1 Einleitung

1.1 Kursziele

In diesem Kurs lernen Sie Windows-Anwendungen mit der Programmiersprache C# und der Entwicklungsumgebung Visual Studio zu erstellen. Zuerst erarbeiten wir die Grundlagen von C# und der objektorientierten Programmierung. Nach den ersten Versuchen mit Konsolen-Anwendungen werden wir uns im zweiten Teil mit der Windows-Forms-Bibliothek vertraut machen und Programme mit grafischen Oberflächen erstellen.

1.2 Programmiersprache C#

Mit der Programmiersprache C# lassen sich Anwendungen für das .NET Framework und .NET Core erstellen. Wie im Stammbaum der Programmiersprachen in Abbildung 1 zu sehen ist, weist C# viele Ähnlichkeiten mit den Programmiersprachen C und C++ auf, enthält aber auch Elemente von Java.

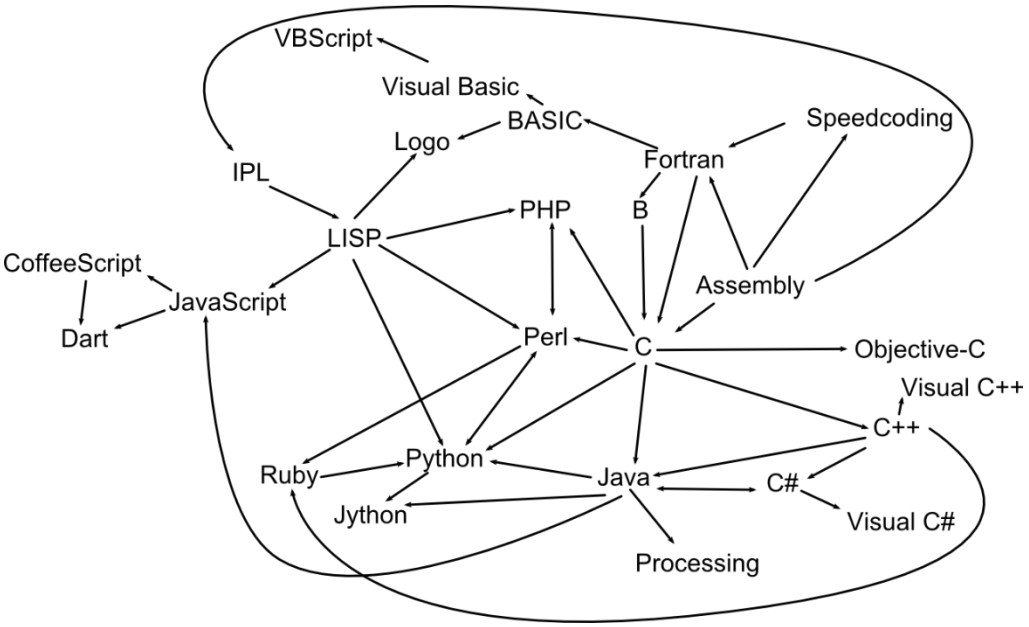

Abbildung 1: Stammbaum der Programmiersprachen Fehler! Verweisquelle konnte nicht gefunden werden.**]**

C# wurde von Microsoft speziell für das .NET Framework entwickelt und ist eine rein objektorientierte Sprache. Mitgearbeitet bei der Entwicklung von C# hat auch Anders Hejlsberg, der Entwickler von Turbo Pascal und Delphi. C# verbindet die Einfachheit von Visual Basic mit der Leistungsfähigkeit von C++, ohne komplizierte Konstrukte wie z.B. Zeiger zu verwenden.

1.3 .NET Framework / .NET Core

1.3.1 Übersicht

Das .NET Framework ist eine von Microsoft entwickelte Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen. Das Framework besteht aus einer Laufzeitumgebung (in der die Programme ausgeführt werden) sowie einer Sammlung von Klassenbibliotheken, Programmierschnittstellen und Dienstprogrammen (Services). Die Bibliotheken bieten viele vorgefertigte Lösungen für gängige Aufgaben an. Somit kann sich der Programmierer ganz auf die anwendungsspezifischen Programmfunktionen konzentrieren.

Aktuell arbeitet Microsoft vermehrt mit .Net-Core, bei .NET-Core wird nur installiert was auch gebraucht wird (bei einem Framework wird Alles installiert, .NET-Core ist in Pakete geteilt). Benötigte Pakete werden automatisch installiert, .Net-Core läuft auf Windows, Mac und Linux. Es können auch Apps für iOS und Android in C# entwickelt werden.

.NET Core ist eine freie und quelloffene Software-Plattform, die zur Entwicklung und Ausführung von Anwendungsprogrammen dient und unter der Koordination von Microsoft entwickelt wird.

.NET Core ist gleichzeitig ein Modernisierungsprojekt von zentralen Komponenten des .NET Frameworks unter verschiedenen Aspekten wie größerer Plattformunabhängigkeit, Open-Source-Entwicklung (Bereitstellung auf GitHub), verbesserter Modularität und vereinfachter Anwendungsentwicklung. Im November 2020 soll unter der Bezeichnung .NET 5.0 alle .NET Plattformen zusammengeführt werden, wobei .NET Core die technologische Basis ist.

Für die Ausführung von .NET-Programmen muss zwingend das .NET installiert sein, da die Programme nach dem Kompilieren in einer Zwischensprache (Intermediate Language IL) vorliegen, die vom Prozessor nicht verstanden wird. Während der Ausführung eines .NET-Programms übersetzt die Laufzeitumgebung (.NET Runtime) mit einem Just-In-Time-Compiler diese Zwischensprache in Maschinencode, der dann vom Prozessor ausgeführt werden kann. Quasi wie ein Interpreter bei z.B. Basic.

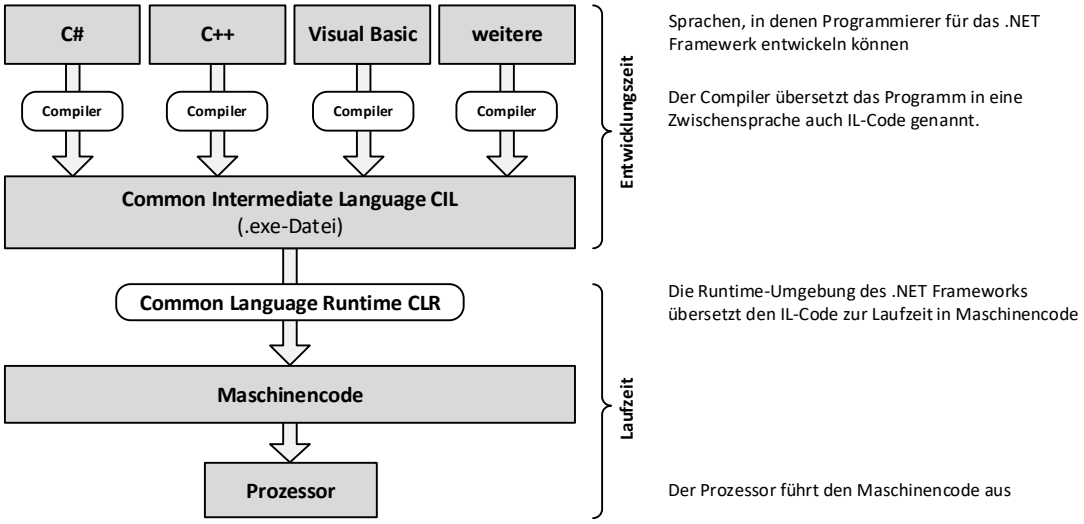

Abbildung 2: Grundprinzip von .NET

Da beim Kompilieren nicht Maschinencode für einen spezifischen Prozessor erzeugt wird, sind .NET-Applikationen relativ plattformunabhängig und können während der Laufzeit für den spezifischen Prozessor, auf dem sie gerade laufen, optimiert werden.

Wie in Abbildung 2 zu sehen ist, können diverse Programmiersprachen zur Entwicklung von .NET-Programmen verwendet werden. C# wurde jedoch speziell für .NET entwickelt und ist deshalb am besten geeignet, um alle Funktionen zu nutzen.

1.3.2 Objektorientierung

.NET ist zu 100 % objektbasiert. Sogar einfache Datentypen wie Integer werden als Objekte behandelt. Auch Zugriffe auf das darunterliegende Betriebssystem werden durch Klassen gekapselt.

1.3.3 Win32-API-Ersatz

Microsoft beabsichtigt längerfristig, die Anwendungsschnittstelle von Windows, die sogenannte Win32-API, welch nur in C oder Assembler geschrieben werden können, durch die Klassen des .NET zu ersetzen. Damit verwischen auch die typischen Merkmale der verschiedenen Programmiersprachen. Ob eine Anwendung mit Visual Basic, C# oder C++ programmiert wird, spielt keine Rolle mehr. Alle Sprachen greifen auf dieselbe Bibliothek zurück. Sprachspezifische Bibliotheken, die mit dem Betriebssystem interagieren, gibt es nicht mehr.

1.3.4 Sprachunabhängigkeit

Software Komponenten können, unabhängig davon, in welcher Programmiersprache sie entwickelt worden sind, in jeder anderen .NET-konformen Sprache verwendet werden. So können z.B. in C# geschriebene Klassen auch in Visual Basic oder C++ zum Einsatz kommen.

1.3.5 Speicherverwaltung

Die Freigabe von nicht mehr benötigtem Speicher war schon immer ein Problem. Unter .NET braucht sich der Entwickler nicht mehr darum zu kümmern, da ein Hintergrundprozess, der sogenannte Garbage Collector, nicht mehr benötigte Objekte automatisch löscht.

1.4 Microsoft Visual Studio

(Quelle: www.dev-insider.de)
Microsoft Visual Studio ist eine integrierte Entwicklungsumgebung, kurz IDE, für höhere Programmiersprachen. Integrierte Tools unterstützen die Entwicklung ASP.NET-Webanwendungen, Desktop-Anwendungen, XML-Webdiensten und mobilen Apps.

Die integrierte Entwicklungsumgebung (Integrated Development Environment) Visual Studio stammt von Microsoft und unterstützt zahlreiche Programmier-Hochsprachen. Hierzu zählen beispielsweise Visual Basic, C, C++, C#, SQL Server, PHP und Python. Darüber hinaus eignet sich Visual Studio auch für die Entwicklung mit Javascript, HTML und CSS.

In Visual Studio lassen sich somit native Win32-/Win64-Programme ebenso entwickeln wie Windows-Apps, dynamische Webservices bzw. Webseiten sowie Anwendungen für das .NET-Framework. Auch die Entwicklung und Implementierung mobiler Apps für Android, iOS und Windows Phone lässt sich mithilfe der durch die Microsoft-Tochtergesellschaft Xamarin bereitgestellten Tools bewerkstelligen.

1.4.1 Der Funktionsumfang von Visual Studio

Die Entwicklungsumgebung Visual Studio ist mit zahlreichen nützlichen Funktionen ausgestattet. Der Editor besitzt beispielsweise eine Online-Hilfe, die auf die aktuelle Position des Cursors reagiert. Außerdem werden im Quelltext Schlüsselwörter farblich hervorgehoben.

Neben einer automatischen Syntaxprüfung verfügt der Editor von Visual Studio auch über das Tool IntelliSense, welches Methoden und Funktionen schon während der Quelltexteingabe automatisch ergänzt. Darüber hinaus ist die Entwicklungsumgebung mit grafischen Schnittstellen zur Einbindung von .NET- und ActiveX-Bibliotheken sowie von Webservices ausgestattet.

1.4.2 Version 2019

(Quelle: Wikipedia)

Die Version Visual Studio 2019 (Interne Versionsnummer 16) für Windows und Mac ist seit dem 2. April 2019 erhältlich. Neuerungen sind u. a. ein überarbeiteter Dialog für das Starten neuer Projekte, KI-unterstützte Eingabehilfen ("Intellicode"), Verbesserungen bei Debugging und Refactoring sowie die Funktion "Live Share" zur Remote-Zusammenarbeit an gemeinsamem Code.

1.4.3 Community-Edition

Am 12. November 2014 erschien erstmals eine neue kostenlose Variante von Visual Studio 2013, die im Funktionsumfang weitgehend der Professional Edition entspricht. Sie heisst Community Edition und darf ebenso wie die Express Editions für kommerzielle Projekte verwendet werden, ist dabei aber beschränkt auf Unternehmen mit einem Jahresumsatz von maximal 1 Million US-Dollar und fünf Nutzer. Private Anwender, Schüler, Studenten und Bildungseinrichtungen dürfen diese Edition unbegrenzt verwenden, zudem ist auch die Entwicklung von Open-Source-Projekten damit unbegrenzt erlaubt. Auch für Visual Studio 2015, 2017 und 2019 ist diese Community Edition mit entsprechenden Lizenzeinschränkungen kostenlos verfügbar. Nach spätestens 30 Tagen müssen Sie sich registrieren.

Um den Funktionsumfang, welchen wir in der Schule nutzen zu installieren, reicht es während der Installation das oberste Kästchen „.NET-Desktopentwicklung“ anzuwählen.

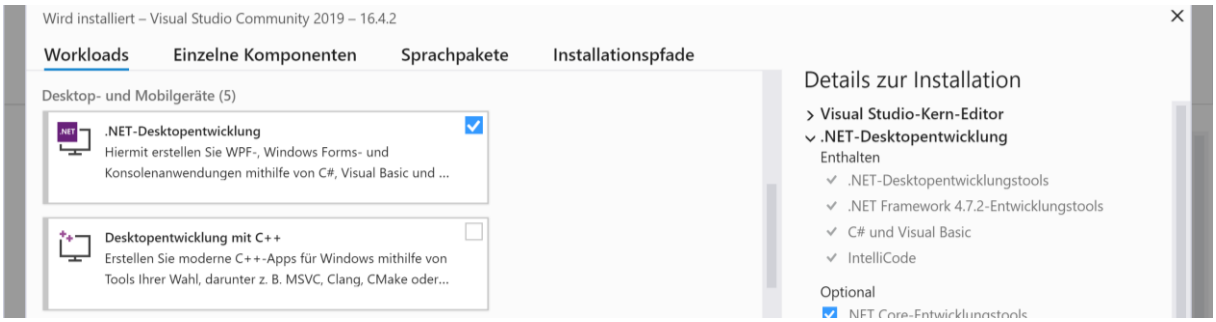

Abbildung 3: Installation Visual Studio

2 Grundlagen

2.1 Das erste Programm

Es ist üblich in einer neuen Programmiersprache als erstes ein Programm zu erstellen, das folgenden Text ausgibt: "Hello World!" Wir wollen dieser Tradition folgen.

