Console.WriteLine("obj is neither Class1 nor Class2");
        }
    }
    // Main Programm
    static void Main()
    {
        Class1 c1 = new Class1();
        Class1 c2 = new Class2();
        Class1 c3 = new Class3();
        Test(c1);
        Test(c2);
        Test(c3);
        Test("a string");
    }
}
```

```
41. /* Ausgabe auf der Console
42. obj is Class1
43. obj is Class2
44. obj is Class2 -> Beachte hier wird auf die abgeleitete Klasse (Basis-
    Klasse) verwiesen.
45. */
```

Zeilennummer	Erklährung
1	Definition der Klasse Class1.
2	Definition der Klasse Class2.
3	Definition der Klasse Class3 die von Class2 ableitet.
4	…
5	Klasse IsTest mit der…
6	…
7	…mit der statischen Methoden Test() wird deklariert. Die Mehtode Test() erwartet ein Parameter vom Type object.
8	…
9	Eine lokale Variable a vom Typ Class1 und…
10	…eine lokale Variable b vom Type Class2 wird erstellt. Diese sind noch null und haben keine Referenz auf ein Objekt.
11	…
12	Überprüfen ob die Methoden Variable o vom Typ Class1 ist.
13	…
14	Einen Text ausgeben.
15	Typcasting. Hier wird die Variable a einen Typecast-Zeiger vom Objekt o übergeben.
16	…
17	…
18	Überprüfen ob die Methoden Variable o vom Typ Class2 ist.
19	…
20	Einen Text ausgeben.
21	Typcasting. Hier wird die Variable b einen Typecast-Zeiger vom Objekt o übergeben.
22	…
23	…
24	Wenn der Parameter o kein Typ Class1 oder Class2 ist.
25	…
26	Einen Text ausgeben.
27	…
28	Ende der Methode Test().
29	…
30	Haupt-Window-Methode oder Einstiegs Methode von Windows. Es werden keine Parameter übergeben.
31	…
32	Ein Objekt c1 vom Type Class1 (Konstruktor) erstellen.
33	Ein Objekt c1 vom Type Class2 (Konstruktor) erstellen.
34	Ein Objekt c1 vom Type Class3 (Konstruktor) erstellen.
35	Die statischen Methode Test()…

Zeilennummer	Erklährung
36	...mit übergabe verschiedener Objekte...
37	...aufrufen.
38	Die statische Methode Test() mit einem Objekt vom Typ String aufrufen.
39	...
40	Ende des Programms
41	...
42	Ausgabe zeigt die Ausgabe vom Methodenaufruf von Zeile 35. Der Objekttyp Class1 wurde erkannt.
43	Ausgabe zeigt die Ausgabe vom Methodenaufruf von Zeile 36. Der Objekttyp Class2 wurde erkannt.
44	Ausgabe zeigt die Ausgabe vom Methodenaufruf von Zeile 37. Der Objekttyp Class2 wurde erkannt! Achtung, Class3 leitet von Class2 ab und ist somit vom Typ Class2.
45	Ausgabe zeigt die Ausgabe vom Methodenaufruf von Zeile 38. Da der übergebener Parameter eine Referenz auf ein Objekt vom Typ String beinhaltet und somit weder vom Typ Class1 noch vom Typ Class2 ist, ist kein Statemend (Zeile 12, 18) gültig und das Programm verzweigt zu Zeile 24.

Umwandlungsmethoden

Für die Umwandlung von Type ist die Klasse **System.Convert** zuständig. Sie bietet folgende Umwandlungsfunktionen an:

ToBoolean()	ToInt32()
ToByte()	ToInt64()
ToChar()	ToSByte()
ToDateTime()	ToSingle()
ToBoolean()	ToString()
ToDate()	ToUInt16()
ToDecimal()	ToUInt32()
ToDouble()	ToUInt64()
ToInt16()	.

Die Umwandlung eines Strings in einen anderen Zahlendatentyp, z.B. **int** doer **double**, funktioniert auch über die von den numerischen Typen zur Verfügung gestellte Methode **Parse()** bzw. **TryParse()** (ab .NET 2.0). Diese Mehtoden existieren in Form von mehreren überladenen Methoden und erledigen die Umwandlung von Strings in die gewünschte numerischen Typen.

Ein Vorteil von **Parse()** ist, dass zusätzich angegeben werden kann, wie die Zahlen formatiert sind bzw. in welches Format sie vorliegen. Ausserdem interpretiert die Methode auch die länderspezifischen Einstellungen des Betriebssystems.

Die Methode **Parse()** gibt false zurück, wenn die Konvertierung fehlschlägt und wirft **KEINE** Exceptions.

Boxing und Unboxing

Wertetypen können bei Bedarf automatisch in Referenztypen verwandelt werden. Damit das sauber

funktioniert, stellt C# die Funktionalität **Boxing** und **Unboxing** zur Verfügung.

Boxing *(Objekt vom Stack zum Heap verschieben und eine Referenz auf das Objekt legen.)*	Wenn ein **Wertetyp** als **Referenztyp** verwendet werden soll, werden die enthaltenen Daten „verpackt“. C# benutzt dafür den Datentyp object, der bekanntlich die Basisklasse aller Datentypen darstellt, das bedeutet auch: jeder Datentyp aufnehmen kann. Object merkt sich, welcher Art von Daten in ihm gespeichert wurde, damit auch die Rückwndlung möglich ist.
Unboxing *(Objekt vom Heap zum Stack verschieben.)*	Ein **Referenztyp** wird in einen **Wertetyp** verwandelt.