2.1.1 Konsolen-Anwendung "Hello World!"

Öffnen Sie die Entwicklungsumgebung Visual Studio und befolgen Sie folgende Anweisungen, um Ihre erste Konsolenanwendung in C# zu erstellen:

1. Wählen Sie im Menü den Befehl *Datei / Neues Projekt*. Dann erscheint der Dialog *Neues Projekt erstellen*.

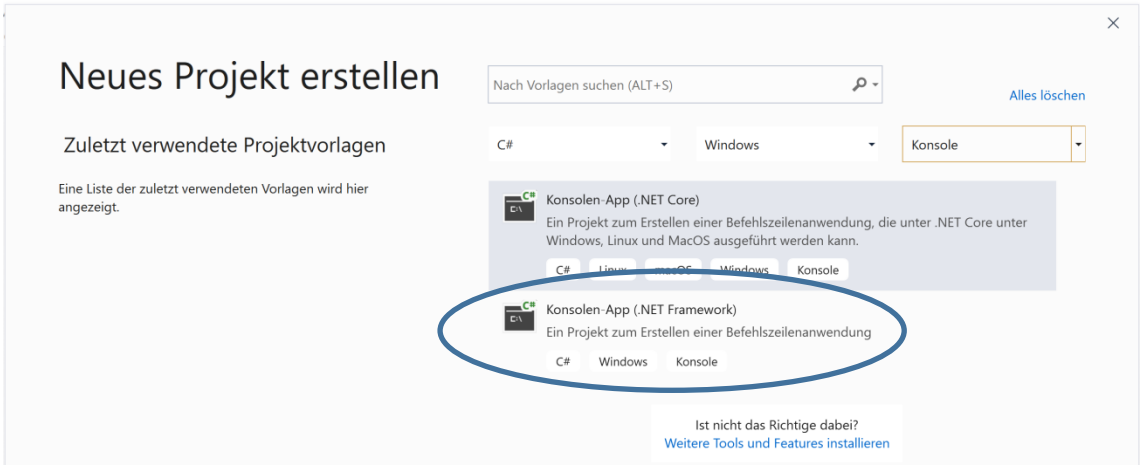

Abbildung 4: Dialog "Neues Projekt"

2. Wählen Sie die Projektvorlage *C# Konsolen-App (.NET Framework)*.

3. Geben Sie im Textfeld den Projektnamen "HelloWorld" ein und klicken Sie auf *OK*. Visual Studio erstellt dann das neue Projekt. Die zum Projekt gehörenden Dateien werden im *Projektmappen-Explorer* auf der rechten Seite angezeigt. Den Quelltext des Programms sehen Sie im Text-Editor auf der linken Seite. Die Anweisungen für Ihr Programm können Sie jetzt in der **`Main`**-Methode ab Zeile 12 eingeben.

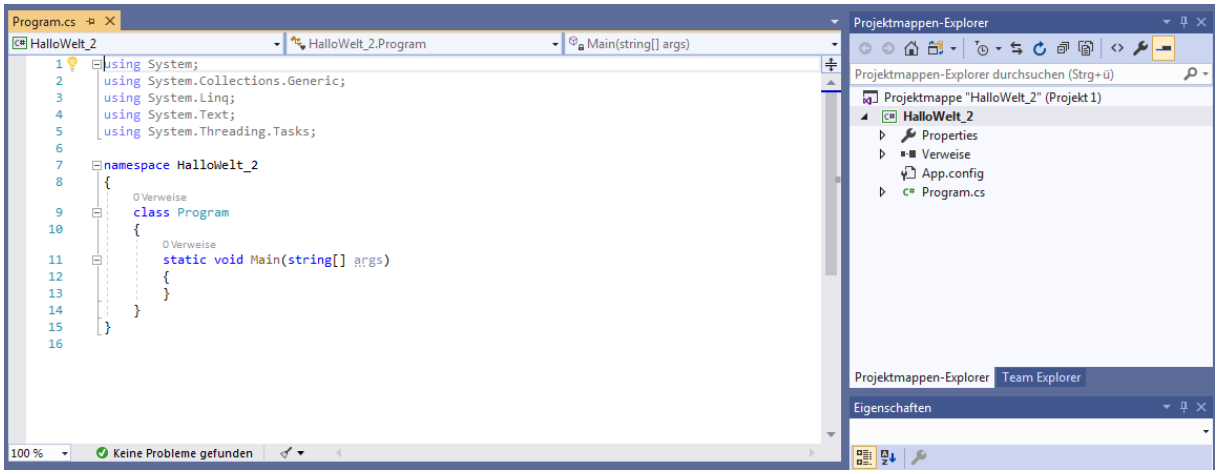

Abbildung 5: Grundgerüst einer Konsolen-Anwendung

4. Geben Sie in der **`Main`**-Methode den Text **`conso`** ein. Während dem Tippen erscheint unterhalb der Zeile ein kleines Fenster mit einer Auswahlliste. Dabei handelt es sich um das IntelliSense-Fenster, das ihre Eingaben mitverfolgt und passende Schlüsselwörter, Funktionen (in C# Methoden genannt) und Klassen vorschlägt. Zusätzlich wird rechts von der Auswahlliste in einem Tooltip die Funktion des aktuell gewählten Elements erklärt.

   So sehen wir, dass die Klasse **`Console`** zur Ein- und Ausgabe bei Konsolenanwendungen gedacht ist. Also genau das was wir benötigen. Drücken Sie die *Enter*-Taste, damit Ihre Eingabe **`conso`** durch **`Console`** ersetzt wird. Was genau eine Klasse ist, werden wir im Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** besprechen.

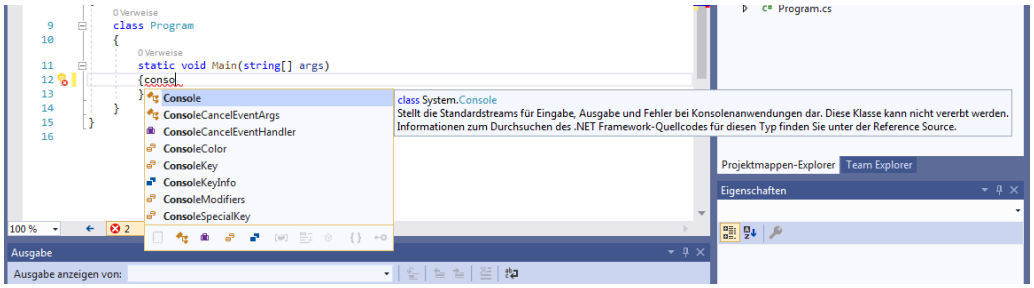

Abbildung 6: IntelliSense-Fenster

5. Geben Sie anschliessend einen Punkt ein. Dabei öffnet sich das IntelliSense-Fenster erneut und zeigt welche Funktionen die Klasse **`Console`** anbietet. Wählen Sie die Methode **`WriteLine`** im IntelliSense-Fenster aus, indem Sie den Methodennamen zu tippen beginnen oder mit den Pfeiltasten die Selektierung verschieben. Drücken Sie wiederum die *Enter*-Taste, um den Methoden-Aufruf ins Programm einzufügen.

   (Oder ganz kurz: Geben Sie „**CW**“ ein und drücken Sie zweimal die Tab-Taste. Das führt zu:

   `Console.WriteLine();`)

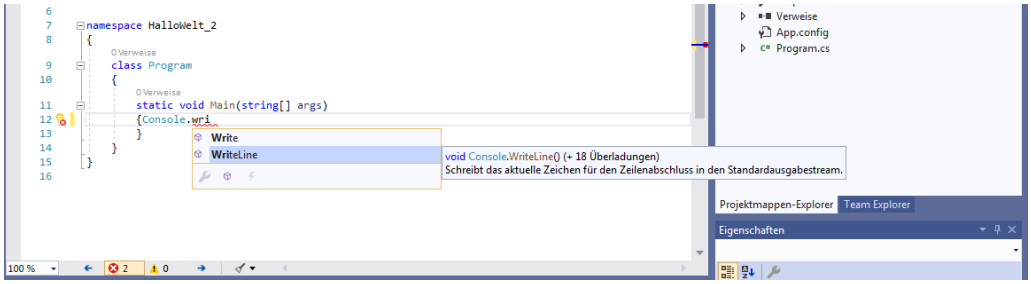

Abbildung 7: Methoden-Aufruf mit IntelliSense eingeben

6. Übergeben Sie der Methode **`WriteLine`** die Zeichenkette "Hello World!":

   `Console.WriteLine("Hello World!");`

7. Damit das Programm nach der Ausgabe des Texts nicht gleich wieder geschlossen wird, rufen wir noch die Methode **`ReadKey`** auf. Somit wartet das Programm auf eine Benutzereingabe bevor das Ende der Main-Funktion erreicht und das Programm geschlossen wird.

   `Console.ReadKey();`

8. Kompilieren und starten Sie das Programm mit dem Menübefehl *Debuggen / Debugging starten* oder mit der Taste *F5*. Dann erscheint ein Konsolenfenster mit der Ausgabe "Hello World!". Um das Programm wieder zu beenden, drücken Sie eine beliebige Taste.

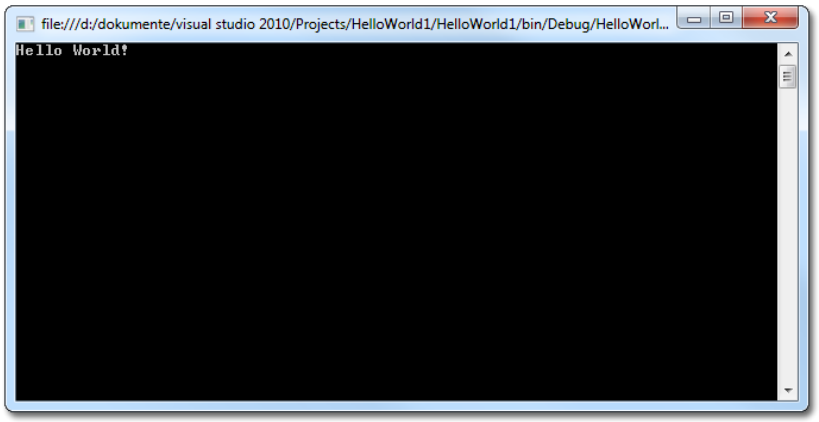

Abbildung 8: HelloWorld Programm

2.1.2 Programmstruktur

Auch wenn unser erstes Programm nicht viel macht, beinhaltet es bereits viele wichtige Elemente einer typischen .NET-Anwendung. Im folgenden Abschnitt werden wir die genaue Bedeutung dieser Elemente näher betrachten. Zur Übersicht zeigt Listing 1 noch einmal den gesamten Quelltext unserer *HelloWorld*-Anwendung.

```
01  using System;
02  using System.Collections.Generic;
03  using System.Linq;
04  using System.Text;
05
06  namespace HelloWorld
07  {
08      class Program
09      {
10          static void Main(string[] args)
11          {
12              Console.WriteLine("Hello World");
13              Console.ReadKey();
14          }
15      }
16  }
```

Listing 1: Programm HelloWorld

2.1.3 Namensräume und using-Direktiven

Die Funktionen der Klassenbibliotheken werden mit sogenannten Namensräumen hierarchisch zu funktionalen Gruppen zusammengefasst. Die Klasse **`Console`** befindet sich z.B. im Namensraum **`System`**. Die vier **`using`**-Direktiven am Beginn des Programms, binden oft verwendete Namensräume ein, damit diese bei Methoden-Aufrufen nicht immer angegeben werden müssen.

Die Anweisung **`Console.WriteLine`** bei Zeile 12 ist also nur eine Kurzform. Ohne die **`using`**-Direktive bei Zeile 1 müssten Sie für den Aufruf der Methode, neben dem Klassennamen auch den Namensraum angeben: **`System.Console.WriteLine`**

Die Entwicklungsumgebung hat auch für unsere Anwendung einen Namensraum (Zeile 6) angelegt. Standardmässig lautet dieser gleich wie der Projektname, kann aber ohne weiteres geändert werden.

2.1.4 Klasse Program

Da es sich beim .NET Framework um ein objektorientiertes, klassenbasiertes Framework handelt, sind alle Typen Klassen. Alle Methoden und Variablen müssen innerhalb von Klassen definiert werden. Dies gilt auch für unser kleines Beispielprogramm, das die Klasse **`Program`** definiert.

`Program` ist der Standardklassename, der die Entwicklungsumgebung beim Erzeugen einer Konsolenanwendung verwendet. Der Klassename hat keine besondere Bedeutung und kann auch geändert werden. Klassen werden mit dem Schlüsselwort **`class`** definiert, danach folgt der Klassenname und in geschweiften Klammern die eigentliche Funktionalität der Klasse.

2.1.5 Main-Methode

In der Klasse **`Program`** gibt es genau eine Methode mit dem Namen **`Main`**. Die **`Main`**-Methode ist der Einstiegspunkt von jedem C#-Programm und muss immer vorhanden sein. Sie wird beim Programmstart als erstes aufgerufen.

Die **`Main`**-Methode kann auch so definiert werden, dass sie eine Zahl an den Aufrufer zurückgibt. Sinnvoll ist dies beispielsweise, wenn Sie eine Konsolenanwendung aus einer Batch-Datei heraus starten und am Schluss das Resultat der Anwendung auswerten möchten. Um in einer Methode einen Wert zurückzugeben, wird in C# das Schlüsselwort **`return`** verwendet (siehe folgendes

Listimg). Dabei bedeutet der Rückgabewert 0 meist, dass das Programm erfolgreich ausgeführt wurde. Folglich wird im Fehlerfall ein Wert ungleich 0 zurückgegeben.

```
static int Main(string[] args)
{
   Console.WriteLine("Hello World");
   Console.ReadKey();
   return 0;
}
```

Listing 2: Main-Methode mit Rückgabewert

2.1.6 Befehlszeilenargumente

Im Parameter **`args`** der **`Main`**-Methode stehen die Befehlszeilenargumente, die Sie beim Start der Konsolenanwendung übergeben. Starten Sie die *HelloWorld*-Anwendung wie in folgendem Beispiel, befinden sich im Array **`args`** die Werte 5 und 10 als Zeichenkette.

```
HelloWorld.exe 5 10
```

Listing 3: Programmstart mit Befehlszeilenargumenten

Wenn sie ihre Anwendung in Visual Studio starten, können Sie auch Befehlszeilenparameter mitgeben. Öffnen Sie dazu im *Projektmappen-Explorer* das Kontext-Menü auf dem Projekt (in unserem Beispiel *HelloWorld*) und wählen Sie den Befehl *Eigenschaften*. Dann erscheinen auf der linken Seite die Projekteinstellungen. Im Register *Debuggen* können Sie dann im Textfeld *Befehlszeilenparameter* die gewünschten Werte eingeben. Danach werden diese Werte jedes Mal, wenn Sie ihre Anwendung aus der Entwicklungsumgebung starten, mitgegeben.

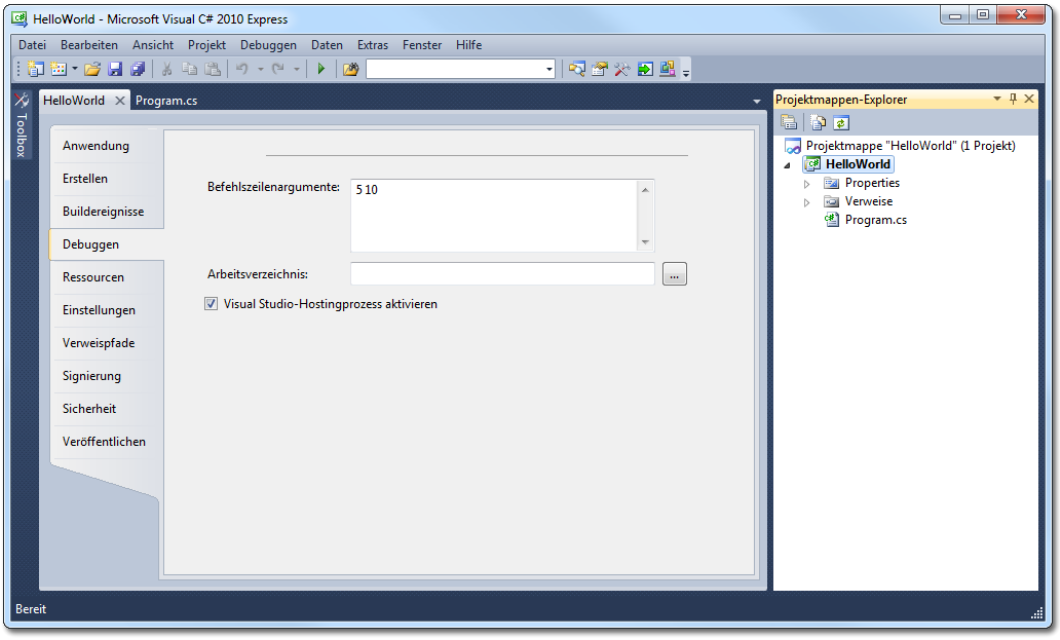

Abbildung 9: Befehlszeilenargumente in den Projekteinstellungen definieren

2.2 Syntax

2.2.1 Anweisungen

In C# werden Anweisungen analog wie in der Programmiersprache C mit einem Semikolon abgeschlossen. Wenn Sie das Semikolon vergessen, erhalten Sie einen Kompilierungsfehler.

```
Console.WriteLine("Hello World!");
Console.ReadKey();
;;;;;
```

Listing 4: Anweisungen

Auch wenn es sinnlos ist, dürfen Sie mehrere Semikolons hintereinanderschreiben, ohne dass explizit eine Anweisung dazwischenstehen muss.

Da ein Semikolon eine Anweisung eindeutig abschliesst, dürfen auch mehrere Anweisungen in eine Zeile geschrieben werden. Umgekehrt kann eine Anweisung auch problemlos auf mehrere Zeilen verteilt werden, ohne dass sich der Compiler daran stört.

```
// Mehrere Anweisungen auf einer Zeile
Console.Write("Hello "); Console.WriteLine("World!");

// Anweisung über mehrere Zeilen verteilt
Console.WriteLine(
   "Hello World!"
);
```

Listing 5: Beispiele für korrekte Anweisungen

2.2.2 Anweisungsblöcke

C#-Programmcode ist blockorientiert, das heisst Anweisungen werden grundsätzlich immer innerhalb geschweifter Klammern geschrieben. Jeder Block kann eine beliebige Anzahl von Anweisungen enthalten. Somit hat ein Anweisungsblock allgemein die folgende Form:

```
{
   Anweisung1;
   Anweisung2;
   ...
}
```

Listing 6: Anweisungsblock

Anweisungsblöcke lassen sich beliebig ineinander verschachteln:

```
{
   Anweisung1;
   {
      Anweisung2;
      Anweisung3;
   }
   Anweisung4;
}
```

Listing 7: Verschachtelung von Anweisungsblöcken

Um die Lesbarkeit des Quelltexts zu erhöhen, wird die Zugehörigkeit einer oder mehrerer Anweisungen zu einem bestimmten Block normalerweise mit Einzügen optisch angedeutet. Im Text-Editor von Visual C# Express können Sie mehrere Zeilen auf einmal einrücken. Markieren Sie dazu die entsprechenden Zeilen und drücken Sie die *Tab*-Taste. Um die Einrückung mehrerer Zeilen rückgängig zu machen, können Sie die Tastenkombination *Shift* + *Tab* verwenden.

Variablen, die innerhalb eines Blockes definiert werden, hören auf zu existieren, wenn der Block verlassen wird. Sie haben nur innerhalb des Blockes ab der Definition Gültigkeit. Normalerweise sind

Anweisungsblöcke Teil einer Methode, Bedingung oder Schleife, können aber auch für sich alleine zur Strukturierung des Quelltexts verwendet werden.

2.2.3 Kommentare

In C# kann der Quelltext auf zwei verschiedene Arten kommentiert werden. Die am häufigsten benutzte Variante ist der Zeilenkommentar, der mit zwei Schrägstrichen **//** eingeleitet wird. Ein Zeilenkommentar gilt für den Rest der gesamten Codezeile, kann jedes beliebige Zeichen enthalten und darf auch nach einer abgeschlossenen Anweisung stehen.

```
// Zeilenkommentar
Console.WriteLine("Hello World!"); // Konsolenausgabe
```

Listing 8: Zeilenkommentare

Blockkommentare werden mit der Zeichenfolge **/*** eingeleitet und mit ***/** beendet. Sie können sich über mehrere Zeilen erstrecken. Blockkommentare dürfen nicht verschachtelt werden.

```
/* Console.WriteLine("Hello World!");
Console.ReadKey(); */

Console.WriteLine/* Kommentar */("Hello World!");
```

Listing 9: Blockkommentare

Die Entwicklungsumgebung Visual C# Express bietet eine interessante Alternative, um grössere Blöcke auf einmal aus- bzw. einzukommentieren. Sie müssen dazu nur sicherstellen, dass die Symbolleiste *Text-Editor* angezeigt wird (Menübefehl: *Ansicht / Symbolleisten / Text-Editor*). Die Symbolleiste enthält zwei Schaltflächen, um markierte Codeblöcke auszukommentieren oder eine Kommentierung wieder aufzuheben (siehe Abbildung 10).

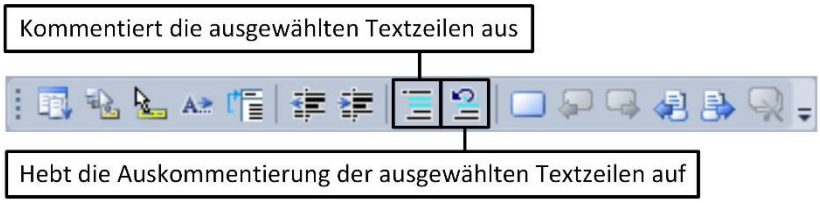

Abbildung 10: Kommentare mit der Symbolleiste Fehler! Verweisquelle konnte nicht gefunden werden.**]**

C# kennt noch eine weitere Variante von Kommentaren. Die sogenannten XML-Kommentare werden mit drei Schrägstrichen **///** eingeleitet und dienen zum Erstellen einer XML-basierten Dokumentation, im Stil der Online-Hilfe des .NET Frameworks.

2.2.4 Gross- und Kleinschreibung

C# gehört zu der Gruppe von Programmiersprachen, die zwischen Gross- und Kleinschreibung unterscheiden. Somit wird die erste Zeile im folgenden Listing zu einem Kompilierungsfehler führen.

```
Console.Writeline("Hello World!");  // Falsch
Console.WriteLine("Hello World!");  // Richtig
```

Listing 10: Gross- und Kleinschreibung

Eine weitere Folge davon ist, dass zwei gleichlautende Bezeichner (z.B. Variablennamen), die sich nur durch Gross- und Kleinschreibung unterscheiden, in C# auch für zwei unterschiedliche Programmelemente stehen.

2.3 Variablen

2.3.1 Datentypen

C# ist eine stark typisierte Programmiersprache. Das heisst jeder Variablen muss ein Datentyp zugewiesen werden. In Tabelle 1 sind alle nativen Datentypen von C# aufgeführt.

Tabelle 1: Datentypen Fehler! Verweisquelle konnte nicht gefunden werden.**],** Fehler! Verweisquelle konnte nicht gefunden werden.**]**

C#-Datentyp	.NET-Datentyp	Speicherbedarf (Bits/Bytes)	Dezimalstellen	Wertebereich
`sbyte`	`System.SByte`	8 / 1	-	-128 bis 127
`byte`	`System.Byte`	8 / 1	-	0 bis 255
`short`	`System.Int16`	16 / 2	-	-32'768 bis 32'767
`ushort`	`System.UInt16`	16 / 2	-	0 bis 65'535
`int`	`System.Int32`	32 / 4	-	-2'147'483'648 bis 2'147'483'647
`uint`	`System.UInt32`	32 / 4	-	0 bis 4'294'967'295
`long`	`System.Int64`	64 / 8	-	-2^{63} bis 2^{63} - 1
`ulong`	`System.UInt64`	64 / 8	-	0 bis 2^{64} - 1
`char`	`System.Char`	16 / 2	-	0 bis 65'535
`float`	`System.Single`	32 / 4	7	1.5×10^{-45} bis 3.4×10^{38}
`double`	`System.Double`	64 / 8	15 bis 16	5.0×10^{-324} bis 1.7×10^{308}
`decimal`	`System.Decimal`	128 / 16	28 bis 29	1.0×10^{-28} bis 7.9×10^{28}
`bool`	`System.Boolean`	8 / 1	-	`true` oder `false`
`string`	`System.String`	variabel	-	ca. 2^{31} Unicode-Zeichen
`object`	`System.Object`	variabel	-	Kann jeden Datentyp enthalten

Wie wir bereits in der Einleitung gesehen haben, ist .NET nicht nur plattform- sondern auch sprachunabhängig. Um dies zu erreichen, gibt das Framework eine Reihe von Datentypen vor, die eine .NET-kompatible Programmiersprache implementieren muss. In obenstehender Tabelle sehen Sie welcher .NET-Datentyp den verschiedenen C#-Datentypen zugrunde liegt. Bei der Programmierung spielt es keine Rolle, ob sie den C#- oder .NET-Namen des Datentyps verwenden.

2.3.2 Variablendeklaration

Bevor Sie Variablen verwenden können, müssen Sie diese deklarieren. In C# können Variablen im Gegensatz zur Programmiersprache C an einer beliebigen Stelle im Programm deklariert werden. Es wird unterschieden zwischen expliziter und impliziter Deklaration. Bei der expliziten Deklaration legen Sie den Datentyp und den Namen der Variablen fest. In Listing 11 sind ein paar Beispiele für mögliche Variablendeklarationen zu sehen.

```
// Einfache Variablendeklarationen
int operand;
float pi;
char a;
string text;

// Die Variable bei der Deklaration mit einem Wert initialisieren
int operand = 125;
float pi = 3.14f;
char a = 'a';
string text = "Hello World!";

// Mehrere Variablen vom selben Typ auf einer Zeile deklarieren
int x, y, z;

// Mehrere Variablen vom selben Typ auf einer Zeile deklarieren und initialisieren
int x, y = 2, z;
```

Listing 11: Deklaration mit expliziter Typangabe

Variablen können, wie in Listing 11 zu sehen ist, bei der Deklaration gleich mit einem Wert initialisiert werden. Der Initialisierungswert wird vor der Zuweisung zunächst im Speicher abgelegt. Dabei muss der Compiler den Datentyp automatisch bestimmen. Bei Dezimalzahlen wählt er immer den Typ **`double`**. Dies führt jedoch z.B. bei der Initialisierung von **`float`**-Variablen zu Problemen, da der Typ **`double`** grösser als **`float`** ist. Deshalb kann der Programmierer dem Compiler mit einem Suffix einen Hinweis geben, um welchen Fliesskommatyp es sich beim Initialisierungswert handelt. Bei der Initialisierung der Variable **`pi`** in Listing 11 wird aus diesem Grund das Suffix **`f`** hinten angehängt.

Tabelle 2: Typsuffix für Fliesskommatypen Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Suffix	Fliesskommatyp
`F oder f`	`float`
`D oder d`	`double`
`M oder m`	`decimal`

Wenn Sie den genauen Datentyp nicht kennen, können Sie eine Variable auch mit dem Typ **`var`** implizit deklarieren. Der Compiler leitet in diesem Fall den Datentyp der Variable aus dem zugewiesenen Wert ab. Deshalb müssen implizit deklarierte Variablen bei der Deklaration zwingend initialisiert werden. Pro Zeile kann immer nur eine implizite Deklaration ausgeführt werden.

```
var operand1 = 125;          // deklariert eine Variable des Typs int
var pi = 3.14;               // deklariert eine Variable des Typs double
var a = 'a';                 // deklariert eine Variable des Typs char
var text = "Hello World!";   // deklariert eine Variable des Typs string

var operand2;                // Fehler, da die Initialisierung fehlt
```

Listing 12: Deklaration mit impliziter Typangabe

2.3.3 Variablennamen

Bei der Vergabe von Variablennamen müssen folgende Regeln eingehalten werden.