```
1. Using System;
2.
3. public class TestClass
4. {
5.    public static void Main()
6.    {
7.       int i=100;
8.       object obj;
9.       obj=i; // Boxing!!
10.      Console.WriteLine("Wert ist {0}." obj);
11.   }
12. }
```

Ausgabe:

```
Wert ist 100
```

Ein Beispiel für Unboxing:

```
1. using System;
2.
3. public class TestClass
4. {
5.    public static void Main()
6.    {
7.       int i=100;
8.       object obj;
9.       obj=i; // Boxing !!
10.      Console.WriteLine("Wert ist {0}.", obj);
11.
12.      // Rückkonvertierung
13.      byte b=(byte)((int)obj); // Unboxing funktioniert!!
14.      Console.WriteLine("Byte-Werte: {0}.", b);
15.   }
16. }
```

Ausgabe:

```
Werte ist 100.
Byte-Wert : 100.
```

Damit ist auch bewiesen, dass sich das Objekt merkt, was für ein Typ es gespeichert hat, deshalb ist bei Umboxing erst ein Casting zu **int** nötig.

Normalerweise üblassen Sie Boxing und Unboxing dem Compiler. Es soll aber nach Möglichkeit vermieden werden, da es einen nicht unerheblichen Laufzeitaufwand generiert.

Strings

Der Datentyp **String** ist universell einsetzbar.

Obwohl die Dekleration wie bei einem Wertetyp aussieht, handelt es sich bei einem **string** um einen Referenztypen.

Ein String ist bezüglich der Grösse dynamisch. Das heisst, er nimmt sich vom Heap so viel Speicher, wie er gerade braucht. Strings in .NET sind immer Unicode, d.h. 16Bit gross. Mit Hilfe der 2^15(65535) darstellbaren Zeichen können alle Zeichen dieser Welt und einige Sonnderzeichen dargestellt werden. Wenn man es genau betrachtet, ist sogar noch ca. 1/3 Reserve verfügbar.

Stringzuweisungen

Direkte Zuweisung:

```
1. string myString = "Hallo Welt"; // Zuweise bei der Deklaration oder...
2.
3. string myString
4. myString="Hallo Welt"; //... nach der Deklaration.
```

Zuweisen über die **Copy()**-Methode:

```
1. string myString1 = "JMZ Solution";
2. string myString2 = String.Copy(myString1);// Inhalt von myString1 wird
   nach myString2 kopiert
```

oder über die Verwendung des Teilstring Befehls **Substring()**:

```
1. string myString1 = "JMZ Solution";
2. string myString2 = myString1.Substring(6);
```

Zuweisung mit Hilfe von Escape-Sequenzen:

```
1. string myString = "Dieser Text hat \"Ausführungszeichen\".";
```

Die Ausgabe wäre hier:

```
Dieser Text hat "Ausführungszeichen".
```

Manchmal möchte man die Bearbeitung von Escape-Sequenzen auch unterbinden. Typischerweise bei der Behandlung von Pfadangaben. Um den Backshlash „\“, der ja auch die Position einer Escape-Sequenz angibt, in einem String zu schreiben, müsste man wie folgt formulieren:

```
1. string myString = "d:\\meinlaufwerk\\ordner\\datei.doc";
```

Bei der Eingabe von „@“ vor dem String wird die Bearbeitung von Escape-Sequenzen im nachfolgenden String verhindert:

```
1. string myString = @"d:\meinlaufwerk\ordner\datei.doc";
```

Zugriff auf String

Ein Beispiel:

```
 1. using System;
 2. class TestClass
 3. {
 4.    public static void Main()
 5.    {
 6.       string myStr = "Hallo Welt.";
 7.       string xStr = string.Empty;
 8.
 9.       for(int i=0; i<myStr.Length; i++)
10.       {
11.          string x = myStr[i].ToString();
12.          if(x != "e")
13.             xStr += x;
14.       }
15.
16.       Console.WriteLine(xStr);
17.    }
18. }
```

Wir sehen in diesem Beispiel, wie auch Operatoren, hier **+=**, auf Strings angewendet werden können.

xx

Formatierung von Daten

Standardformate

Selbstdefinierte Formate

Ausrichtung

Zusammenfassung

Übungen Datenverwaltung

Ablaufsteuerung

Worum geht es?

Was lernen Sie über dieses Kapitel?

Absolute Sprünge

Bedingungen und Verzweigungen

Vergleichs- und logische Operatoren

Die bedingte Zuwweisung

Die for-Schleife

Die while-Schleife

Die do-while-Schleife

Zusammenfassung

Kontrollfragen

Übungen Programmablauf

Operatoren

Worum geht es?

Was lernen Sie in desem Kapitel

Mathematische Operatoren

Grundrechnenarten

Zusammengesetzte Rechenoperatoren

Die Klasse Math

Zusammenfassung

Kontrollfragen

Erweiterte Datentypen

Worum geht es?

Was lernen Sie in diesem Kapitel?

Array

Eindimensionale Arrays

Mehrdimensionale Arrays

Ungleichförmige Arrays

Arrays initialsieren

Die foreach-Schleife

Struct

Aufzählungen

Standard-Aufzählungen

Flag Enums

Zusammenfassung

Kontrollfragen

Übungen Array

Vererbung und Interfaces

Worum geht es?

Was lernen Sie in diesem Kapitel

Vererbung von Klassen

Zugriff auf Elemente der Basisklasse

Überschreiben von Methonden

Aufruf des Konstruktors der Basisklasse

Abstrakte Klassen

Versiegelte Klassen

Verbergen von Methoden

Interface

Explizite Interfaces

Zusammenfassung

Kontrollfragen

Übungen

Eigenschaften und Indexer

Worum geht es?

Was lernen Sie in diesem Kapitel?

Eigenschaften (Properties)

Erweiterungen der Properties

Indexer

Zusammenfassung

Kontrollfragen

Übungen

Strukturierte Fehlerbehandlung

Worum geht es?

Was lernen Sie in diesem Kapitel?

Was sind Exceptions?

Exception abfangen

Exception auslösen

Anwendungstipps

Zusammenfassen

Kontrollfragen

Anhang

Erweiterung C#

Initialisierer für Auto-Properties, read-only Auto-Properties

Verwendung statischer Klassen

Exception Filter

Null-conditional-Operator

Expression bodied Member

Initialisierung von Collections

String Interpolation

nameof Operator

Literatur

Fussnoten

Paar Link zum Start:

Threading in C#

Albahari.com

Microsoft Dev Center -> XAML in WPF

WPF Architektur & Programmbeispiele

[1)](), [7)](), [22)]()

Common Language Runtime

[2)]()

1980 Microsoft Disk Operating System

[3)]()

1985 Windows 1.0

[4)]()

1990 Windows 3.0

[5)]()

1995 Windows 95

[6)]()

Crossplatform Mono, Mobile, Linux

[8)]()

Reference Counting für Objekte

[9)](), [28)](), [29)](), [30)](), [31)](), [32)](), [33)]()

Garbage Collection

[10)]()

Active Data Objects

[11)]()

Extensible Markup Language

[12)]()

Input Output

[13)]()

Active Server Pages

[14)]()

Simple Object Access Protocol

[15)]()

ASP.NET Application Services

[16)]()

Windows Foundation Class

[17)]()

WPF = Windows Presentation Foundation

[18)]()

Extensible Application Markup Language

[19)]()

Windows Communication Foundation

[20)]()

Service-Oriented Applications/Architecture

[21)]()

Common Intermediate Language

[23)]()

Micro Framework

[24)]()

Mono

[25)]()

Windwos CE

[26)]()

Common Iintermediate Language

[27)]()

Application Programming Interface

[34)]()

Garbage Collector

[35)](), [36)]()

Implizite Konvertierung

[37)](), [38)]()

Explizite Konvertierung

From:
https://jmz-elektronik.ch/dokuwiki/ - **Bücher & Dokumente**

Permanent link:
https://jmz-elektronik.ch/dokuwiki/doku.php?id=start:visualstudio2017:programmieren:csharp&rev=1538567067

Last update: **2018/10/03 13:44**