- Variablennamen müssen mit einem Buchstaben oder Unterstrich beginnen
- Variablennamen dürfen Buchstaben, Ziffern und den Unterstrich enthalten
- Leerzeichen sind in Variablennamen nicht erlaubt
- Der Compiler unterscheidet bei Variablennamen die Gross- und Kleinschreibung
- Variablennamen müssen eindeutig sein. Sie dürfen nicht gleich lauten wie eine andere Variable, ein Schlüsselwort, eine Methode oder eine Klasse

Listing 13 zeigt einige Beispiele von korrekten und fehlerhaften Variablennamen.

```
// Korrekte Variablennamen
long myVar;
long MyVar;
byte result_12;
int _carColor;

// Fehlerhafte Variablennamen
int 34M;
string message Text;
long salary%Tom;
```

Listing 13: Korrekte und fehlerhafte Variablennamen

Grundsätzlich können Sie Variablennamen frei wählen, solange Sie die oben aufgestellten Regeln beachten. Allerdings halten sich die meisten Programmierer zudem an die von Microsoft veröffentlichten Entwurfsrichtlinien für .NET. Darin wird definiert, dass für Variablennamen die sogenannte Camel-Case-Schreibweise verwendet werden soll. Dabei beginnt der Name der Variable immer mit einem Kleinbuchstaben und alle weiteren Wörter, die im Namen vorkommen, mit einem Grossbuchstaben. Wir werden uns in den folgenden Beispielen auch an diese Regel halten.

```
int measureValue;
string errorText;
```

Listing 14: Beispiele für die Camel-Case-Schreibweise

2.3.4 Typkonvertierung

Immer dann, wenn bei einer Operation zwei unterschiedliche Datentypen im Spiel sind, muss der Typ, der rechts vom Zuweisungsoperator steht, in den Typ umgewandelt werden, der sich auf der linken Seite befindet. Prinzipiell wird zwischen impliziter und expliziter Konvertierung unterschieden.

Implizite Konvertierung

Erfolgt eine Umwandlung in einen Datentyp mit einem grösseren Fassungsvermögen als der Ursprungstyp, spricht man von einer impliziten Konvertierung. Eine solche Konvertierung nimmt der Compiler automatisch vor ohne spezielle Angaben durch den Programmierer.

```
int intVar = 32;
double doubleVar = intVar;   // Korrekte implizite Konvertierung
short shortVar = intVar;     // Fehler: Zieldatentyp ist kleiner als der Ursprungstyp
```

Listing 15: Implizite Konvertierung

In Abbildung 11 ist eine Übersicht der erlaubten, impliziten Konvertierungen zu sehen. Die Pfeilrichtung zeigt immer eine Konvertierung in einen grösseren Datentyp an. Demzufolge wird ein **`byte`** problemlos implizit in einen **`short`**, **`int`**, **`long`** usw. konvertiert. Umgekehrt ist die implizite Konvertierung z.B. von **`int`** in **`byte`** nicht möglich. Beachten Sie zudem, dass es keine impliziten Konvertierungen zwischen den Gleitkommatypen **`float`** bzw. **`double`** und **`decimal`** gibt.

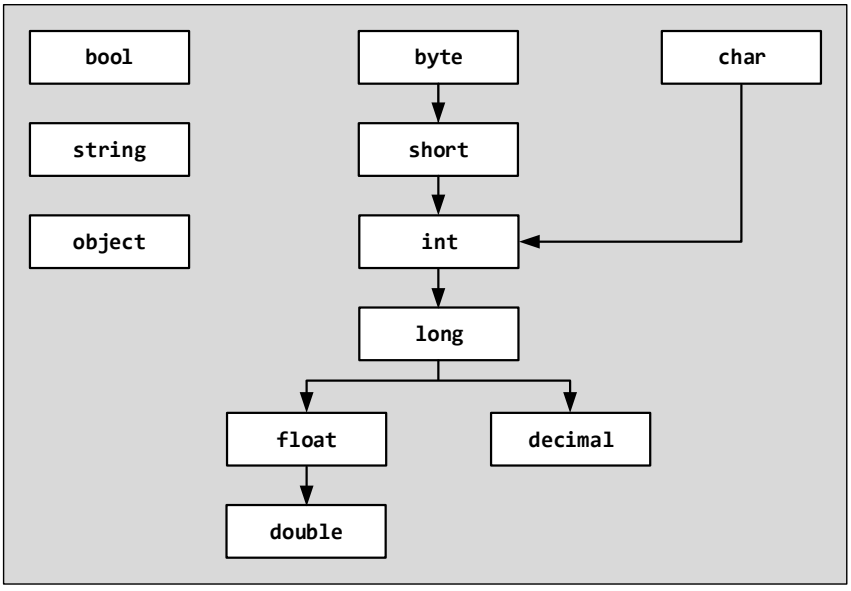

Abbildung 11: Implizite Konvertierung einfacher Datentypen Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Eine besondere Stellung nehmen die Typen **`bool`**, **`string`**, **`char`** und **`object`** ein. Mit einem **`bool`** oder einem **`string`** sind keine impliziten Konvertierungen möglich, ein **`char`** kann mit Ausnahme von **`byte`** und **`short`** jedem anderen Typ zugewiesen werden. Variablen vom Typ **`object`** unterliegen anderen Regeln, die wir ab Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** besprechen werden.

Explizite Konvertierung

Unter expliziter Konvertierung versteht man die ausdrückliche Anweisung an den Compiler, den Wert eines bestimmten Datentyps in einen anderen umzuwandeln. Die explizite Konvertierung hat eine

sehr einfache Syntax. Vor dem zu konvertierenden Ausdruck wird in runden Klammern der Zieldatentyp der Konvertierung angegeben:

```
int intVar = 32;
short shortVar = (short)intVar;
```

Listing 16: Explizite Konvertierung

Bei expliziten Konvertierungen erlaubt der Compiler auch Umwandlungen von einem grösseren in einen kleineren Datentyp. Dabei werden die höherwertigen Bits, die nicht in den Zieldatentyp passen, verworfen. Das bedeutet, wenn der zu konvertierende Ausdruck einen Wert hat, der grösser ist als der Maximalwert des Zieldatentyps, kommt es bei der Konvertierung zu einem Überlauf und der Wert wird verändert (siehe Listing 17).

```
int intVar = 100000;
double doubleVar = intVar;         // doubleVar = 100000
ushort ushortVar = (ushort)intVar; // ushortVar = 34464
```

Listing 17: Überlauf bei der expliziten Konvertierung

Dieses Verhalten kann zu sehr schwer zu lokalisierbaren Fehlern in einer laufenden Anwendung führen. Deshalb sollten Sie explizite Konvertierungen möglichst sparsam verwenden.

Noch besser ist es, wenn Sie die Konvertierungsmetoden der Klasse **`Convert`** aus dem Namensbereich **`System`** verwenden. Kommt es bei einer Konvertierung zu einem Überlauf, melden dies die Methoden mit einer **`OverflowException`**.

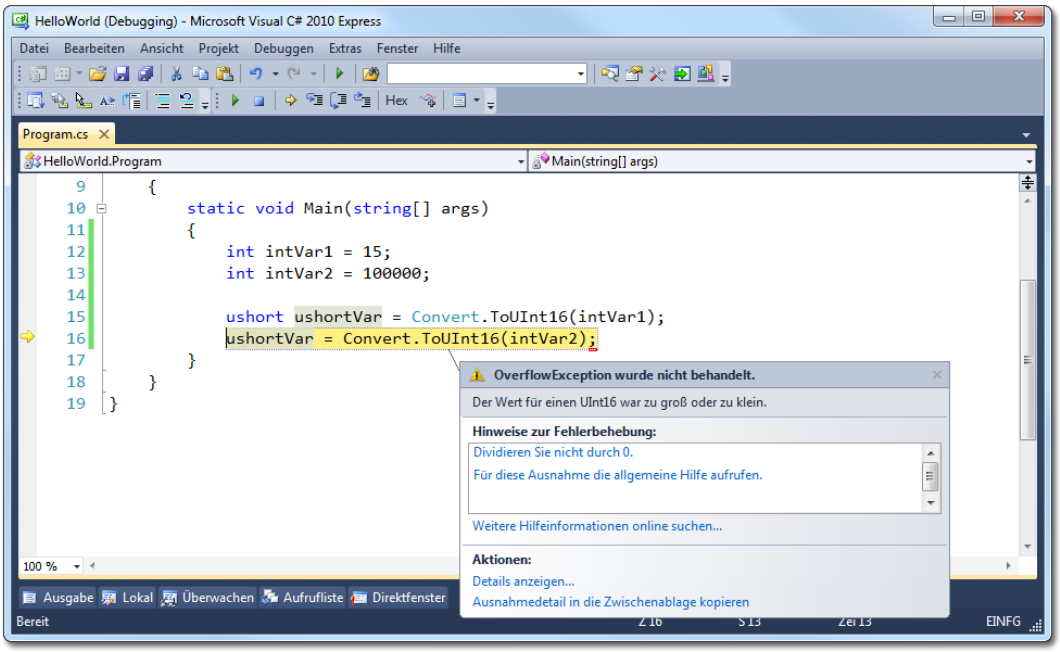

Abbildung 12: Bei Konvertierungen mit der Klasse Convert werden Überläufe zur Laufzeit erkannt

Dabei wird das Programm an der Stelle des Überlaufs angehalten und ein entsprechender Hinweis angezeigt (siehe oben). Was genau eine **`Exception`** ist und wie man auf diese reagiert, werden wir in Kapitel 2.9 kennenlernen. In der IntelliSense-Auswahlliste werden Sie sehen, dass die Klasse **`Convert`** für alle nativen Datentypen von C# eine Konvertierungsmethode anbietet.

Text in Zahlen umwandeln

Um eine Zahl, die als Zeichenkette vorliegt, in einen Zahlentyp umzuwandeln, können Sie neben der **`Convert`**-Klasse auch die **`Parse`**-Methode des entsprechenden Datentyps verwenden. Falls sich ungültige Zeichen (z.B. Buchstaben) in der Zeichenkette befinden, wird das Programm mit einer **`FormatException`** angehalten. Wenn der Zahlenwert in der Zeichenkette zu gross für den Datentyp ist, gibt es eine **`OverflowException`**.

```
string intStr = "5", doubleStr = "4.5";
int intValue = int.Parse(intStr);
double doubleValue = double.Parse(doubleStr);
```

Listing 18: Text in Zahl umwandeln

Zahlen in Text umwandeln

Alle Datentypen, die eine Zahl darstellen, besitzen eine Methode mit den Namen **`ToString`**, mit der die Zahl in eine Zeichenkette umgewandelt werden kann.

```
int intValue = 55;
string str = intValue.ToString();
```

Listing 19: Zahl in Text umwandeln

2.4 Operatoren

2.4.1 Arithmetische Operatoren

Neben den üblichen Operatoren für die vier Grundrechenarten Addition, Subtraktion, Division und Multiplikation kennt C# noch drei weitere Operatoren mit besonderer Bedeutung:

Tabelle 3: Arithmetische Operatoren Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Operator	Beschreibung	Beispiel
`+ - * /`	Addition, Subtraktion, Multiplikation, Division	`a = b + c`
`%`	Modulo-Operator, liefert den Restwert einer Division	`a = b % c`
`++`	Inkrement-Operator, erhöht den Inhalt des Operanden um 1.	`++a // a = a + 1`
`--`	Dekrement-Operator, verringert den Inhalt des Operanden um 1.	`--a // a = a - 1`

Multiplikationen und Divisionen haben einen höheren Vorrang als Additionen und Subtraktionen und werden somit zuerst ausgeführt. Wünschen Sie ein anderes Verhalten, müssen Sie runde Klammern einsetzen um die Ausführungsreihenfolge zu beeinflussen.

```
int result1 = 10 – 5 * 2;       // Wert von result1 ist 0
int result2 = (10 – 5) * 2;     // Wert von result2 ist 10
int result3 = 10 – 4 / 2;       // Wert von result3 ist 8
int result4 = (10 – 4) / 2;     // Wert von result4 ist 3
```

Listing 20: Ausführungsreihenfolge mit Klammern beeinflussen

Die Division zweier ganzer Zahlen kann, wie in Listing 21 zu sehen ist, falsche Resultate liefern. Das liegt daran, dass das Resultat vor der Zuweisung an die Variable noch zwischengespeichert wird. Dazu wird so viel Speicher reserviert, wie der grösste Typ der beiden beteiligten Operanden benötigt. In unserem Fall wäre das die Speichergrösse für eine ganze Zahl. Deshalb wird der Dezimalteil des Ergebnisses abgeschnitten. Zur Lösung dieser Problematik müssen Sie jeweils sicherstellen, dass einer der beiden Operanden als Dezimalzahl erkannt wird.

```
double w = 3 / 4;           // Falsch:  w = 0
double x = 3.0 / 4;         // Korrekt: x = 0.75
double y = 3f / 4;          // Korrekt: y = 0.75
double z = (double)3 / 4    // Korrekt: z = 0.75
```

Listing 21: Besonderheiten bei der Division

Wie in Listing 22 zu sehen ist, spielt die Lage des Inkrement-Operators eine Rolle. Das Ergebnis der Operation **++a** (Präfix) ist der Wert des Operanden nach der Erhöhung. Umgekehrt erhalten wir bei der Operation **a++** (Postfix) den Wert des Operanden vor der Erhöhung. Dasselbe Verhalten gilt auch für den Dekrement-Operator.

```
// Präfix-Operation
int a = 5;
int b = ++a; // b = 6

// Postfix-Operation
int a = 5;
int b = a++; // b = 5
```

Listing 22: Unterschied zwischen Präfix- und Postfix-Operationen

2.4.2 Vergleichsoperatoren

Vergleichsoperatoren vergleichen zwei Ausdrücke miteinander. Das Ergebnis ist immer ein boolscher Wert, also entweder **`true`** oder **`false`**.

Tabelle 4: Vergleichsoperatoren Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Operator	Beschreibung	Beispiel
`==`	Das Ergebnis der Operation ist **`true`**, wenn **a** gleich **b** ist.	`if (a == b)`
`!=`	Das Ergebnis der Operation ist **`true`**, wenn **a** ungleich **b** ist.	`if (a != b)`
`>`	Das Ergebnis der Operation ist **`true`**, wenn **a** grösser **b** ist.	`if (a > b)`
`<`	Das Ergebnis der Operation ist **`true`**, wenn **a** kleiner **b** ist.	`if (a < b)`
`<=`	Das Ergebnis der Operation ist **`true`**, wenn **a** kleiner oder gleich **b** ist.	`if (a <= b)`
`>=`	Das Ergebnis der Operation ist **`true`**, wenn **a** grösser oder gleich **b** ist.	`if (a >= b)`

2.4.3 Logische Operatoren

Logische Operatoren liefern als Resultat ebenfalls einen boolschen Wert.

Tabelle 5: Logische Operatoren Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Operator	Beschreibung	Beispiel
`!`	Logisches NOT: Gibt **`true`** zurück, wenn der Ausdruck **`false`** ist.	`if (!(a > b))`
`&&`	Logisches AND: Der Ausdruck **a && b** ist **`true`**, wenn sowohl **a** als auch **b** **`true`** sind. Zuerst wird **a** ausgewertet. Sollte **a** **`false`** sein, ist in jedem Fall der Gesamtausdruck unabhängig von **b** **`false`**. Deshalb wird **b** nicht mehr ausgewertet.	`if ((a > b) && (b < c))`
`||`	Logisches OR: Der Ausdruck **a || b** ist **`true`**, wenn entweder **a** oder **b** **`true`** ist. Zuerst wird **a** ausgewertet. Sollte **a** **`true`** sein, ist in jedem Fall der Gesamtausdruck unabhängig von **b** **`true`**. Deshalb wird **b** nicht mehr ausgewertet.	`if ((a > b) || (b < c))`

2.4.4 Bitweise Operatoren

Mit den bitweisen Operatoren können Sie einzelne Bits von Operanden abfragen oder manipulieren.

Tabelle 6: Bitweise Operatoren Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Operator	Beschreibung	Beispiel
`~`	Binäres NOT (bitweise) (Einerkomplement)	`a = ~c;`
`|`	Binäres OR (bitweise)	`a = b | c`
`&`	Binäres AND (bitweise)	`a = b & c`
`^`	Binäres XOR (bitweise)	`a = b ^ c`
`<<`	Binäres Linksschieben	`a = b << c; //= b * 2^c`
`>>`	Binäres Rechtsschieben	`a = b >> c; //= b / 2^c`

2.4.5 Zuweisungsoperatoren

Kombinierte Zuweisungsoperatoren machen den Code eher unübersichtlicher. Der Vollständigkeit halber, werden sie hier aber trotzdem erwähnt. Bis auf die Ausnahme des einfachen Gleichheitszeichens dienen alle anderen Zuweisungsoperatoren zur verkürzten Schreibweise einer Anweisung, dabei ist der linke Operand einer Operation gleichzeitig auch der Empfänger des Ergebnisses.

Tabelle 7: Zuweisungsoperatoren Fehler! Verweisquelle konnte nicht gefunden werden.**]**

Operator	Beschreibung	Beispiel
`=`	Zuweisungsoperator	`a = b;`
`+=`	Kombinierter Zuweisungsoperator	`a += b // a = a + b`
`-=`	Kombinierter Zuweisungsoperator	`a -= b // a = a - b`
`*=`	Kombinierter Zuweisungsoperator	`a *= b // a = a * b`
`/=`	Kombinierter Zuweisungsoperator	`a /= b // a = a / b`
`%=`	Kombinierter Zuweisungsoperator	`a %= b // a = a % b`
`&=`	Kombinierter Zuweisungsoperator	`a &= b // a = a & b`
`|=`	Kombinierter Zuweisungsoperator	`a |= b // a = a | b`
`^=`	Kombinierter Zuweisungsoperator	`a ^= b // a = a ^ b`

2.5 Ein- und Ausgabe

Bei unserem ersten Programm haben wir die Klasse **`Console`** bereits kurz kennengelernt. In diesem Kapitel betrachten wir ihre Funktionen zur Ein- und Ausgabe bei Konsolenanwendungen noch etwas genauer.

2.5.1 Write und WriteLine

Die Methode **`WriteLine`** kann nicht nur Texte, sondern alle nativen Datentypen von C# ausgeben.

```
int intVal = 125;
float floatVal = 3.1415;
bool boolVal = true;

Console.WriteLine(intVal);
Console.WriteLine(floatVal);
Console.WriteLine(boolVal);
```

Listing 23: Ausgabe mit WriteLine

Wenn Sie bei der Ausgabe Zahlenwerte mit Text kombinieren möchten, gibt es zwei Möglichkeiten. Entweder sie verketten mit Hilfe des **+** Operators einzelne Textteile mit den Zahlenwerten oder Sie verwenden die sogenannte kombinierte Formatierung von .NET.

Bei der kombinierten Formatierung fügen Sie in der Zeichenkette an den Stellen, wo Sie die Zahlenwerte einsetzen möchten, einen Platzhalter ein. Ein Platzhalter besteht aus zwei geschweiften Klammern mit einer Nummer dazwischen. Beim Aufruf von **`WriteLine`** geben Sie dann die Zahlenwerte als zusätzliche Parameter mit. Die **`WriteLine`**-Methode setzt dann diese Werte vor der Ausgabe an den richtigen Stellen in der Zeichenkette ein. Der erste Parameter wird beim Platzhalter mit der Nummer 0 eingesetzt, der zweite beim Platzhalter mit der Nummer 1 usw.

```
double distance1 = 5.126;
double distance2 = 3.258;

// String mit + Operator zusammensetzen
Console.WriteLine("Die Distanzen betragen " + distance1 + " und " + distance2 + " km.");

// String mit Kombinierter Formatierung von .NET zusammensetzen
Console.WriteLine("Die Distanzen betragen {0} und {1} km.", distance1, distance2);
```

Listing 24: Zahlenwerte mit Text kombinieren

Mit Hilfe der kombinierten Formatierung können die einzusetzenden Werte auch gleich formatiert werden. Dazu fügen Sie beim entsprechenden Platzhalter nach der Parameternummer zusätzlich ein Doppelpunkt und eine Formatierungsanweisung ein. Möchten Sie z.B. eine Dezimalzahl auf zwei Kommastellen runden, würde der Platzhalter wie in Listing 25 aussehen. Eine Liste mit allen möglichen Formatierungsanweisungen finden Sie am Ende des Teilkapitels.

```
double distance = 3.258;
Console.WriteLine("Distanz gerundet: {0:0.00}", distance);    // Ausgabe: Distanz gerundet: 3.26
```

Listing 25: Werte beim Einsetzen runden

Die kombinierte Formatierung wird nicht nur von der **`WriteLine`**-Methode, sondern auch von der **`Fromat`**-Methode des Datentyps **`string`** unterstützt.

```
double distance = 3.258;
string text = string.Format("Distanz gerundet: {0:0.00}", distance);
```

Listing 26: Kombinierte Formatierung mit der Methode Format

Die **`Write`**-Methode fügt im Gegensatz zu **`WriteLine`** keinen Zeilenumbruch am Ende der Ausgabe ein. Sonst gibt es keinen Unterschied zwischen den beiden Methoden.

Sie können Zeichenketten auch mit sogenannten Escape-Zeichen formatieren. Alle diese Sonderzeichen werden mit einem umgekehrten Schrägstrich (Backslash) eingeleitet. Möchten Sie z.B. mitten in einer Zeichenkette einen Zeilenumbruch haben, können Sie das Escape-Zeichen **\n** einfügen. Tabelle 8 zeigt eine Zusammenfassung der wichtigsten Escape-Zeichen.

Tabelle 8: Escape-Zeichen

Zeichen	Bedeutung
\"	Doppeltes Anführungszeichen
****	Umgekehrter Schrägstrich (Backslash)
\a	Gibt ein akustisches Signal aus (Warnsignal)
\b	Rückschritt
\f	Vorschub
\n	Neue Zeile
\r	Wagenrücklaufzeichen
\t	Tabulator

In Listing 27 finden Sie ein paar Beispiele für die Verwendung von Escape-Zeichen.

```
// Ausgabe:
// Die Datei "Hallo.txt" befindet sich in
// C:\Folder\SubFolder

Console.Write("Die Datei \"Hallo.txt\" befindet sich in\nC:\\Folder\\SubFolder");
```

Listing 27: Escape-Zeichen

2.5.2 Read und ReadLine

Wenn Sie die **ReadLine**-Methode aufrufen, hält das Programm an und wartet auf Benutzereingaben. Mit der *Enter*-Taste wird die Eingabe abgeschlossen und die Methode liefert die eingegebenen Zeichen als **string** zurück. Wenn Sie einen Zahlenwert einlesen möchten, können Sie die eingelesene Zeichenkette anschliessend mit der **Convert**-Klasse oder mit der **Parse**-Methode des Zieldatentyps konvertieren.

```
int radius;

Console.Write("Geben Sie den Kreisradius ein: ");
radius = int.Parse(Console.ReadLine());
```

Listing 28: Zahlenwert einlesen

Neben der **ReadLine**-Methode gibt es auch noch die **Read**-Methode. Beim Aufruf dieser Methode hält das Programm auch an und wartet auf Benutzereingaben, bis die Eingabe mit *Enter* abgeschlossen wird. Allerdings liefert die **Read**-Methode nicht die gesamte Eingabe auf einmal zurück, sondern liest Zeichen für Zeichen aus dem Eingabebuffer. Erst wenn alle Zeichen aus dem Buffer ausgelesen worden sind, ist die **Read**-Methode wieder blockierend und leitet die nächste Benutzereingabe ein. Der Rückgabewert der **Read**-Methode ist der ASCII-Wert des jeweiligen Zeichens.

Strings formatieren

Tabelle 9: Zahlen (value = 1000000)

Symbol	Typ	Aufruf	Ergebnis
c	Währung (currency)	**string.Format("{0:c}",value)**	Fr. 1'000'000.00
d	Dezimalzahl (decimal)	**string.Format("{0:d}",value)**	1000000
e	Wissenschaftlich (scientific)	**string.Format("{0:e}",value)**	1.00E+06
f	Festkommazahl (fixed point)	**string.Format("{0:f}",value)**	1000000.00
g	Generisch (general)	**string.Format("{0:g}",value)**	1000000
n	Tausender Trennzeichen	**string.Format("{0:n}",value)**	1'000'000.00
x	Hexadezimal	**string.Format("{0:x4}",value)**	f4240

Tabelle 10: Zahlen individuell formatieren (value = 1000000)

Symbol	Typ	Aufruf	Ergebnis
0	0-Platzhalter	**string.Format("{0:00.0000}",value)**	1000000.0000
#	Zahl-Platzhalter	**string.Format("{0:(#).##}",value)**	(1000000)
.	Dezimalpunkt	**string.Format("{0:0.0}",value)**	1000000.0
,	Tausender Trennzeichen	**string.Format("{0:0,0}",value)**	1'000'000
,.	Ganzzahliges Vielfaches von 1.000	**string.Format("{0:0,.}",value)**	1000
%	Prozentwert	**string.Format("{0:0%}",value)**	100000000%
e	Exponenten-Platzhalter	**string.Format("{0:00e+0}",value)**	10e+5

Tabelle 11: Datumswerte formatieren (value = 08.02.2012 07:53, Typ = DateTime)

Symbol	Typ	Aufruf	Ergebnis
d	Kurzes Datumsformat	string.Format("{0:d}",value)	08.02.2012
D	langes Datumsformat	string.Format("{0:D}",value)	Mittwoch, 8. Februar 2012
t	kurzes Zeitformat	string.Format("{0:t}",value)	07:53
T	langes Zeitformat	string.Format("{0:T}",value)	07:53:41
f	Datum + Uhrzeit (kurz)	string.Format("{0:f}",value)	Mittwoch, 8. Februar 2012 07:53
F	Datum + Uhrzeit (lang)	string.Format("{0:F}",value)	Mittwoch, 8. Februar 2012 07:53:41
g	Standard-Datum (kurz)	string.Format("{0:g}",value)	08.02.2012 07:53
G	Standard-Datum (lang)	string.Format("{0:G}",value)	08.02.2012 07:53:41
M	Tag des Monats	string.Format("{0:M}",value)	08 Februar
r	RFC1123 Datumsformat	string.Format("{0:r}",value)	Wed, 08 Feb 2012 07:53:41 GMT
s	sortierbares Datumsformat	string.Format("{0:s}",value)	2012-02-08T07:53:41
u	universell sortierbares Datumsformat	string.Format("{0:u}",value)	2012-02-08 07:53:41Z
U	universell sortierbares GMT-Datumsformat	string.Format("{0:U}",value)	Mittwoch, 8. Februar 2012 06:53:41
Y	Jahr/Monats-Muster	string.Format("{0:Y}",value)	Februar 2012

Tabelle 12: Datumswerte individuell formatieren (value = 08.02.2012 07:53, Typ = DateTime)

Symbol	Typ	Aufruf	Ergebnis
dd	Tag	string.Format("{0:dd}",value)	08
ddd	Tag Name (Kürzel)	string.Format("{0:ddd}",value)	Mi
dddd	Tag Name (ausgeschrieben)	string.Format("{0:dddd}",value)	Mittwoch
gg	Ära	string.Format("{0:gg}",value)	n. Chr.
hh	Stunde zweistellig	string.Format("{0:hh}",value)	07
HH	Stunde zweistellig (24-Stunden)	string.Format("{0:HH}",value)	07
mm	Minute	string.Format("{0:mm}",value)	53
MM	Monat	string.Format("{0:MM}",value)	02
MMM	Monat Name (Kürzel)	string.Format("{0:MMM}", value)	Feb
MMMM	Monat Name (ausgeschrieben)	string.Format("{0:MMMM}",value)	Februar
ss	Sekunde	string.Format("{0:ss}",value)	41
tt	AM oder PM (nur englisch)	string.Format("{0:tt}",value)	
yy	Jahr zweistellig	string.Format("{0:yy}",value)	12
yyyy	Jahr vierstellig	string.Format("{0:yyyy}",value)	2012
zz	Zeitzone (kurz)	string.Format("{0:zz}",value)	+01
zzz	Zeitzone (lang)	**string.Format("{0:zzz}",value)**	+01:00

2.6 Arrays

Arrays, auch Datenfelder genannt, ermöglichen es, eine beinahe beliebig grosse Anzahl von Variablen mit dem gleichen Namen und dem gleichen Datentyp zu definieren. Unterschieden werden die einzelnen Elemente nur anhand eines Index. Arrays kommen hauptsächlich dann zum Einsatz, wenn in Programmschleifen dieselben Operationen auf alle oder einen Teil der Elemente ausgeführt werden sollen.

2.6.1 Deklaration und Initialisierung

Mit der ersten Anweisung in Listing 29 wird ein Array mit dem Namen **intArr** deklariert, dessen Elemente den Datentyp **int** haben. Die Eckigen Klammern hinter dem Datentyp geben an, dass es sich bei der Variable um ein Array handelt. Danach folgt der Bezeichner des Arrays. Wie viele Elemente das Array enthalten wird, ist noch nicht festgelegt.

```
int[] intArr;
intArr = new int[3];

// Die Anzahl Array-Elemente kann auch über eine Variable mitgeteilt werden
int count = 3;
int[] intArr = new int[count];
```

Listing 29: Deklaration und Initialisierung eines int-Arrays

In der zweiten Zeile wird das Array initialisiert. Ein Array ist in C# ein Objekt, deshalb unterscheidet sich die Initialisierung von der einer herkömmlichen Variable. Das Schlüsselwort **new** kennzeichnet die Erzeugung eines Objekts. Dahinter wird der Datentyp und in eckigen Klammern die gewünschte Anzahl Array-Elemente angegeben. In unserem Beispiel verwaltet das Array **intArr** drei Integer-Zahlen.

Manchmal ist zur Entwicklungszeit die für ein Array erforderliche Grösse noch nicht bekannt. In diesem Fall kann die Array-Grösse auch über eine Variable festgelegt werden, die zur Laufzeit mit einem konkreten Wert initialisiert wird.

```
Console.Write("Geben Sie die Anzahl Array-Elemente ein: ");
int count = int.Parse(Console.ReadLine());

int[] intArr = new int[count];
```

Listing 30: Festlegung der Array-Grösse über eine Variable

Bei der Erzeugung eines Arrays werden alle Array-Elemente mit dem Standardwert des entsprechenden Datentyps initialisiert (Standardwert von **int** ist 0). Steht zum Deklarationszeitpunkt bereits fest, welche Daten die Array-Elemente enthalten sollen, können diese in geschweiften Klammern übergeben werden. Es ist zu beachten, dass die Anzahl Initialisierungswerte mit der Anzahl Array-Elemente übereinstimmen muss. Sonst gibt es einen Kompilierungsfehler.

```
int[] intArr;
intArr = new int[3] { 12, 34, 2 };

// Fehler: Die Anzahl Initialisierungswerte muss mit der Anzahl Array-Elemente übereinstimmen
intArr = new int[3] { 12, 34 };

// Werden die Array-Elemente initialisiert, muss deren Anzahl nicht angegeben werden.
intArr = new int[] { 12, 34, 2 };
```

Listing 31: Initialisierung der Array-Elemente

2.6.2 Zugriff auf Array-Elemente

Bei der Initialisierung eines Arrays werden die einzelnen Elemente durchnummeriert. Dabei hat das erste Element den Index 0. Folglich ist der Index des letzten Elements die Array-Länge minus 1.

```
int[] intArr = new int[3];

intArr[0] = 5;           // Dem ersten Element den Wert 5 zuweisen
intArr[2] = 22;          // Dem letzten Element den Wert 22 zuweisen
int intVal = intArr[2];  // Den Wert des letzten Elements der Variable intVal zuweisen
```

Listing 32: Zugriff auf Array-Elemente

Wird versucht auf ein Array-Element zuzugreifen, das gar nicht existiert (Index < 0 oder Index ≥ Array-Länge) wird die Ausnahme **`IndexOutOfRangeException`** ausgelöst und das Programm beim fehlerhaften Zugriff angehalten.

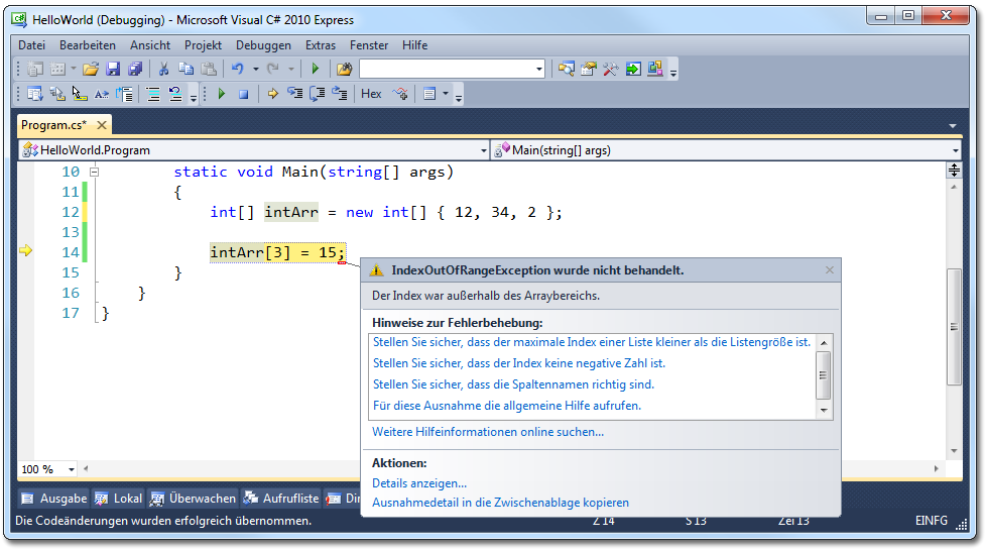

Abbildung 13: Überschreitung der Array-Grenze

2.6.3 Mehrdimensionale Arrays

Zur Darstellung komplexerer Datenstrukturen (z.B. räumliche Daten), sind die bisher betrachteten eindimensionalen Arrays nicht besonders gut geeignet. Daher kommen in der Praxis häufig auch mehrdimensionale Arrays zum Einsatz. Die Anzahl der Dimensionen ist prinzipiell nicht begrenzt. Aus Gründen der Übersicht, wird jedoch selten mit mehr als zwei Dimensionen gearbeitet.

Bei der Deklaration von mehrdimensionalen Arrays werden in den eckigen Klammern kommagetrennt die Anzahl Elemente pro Dimension angegeben.

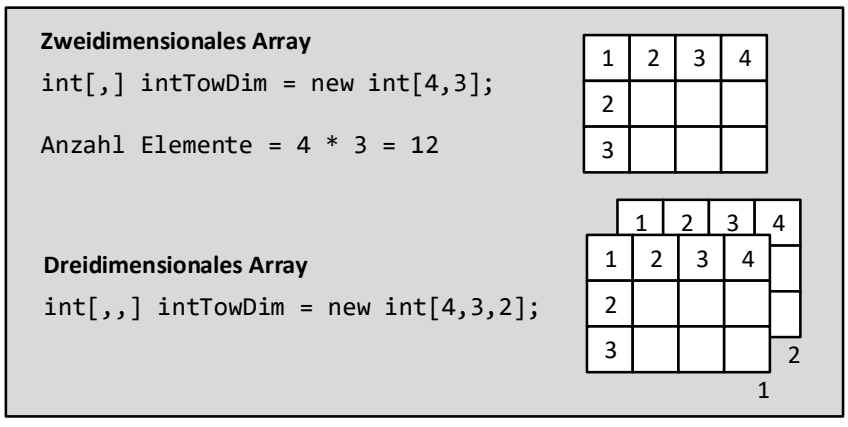

Abbildung 14: Deklaration mehrdimensionaler Arrays

Die gesamte Anzahl Elemente eines mehrdimensionalen Arrays ist das Produkt der Elementzahlen der einzelnen Dimensionen. Ein zweidimensionales Array mit 4 Spalten und 3 Zeilen hat folglich 12 Elemente.

Die Initialisierung von Elementen bei mehrdimensionalen Arrays ist etwas komplexer. In Listing 33 wird ein zweidimensionales Array initialisiert. Die Werte einer Spalte werden dabei wiederum mit geschweiften Klammern in Untergruppen gegliedert.

```
int[,] table = new int[4,3] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };

int intVal = table[1,0]; // intVal = 4
```

Listing 33: Initialisierung der Array-Elemente bei einem zweidimensionalen Array

2.6.4 Anzahl Array-Elemente abfragen

Jedes Array hat die Eigenschaft **`Length`**, mit der die Anzahl Elemente abgerufen werden kann. Dies ist vor allem nützlich, wenn Sie ein Array mit einer Schleife durchlaufen möchten.

```
int[] intArr = new int[3];
Console.WriteLine("Länge des Arrays: " + intArr.Length);
```

Listing 34: Abfrage der Anzahl Array-Elemente mit der Eigenschaft Length

Bei mehrdimensionalen Arrays, gibt die Eigenschaft **`Length`** die gesamte Anzahl Elemente zurück. Möchten Sie nur die Anzahl Elemente einer bestimmten Dimension wissen, können Sie die Methode **`GetLength`** aufrufen. Die Methode erwartet als Parameter die gewünschte Dimension. Dabei gilt, dass die erste Dimension mit 0 angegeben wird, die zweite mit 1 usw.

```
int[,] table = new int[4,3];
Console.WriteLine("Anzahl Zeilen: " + table.GetLength(1));
```

Listing 35: Abfrage der Anzahl Array-Elemente einer bestimmten Dimension

2.7 Dynamische Listen

Ein grosser Nachteil von Arrays ist ihre fixe Grösse. Bereits bei der Erzeugung müssen Sie diese festlegen. Oft ist aber zu diesem Zeitpunkt die erforderliche Anzahl Elemente noch gar nicht bekannt. Aus diesem Grund sind im Namensraum **`Systems.Collections`** spezielle Auflistungsklassen definiert. Mit deren Hilfe können Sie Listen erstellen, deren Grösse beim Hinzufügen von neuen Elementen dynamisch wächst. Es gibt viele Arten von Listen mit unterschiedlichen Eigenschaften. Wir werden aus Zeitgründen nur die beiden generischen Typen **`List<>`** und **`Dictionary<>`** näher betrachten.

2.7.1 List<>

Mit der Klasse **`List<>`** können Sie einfache Listen erstellen. In C# sind übrigens auch alle Listen Objekte und werden deshalb mit Schlüsselwort **`new`** erzeugt (siehe Listing 36). Neben dem Typ **`List`** geben Sie in spitzen Klammern den Typ der Listen-Elemente an. So wird in folgendem Beispiel eine Liste für **`int`**-Werte und eine für Zeichenketten erstellt. Eine neu erzeugte Liste ist standardmässig leer. Sie können jedoch der Liste beim Erzeugen, ähnlich wie bei Arrays, Werte in geschweiften Klammern übergeben.

```
// Listen erzeugen
List<int> intList = new List<int>();
List<string> stringList = new List<string>();

// Liste beim Erzeugen mit Werten initialisieren
List<int> intList = new List<int>() { 1, 2, 3, 4, 5, 6 };
List<string> stringList = new List<string>() { "Text1", "Text2", "Text3" };
```

Listing 36: Erzeugen von Listen

Mit den Methoden **`Add`**, **`Insert`** und **`Remove`** können Sie Werte in die Liste einfügen und entfernen. Bein Einfügen mit **`Insert`** wird der Wert beim angegebenen Index eingefügt. Dadurch werden alle nachfolgenden Elemente um eine Stelle nach hinten verschoben. Mit der Methode **`AddRange`** können Sie alle Werte einer anderen Liste oder eines Arrays auf einmal einfügen. Bei der Methode **`RemoveAt`** geben Sie nicht den Wert an, den Sie entfernen möchten, sondern dessen Index.

```
int intVar = 10;
List<int> intList = new List<int>();

// Der Liste neue Werte hinzufügen
intList.Add(1);
intList.Add(55);
intList.Add(intVar);

// Den Wert 3 an der zweiten Stelle einfügen
intList.Insert(1, 3);

// Andere Liste einfügen
List<int> intList2 = new List<int>() { 10, 11, 12, 13, 14 };
intList.AddRange(intList2);

// Den Wert 55 aus der Liste entfernen
intList.Remove(55);

// Den ersten Wert entfernen
intList.RemoveAt(0);
```

Listing 37: Elemente hinzufügen und entfernen

Die Eigenschaft **`Count`** gibt die Anzahl Elemente einer Liste zurück.

```
Console.WriteLine("Anzahl Werte in der Liste: {0}", intList.Count);
```

Listing 38: Anzahl Elemente abrufen

Um den Wert von bestehenden Listen-Elementen abzurufen oder zu verändern, können Sie, wie bei Arrays, mit eckigen Klammern und dem Index auf das entsprechende Element zugreifen.

```
// Wert eines Listenelements abrufen
Console.WriteLine("Wert Element 2: {0}", intList[2]);

// Wert eines Listenelements ändern
intList[2] = 42;
```

Listing 39: Element-Werte abrufen und ändern

Die Methode **`IndexOf`** gibt den Index des angegebenen Werts in der Liste an. Wenn sich der Wert nicht in der Liste befindet, gibt die Methode -1 zurück.

```
Console.WriteLine("Index von Wert 2: {0}", intList.IndexOf(2));
```

Listing 40: Index eines Werts ermitteln

2.7.2 Dictionary<>

In einem **`Dictionary<>`** können Sie einen Wert zusammen mit einem Schlüssel speichern. Das heisst ein **`Dictionary`** enthält immer Wertepaare. Welche Datentypen der Schlüssel und der Wert haben, wird ebenfalls in spitzen Klammern definiert. Beim ersten handelt es sich um den Typ des Schlüssels und beim zweiten um den Typ des Werts. Auch beim **`Dictionary`** können beim Erzeugen Werte übergeben werden. Dabei werden die Wertepaare mit zusätzlichen Klammern gruppiert.

```
// Dictionary erzeugen
Dictionary<int, string> plzDict = new Dictionary<int, string>();

// Dictionary beim Erzeugen mit Werten initialisieren
Dictionary<int, string> plzDict = new Dictionary<int, string>() {
   { 3000, "Bern"},
   { 3098, "Köniz" } };
```

Listing 41: Erzeugen eines Dictionary

Mit der **`Add`**-Methode können Sie neue Wertepaare einfügen. Dabei handelt es sich beim ersten Parameter um den Schlüssel und beim zweiten um den Wert. Wertepaare lassen sich mit der Methode **`Remove`** auch wieder entfernen. Dazu müssen Sie den Schlüssel des entsprechenden Paars angeben.

Der Schlüssel kann wie ein Array- oder Listen-Index verwendet werden. Wenn Sie den Schlüssel in eckigen Klammern angeben können Sie den zugehörigen Wert abrufen oder verändern. Dabei wird auch klar, dass der Schlüssel innerhalb eines **`Dictionary`** immer eindeutig sein muss. Sonst wäre nicht mehr klar, auf welchen Wert zugegriffen werden soll. Wenn Sie versuchen bei einem **`Dictionary`** denselben Schlüssel zweimal hinzuzufügen gibt es eine **`ArgumentException`**.

```
// Wertepaare einfügen
plzDict.Add(3000, "Bern");
plzDict.Add(3098, "Köniz");
plzDict.Add(3145, "Niederscherli");

// Wertepaar entfernen
plzDict.Remove(3000);

// Wert mit dem Schlüssel abfragen
Console.WriteLine("{0} ist ein schönes Dorf!", plzDict[3098]);

// Der zum Schlüssel "3098" gehörende Wert verändern
plzDict[3098] = "Köniz BE";
```

Listing 42: Wertepaare hinzufügen, entfernen und verändern

Wenn Sie sich nicht sicher sind, ob ein **Dictionary** einen bestimmten Schlüssel bereits enthält, können Sie dies mit der Methode **ContainsKey** überprüfen. Ist der Schlüssel vorhanden, gibt die Methode den Wert **true** zurück.

```
if (!plzDict.ContainsKey(3000))
{
   plzDict.Add(3000, "Bern");
}
```

Listing 43: Überprüfen, ob ein Schlüssel bereits vorhanden ist

Die Eigenschaft **Count** gibt die Anzahl Wertepaare eines **Dictionary** zurück. Mit den Eigenschaften **Keys** und **Values** können Sie eine Liste mit allen Schlüsseln bzw. Werten abfragen.

Mit einem **Dictionary** können Sie also einen Wert mit einem aussagekräftigen Index bzw. Schlüssel verknüpfen.

2.8 Kontrollstrukturen

Programme müssen Entscheidungen treffen, die vom aktuellen Zustand oder von den Benutzereingaben abhängen. Jede Programmiersprache kennt daher Kontrollstrukturen, um den Programmablauf der aktuellen Situation anpassen zu können. In diesem Abschnitt werden Sie die Möglichkeiten kennenlernen, die Sie unter C# nutzen können.

2.8.1 if() else

Die **if**-Anweisung wird verwendet, wenn gewisse Programmteile nur beim Auftreten einer bestimmten Bedingung ausgeführt werden sollen. In Listing 44 sehen Sie die Syntax der **if**-Anweisung. Der **else**-Zweig kann auch weggelassen werden, falls er nicht erforderlich ist. Wenn im **if**- oder **else**-Zweig mehrere Anweisungen ausgeführt werden, müssen diese mit geschweiften Klammern umschlossen werden.

```
if (Bedingung)
   Anweisung1;
else
   Anweisung2;

if (Bedingung)
{
   Anweisung1;
   Anweisung2;
   Anweisung3;
}
else
{
   Anweisung4;
   Anweisung5;
}
```

Listing 44: Syntax der if-Anweisung

Beachten Sie, dass die zu prüfende Bedingung hinter dem Schlüsselwort **if** grundsätzlich immer einen boolschen Wert, also **true** oder **false**, zurückliefern muss.

In der Praxis kommt es häufig vor, dass mehrere Bedingungen, die sich gegenseitig ausschliessen, der Reihe nach ausgewertet werden müssen. Dazu können **if**-Anweisungen auch verkettet werden, wie in Listing 45 zu sehen ist.

```
if (a < 10)
{
   // a ist kleiner 10
   ...
}
else if (a > 10)
{
   // a ist grösser 10
   ...
}
else
{
   // a ist gleich 10
   ...
}
```

Listing 45: Verkettete if-Anweisung

Für einfache bedingte Zuweisungen, wird anstelle einer **if**-Anweisung oft der **?**-Operator verwendet. Möchten wir z.B. von zwei Zahlenwerten den grösseren einer Variablen zuweisen, würde die **if**-Anweisung wie in Listing 46 aussehen. Mit dem **?**-Operator lässt dich das ganze etwas kürzer schreiben. Zuerst wird die Bedingung angegeben. Danach folgen der Operator und zwei Anweisungen, die mit einem Doppelpunkt getrennt werden. Wenn die Bedingung erfüllt ist wird die

erste Anweisung ausgeführt sonst die zweite. Schliesslich wird das Resultat der ausgeführten Anweisung zurückgegeben. In unserem Beispiel ist es der Wert von Variable **b**.

```
int a = 5, b = 10, c;

// Bedingte Zuweisung mit der if-Anweisung
if (a > b)
   c = a;
else
   c = b;

// Bedingte Zuweisung mit dem ?-Operator
c = a > b ? a : b;
```

Listing 46: Bedingte Zuweisung mit dem ?-Operator

2.8.2 switch()

Mit der **switch**-Anweisung lässt sich der Programmablauf ähnlich wie mit der **if**-Anweisung steuern. Dabei wird überprüft, ob der hinter **switch** aufgeführte Ausdruck, der entweder eine Ganzzahl oder eine Zeichenfolge sein muss, mit einer der hinter **case** angegebenen Konstanten übereinstimmt. Dabei wird nacheinander zuerst mit der **Konstante1** verglichen, danach mit der **Konstante2** usw. Stimmt der Ausdruck mit einer Konstante überein, werden alle folgenden Anweisungen bis zur nächsten **break**-Anweisung ausgeführt. Stimmt der Ausdruck mit keiner der Konstanten überein, werden die Anweisungen hinter der **default**-Marke ausgeführt. Der **default**-Block ist optional und kann auch weggelassen werden.

```
switch(Ausdruck)
{
   case Konstante1:
      // Anweisungen
      break;
   case Konstante2:
      // Anweisungen
      break;
   ...
   default:
      // Anweisungen
      break;
}
```

Listing 47: Syntax der switch-Anweisung

Wenn sie in mehreren Fällen dieselben Anweisungen aufführen möchten, können Sie diese zusammenfassen indem Sie die entsprechenden **break**-Anweisungen weglassen.

```
int value = 2;

switch (value)
{
   case 1:
   case 2:
      // value hat den Wert 1 oder 2
      // Anweisungen
      break;
   case 3:
      // Anweisungen
      break;
   default:
      // Anweisungen
      break;
}
```

Listing 48: Syntax switch-Anweisung

2.8.3 while()

Eine **while**-Schleife wird ausgeführt, solange die Bedingung wahr, also **true** ist. Die Schleife wird beendet, wenn die Bedingung **false** ist.

```
while (Bedingung)
{
   // Anweisungen
}

// Beispiel
int i = 0;

while (i < 10)
{
   // Schleife wird 10 mal durchlaufen.
   ++i;
}
```

Listing 49: Syntax der while-Schleife

Wenn die Bedingung bereits bei der ersten Überprüfung nicht erfüllt ist, werden die Anweisungen im Schleifenkörper nie ausgeführt. Da die Bedingung beim Eintritt in die Schleife geprüft wird spricht man auch von einer kopfgesteuerten Schleife.

Möchten Sie nur auf ein bestimmtes Ereignis warten, können Sie den Schleifenkörper einfach weglassen. Zu beachten ist, dass in diesem Fall die Schleife mit einem Semikolon abgeschlossen werden muss.

```
while (Bedingung);
```

Listing 50: while-Schleife ohne Schleifenkörper

2.8.4 do while()

Bei der **do**-**while**-Schleife wird im Gegensatz zur **while**-Schleife die Bedingung am Ende der Schleife geprüft. Man spricht deshalb auch von einer fussgesteuerten Schleife. Die Folge ist, dass die Anweisungen innerhalb des Schleifenkörpers mindestens einmal ausgeführt werden. Beachten Sie, dass die Schleife mit einem Semikolon abgeschlossen werden muss.

```
do
{
   // Anweisungen
} while (Bedingung);

// Beispiel
int i = 0;

do
{
   // Schleife wird 10 mal durchlaufen.
   ++i;
} while (i < 10);
```

Listing 51: Syntax der do-while-Schleife

2.8.5 for()

Die **for**-Schleife wird meistens dann eingesetzt, wenn bekannt ist, wie oft bestimmte Anweisungen ausgeführt werden müssen. Die allgemeine Syntax ist in Listing 52 zu sehen. Um die Anzahl der Durchläufe einer **for**-Schleife festzulegen, wird ein Schleifenzähler benötigt. Im Schleifenkopf wird durch **Ausdruck1** der Startwert und durch **Ausdruck2** der Endwert des Zählers festgelegt. Der dritte Ausdruck bestimmt, um welchen Wert der Zähler bei jedem Durchlauf erhöht wird.

```
for (Ausdruck1; Ausdruck2; Ausdruck3)
{
   // Anweisungen
}

// Beispiel
int[] intArr = new int[] { 1, 2, 3, 4, 5 };

for (int i = 0; i < intArr.Length; ++i)
{
   Console.WriteLine("Wert von Array-Element {0}: {1}", i, intArr[i]);
}
```

Listing 52: Syntax der for-Schleife

Im Beispiel von Listing 52 werden die Werte eines Arrays mit Hilfe einer **for**-Schleife ausgegeben. Der Schleifenzähler heisst **i** und sein Startwert ist 0. Er wird bei jedem Schleifendurchlauf um den Wert 1 erhöht. Wenn der Wert von **i** gleich der Array-Länge ist, wird die Schleife abgebrochen und das Programm fortgesetzt. Der Schleifenzähler **i** entspricht somit dem Array-Index und kann zum Abrufen der Element-Werte benutzt werden. In C# ist es möglich den Zähler direkt im Schleifenkopf zu deklarieren.

Die drei Ausdrücke im Schleifenkopf der **for**-Schleife sind optional. Lässt man alle weg, erhält man eine Endlosschleife. Wichtig ist, dass die zwei Semikolons immer vorhanden sind. Wobei eine „**while (1)“** für eine Endlosschleife vor zu ziehen ist.

```
int a = 0;

for (; a < 10; ++a)
{
}

for (int b = 0; b < 10;)
{
   b += 2;
}

// Endlosschlaufe
for (;;)
{
}
```

Listing 53: Beispiele für for-Schlaufen, die nicht alle drei Ausdrücke im Schleifenkopf verwenden

2.8.6 foreach()

C# bietet mit der **foreach**-Schleife noch eine elegantere Methode, um Arrays und dynamische Listen vom ersten bis zum letzten Element zu durchlaufen.

```
foreach (Datentyp Bezeichner in Array-Bezeichner)
{
   // Anweisungen
}

// Beispiel
int[] intArr = new int[] { 1, 2, 3, 4, 5 };

foreach (int value in intArr)
{
   Console.WriteLine("Wert von Array-Element: {0}", value);
}
```

Listing 54: Syntax der foreach-Schleife

Anstatt jedes Element über seinen Index anzusprechen, wird bei der **foreach**-Schleife eine Laufvariable verwendet, die bei jedem Durchlauf auf ein anderes Array-Element verweist. Daher ist die Indexangabe überflüssig. Die Laufvariable muss zwingend im Schleifenkopf deklariert werden, sonst gibt es einen Kompilierungsfehler.

Den Laufvariablen dürfen keine Werte zugewiesen werden. Somit können Sie mit einer **foreach**-Schleife die Werte eines Arrays nur auslesen aber nicht verändern.

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

foreach (int value in intArr)
{
   value = 0; // Fehler
}
```

Listing 55: Die Werte der Array-Elemente können in einer foreach-Schleife nicht geändert werden

2.8.7 Schleifen mit "break" vorzeitig beenden

Mit der **break**-Anweisung (wie auch mit der **continue**-Anweisung, welche im nächsten Unterkapitel behandelt wird) vorsichtig umgehen, da diese den Programmablauf unter Umständen sehr kompliziert macht. Trotzdem ist es manchmal erforderlich eine Schleife vor der regulären Abbruchbedingung zu beenden. Dies kann mit der **break**-Anweisung erreicht werden. Wenn Sie diese im Schleifenkörper aufrufen, wird die Schleife augenblicklich abgebrochen.

Sie möchten z.B. herausfinden, ob sich in einem Array ein bestimmter Wert befindet. Also durch laufen Sie die Array-Werte mit einer **for**-Schleife und vergleichen sie mit dem gesuchten Wert. Wenn Sie den Wert gefunden haben, wäre es Sinnlos auch noch den Rest des Arrays zu durchsuchen. Deshalb können Sie die Schlaufe mit der **break**-Anweisung abbrechen und so Zeit und Ressourcen sparen.

```
int[] intArr = new int[] { 100, 22, 5, 67, 32, 3, 45, 65, 70 };

bool valueFound = false;

// Prüfen, ob das Array den Wert 5 enthält
for (int i = 0; i < intArr.Length; ++i)
{
   if (intArr[i] == 5)
   {
      valueFound = true;
      break;
   }
}

if (valueFound)
{
   // Anweisungen
}
```

Listing 56: for-Schleife mit break beenden

2.8.8 Schleifenanweisungen mit "continue" überspringen

Bei bestimmten Anwendungen kann es erforderlich sein, dass alle oder ein Teil der Anweisungen im Schleifenkörper nicht bei jedem Durchlauf ausgeführt werden. Dies kann mit der Anweisung **continue** erreicht werden. Wenn Sie **continue** aufrufen, werden alle folgenden Anweisungen im Schleifenkörper übersprungen. Sie gelangen also direkt zum nächsten Schleifendurchlauf.

Listing 57 zeigt eine mögliche Anwendung. Es sollen nur Array-Werte, die grösser als 4 sind im Konsolenfenster ausgegeben werden. Deshalb wird mit einer **if**-Anweisung geprüft, ob die Bedingung erfüllt ist. Ist dies nicht der Fall, wird **continue** aufgerufen und somit die Konsolenausgabe übersprungen.

```
int[] intArr = new int[] { 10, 22, 1, 3, 44, 2, 4, 65 };

for (int i = 0; i < intArr.Length; ++i)
{
   // Werte die kleiner sind als 4 nicht ausgeben
   if (intArr[i] < 4)
   {
      continue;
   }

   Console.WriteLine("Wert von Array-Element {0}: {1}", i, intArr[i]);
}
```

Listing 57: Anweisungen mit continue überspringen

2.9 Exception-Handling

2.9.1 Allgemein

In den vorherigen Kapiteln haben wir gesehen, dass beim Aufruf von Methoden im Fehlerfall eine Ausnahme, eine sogenannte **`Exception`**, ausgelöst wird. Führen wir ein Programm im Debug-Modus aus (*Debuggen / Debugging starten* oder *F5*) stoppt die Entwicklungsumgebung im Falle einer Ausnahme das Programm an der entsprechenden Stelle und zeigt Informationen über die **`Exception`** an. Starten wir das Programm jedoch ganz normal (*Debuggen / Starten ohne Debugging* oder *Strg + F5*), stürzt das Programm beim Eintreten einer **`Exception`** ab und muss beendet werden.

Bei der Applikation in Abbildung 15 wurde eine ungültige Zahl eingegeben, was zu einer unbehandelten **`FormatException`** und somit zum Absturz des Programms geführt hat.

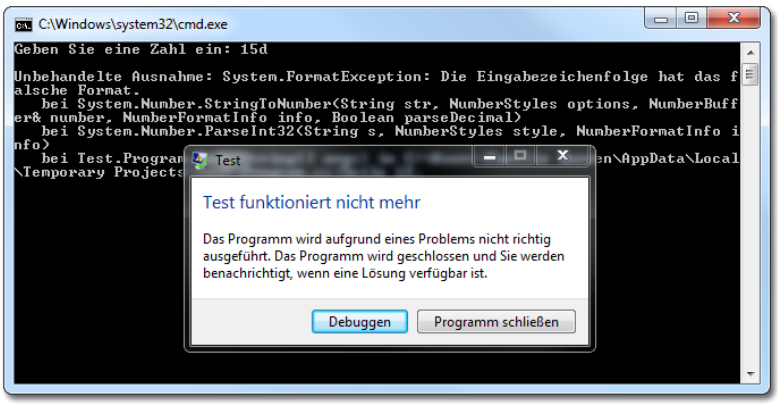

Abbildung 15: Tritt eine unbehandelte Exception auf, stürzt das Programm ab

2.9.2 try catch

Dieses Verhalten ist natürlich völlig inakzeptabel, da Programme sonst bei den geringsten Eingabefehlern bereits abstürzen würden. Deshalb gibt es in C# die **`try-catch`**-Anweisung. Mit dieser können Ausnahmen abgefangen und behandelt werden. So könnte z.B. bei einer fehlerhaften Eingabe der Anwender informiert und zur erneuten Eingabe aufgefordert werden.

Damit wir eine **`Exception`** abfangen können, müssen wir zuerst wissen, welche Ausnahmen bei einem Methodenaufruf überhaupt möglich sind. Dazu können Sie den Mauszeiger auf den Methodenaufruf schieben. Dann erscheint ein Tooltip, der den Zweck der Methode erklärt und alle möglichen Ausnahmen auflistet.

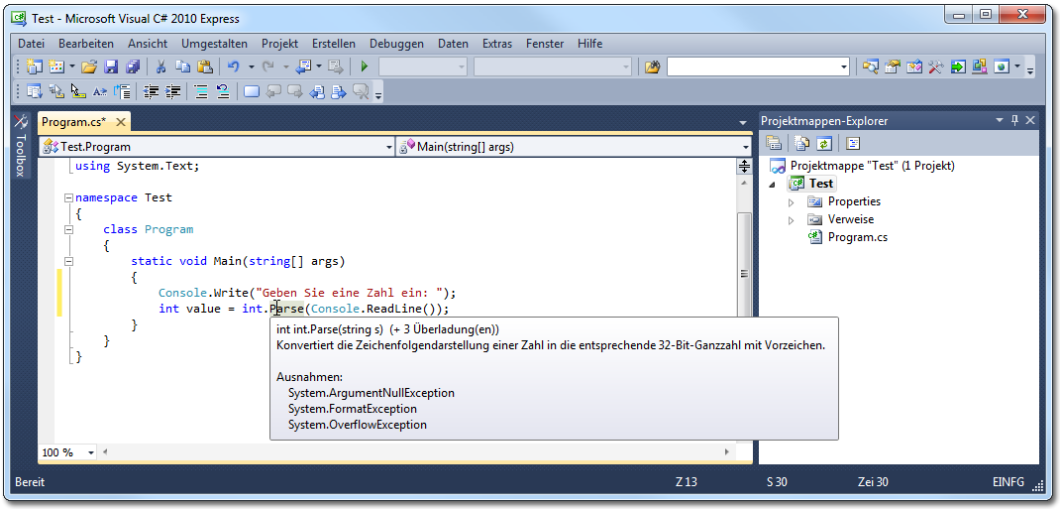

Abbildung 16: Anzeigen, welche Ausnahmen bei einem Methoden-Aufruf auftreten können

Wenn Sie nähere Informationen zu den Ursachen der Ausnahmen haben möchten, setzen Sie den Cursor auf den Methoden-Aufruf und drücken Sie die *F1*-Taste. Dann öffnet Visual C# Express die Online-Hilfe für die entsprechende Methode. Wie in Abbildung 17 zu sehen ist, werden dort alle möglichen Ausnahmen einer Methode beschrieben.

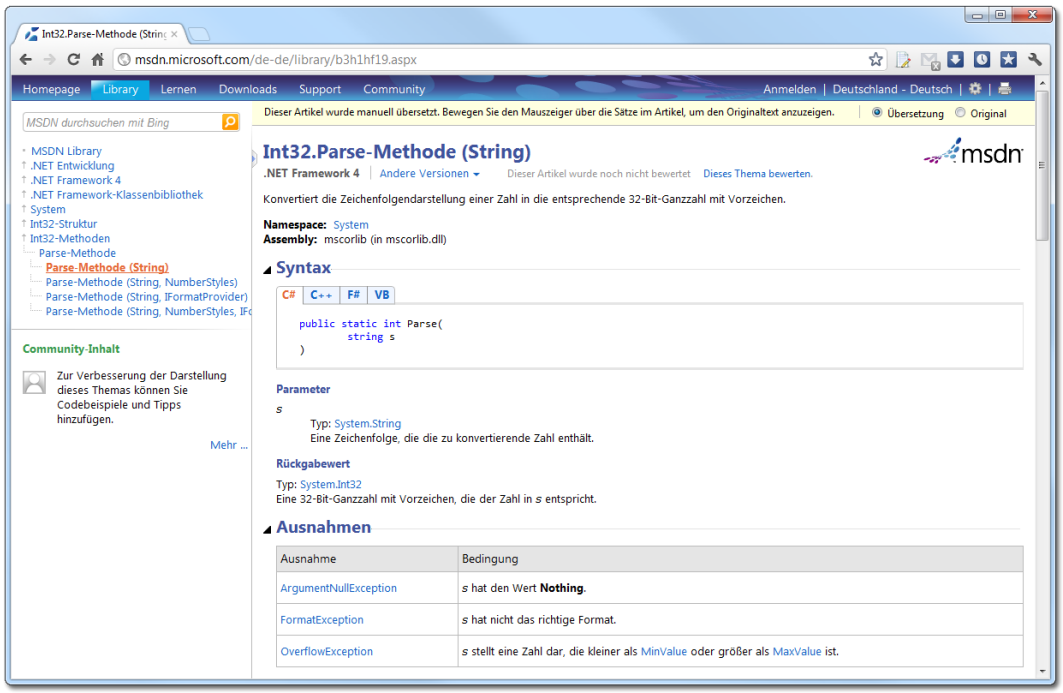

Abbildung 17: Online-Hilfe zu der Methode Int32.Parse

Beim Beispiel in Abbildung 16 wird der Anwender aufgefordert eine Zahl einzugeben. Die eingegebene Zeichenkette wird dann in einen Zahlenwert vom Typ **`int`** konvertiert. Bei der Konvertierung kann es laut Online-Hilfe zu drei verschiedenen Ausnahmen kommen. Die **`FormatException`** tritt auf, wenn der Anwender ungültige Zeichen (z.B. Buchstaben) eingibt und die **`OverflowException`** zeigt an, dass die Zahl für die **`int`**-Variable zu gross ist. Die Ausnahme **`ArgumentNullException`** kann in unserem Fall nicht auftreten, da die Methode **`ReadLine`** nie den Wert **`null`** zurückgibt.

Listing 58 zeigt, wie Sie mir der **`try`**-**`catch`**-Anweisung Ausnahmen abfangen und behandeln können. Zuerst umschliessen Sie die Code-Stelle (es können auch mehrere Zeilen sein) bei der eine Ausnahme auftreten kann mit einem **`try`**-Block. Anschliessend fügen Sie für jede Ausnahme die Sie an dieser Stelle behandeln möchte einen **`catch`**-Block ein. In den **`catch`**-Blöcken können Sie Anweisungen platzieren, die sie im Fehlerfall ausführen möchten.

```
Console.Write("Geben Sie eine Zahl ein: ");

try
{
   int r = int.Parse(Console.ReadLine());
}
catch(FormatException)
{
   Console.WriteLine("Die Eingabe enthält ungültige Zeichen!");
}
catch(OverflowException)
{
   Console.WriteLine("Die Zahl ist zu gross!");
}
```

Listing 58: try-catch-Anweisung

Ausnahmen, die Sie nicht mit einem **`catch`**-Block abfangen, werden an die aufrufende Methode weitergegeben. Spätestens in der **`Main`**-Methode müssen alle Ausnahmen abgefangen werden, sonst

stürzt die Applikation ab. Im Beispiel von Listing 59 wird in der Methode **ReadIntValue** nur die **FormatException** abgefangen. Wenn bei der Umwandlung eine **OverflowException** auftritt, wird diese an die **Main**-Methode weitergegeben, wo sie abgefangen und behandelt wird. Zu beachten ist, dass im Falle einer **FormatException**, der **return**-Befehl auf Zeile 31 nicht mehr ausgeführt wird. Da eine Methode mit Rückgabewert aber in jedem Fall (auch im Fehlerfall) einen Wert zurückgeben muss, gibt es auf Zeile 38 ein weiteres **return**, das nur bei einer **FormatException** aufgerufen wird und den Wert -1 zurückgibt.

```
01  using System;
02  using System.Collections.Generic;
03  using System.Linq;
04  using System.Text;
05
06  namespace ExceptionHandling
07  {
08      class Program
09      {
10          static void Main(string[] args)
11          {
12              Console.Write("Geben Sie eine Zahl ein: ");
13
14              try
15              {
16                  int value = ReadIntValue();
17                  break;
18              }
19              catch(OverflowException)
20              {
21                  Console.WriteLine("Die Zahl ist zu gross!");
22              }
23
24             Console.ReadKey();
25          }
26
27          static int ReadIntValue()
28          {
29              try
30              {
31                  return int.Parse(Console.ReadLine());
32              }
33              catch(FormatException)
34              {
35                  Console.WriteLine("Die Eingabe enthält ungültige Zeichen!");
36              }
37
38              return -1;
39          }
40      }
41  }
```

Listing 59: Exception-Handling auf verschiedenen Ebenen

2.9.3 finally

Nehmen wir an, dass Sie in einer Methode eine Datei öffnen Daten auslesen und am Schluss die Datei wieder schliessen. Falls es während dem Auslesen der Daten zu einer Ausnahme kommt, wird die Datei gar nicht mehr geschlossen, da der normale Programmfluss unterbrochen wird und die Anweisung zum Schliessen der Datei gar nicht mehr erreicht wird.

Die Fehlerbehandlung von C# bietet dazu optional noch eine weitere Klausel an, in der solche Aufräumarbeiten erledigt werden können. Unmittelbar nach dem letzten **catch**-Block, können Sie einen weiteren Anweisungsblock einfügen, der mit dem Schlüsselwort **finally** eingeleitet wird.

```
static void ReadFileData()
{
   try
   {
      // Datei öffnen
      ...
      // Daten auslesen
      ...
   }
   catch(Exception)
   {
      // Fehlerbehandlung
      ...
   }
   finally
   {
      // Datei schliessen
      ...
   }
}
```

Listing 60: try-catch-Anweisung mit finally-Block

Sobald das Programm in einen **try**-Block eingetreten ist, wird der zugehörige **finally**-Bock in jedem Fall ausgeführt, unabhängig davon, ob eine Ausnahme ausgelöst worden ist oder nicht. Es ist also sicherer, wenn Sie reservierte Ressourcen wie z.B. Dateien im **finally**-Block freigeben, da die Freigabe dann auch im Fehlerfall erfolgt.

2.9.4 Eigene Ausnahmen auslösen

Sie können selbst auch Ausnahmen mit dem Schlüsselwort **throw** auslösen. Nehmen wir an, Sie schreiben eine Methode, die zwei Zahlenwerte multipliziert und das Resultat als **int**-Wert zurückgibt. Wenn das Resultat der Multiplikation grösser ist als der maximale **int**-Wert, kommt es zu einem Überlauf und das Resultat wird verfälscht. Schön wäre es deshalb, wenn Ihre Methode in diesem Fall eine **OverflowException** auslösen würde.

Listing 61 zeigt, wie sie eine **OverflowException** auslösen können. Bei Ausnahmen handelt es sich auch um Objekte. Deshalb müssen Sie diese zuerst mit dem Schlüsselwort **new** erzeugen, bevor Sie sie mit **throw** auslösen können. Später werden wir sehen, dass Sie auch eigene **Exception**-Typen definieren können.

```
01  static int Multiply(int a, int b)
02  {
03     if ((long)a * b > int.MaxValue)
04     {
05        throw new OverflowException();
06     }
07
08     return a * b;
09  }
```

Listing 61: Eigene Exception auslösen

Bei der Prüfung auf Zeile 3 wird das Resultat des Ausdrucks **a * b** vor dem Vergleich mit dem Maximalwert noch im Speicher abgelegt. Auch in diesem Fall wird so viel Speicherplatz reserviert, wie der grösste Operand belegt. Deshalb wird ein Operand in den Typ **long** konvertiert, da sonst das Resultat der Multiplikation beim Ablegen in den Speicher auf einen **int**-Wert begrenzt würde und daher nie grösser als der Maximalwert wäre.