Christian Wenz
Tobias Hauser
Jürgen Kotz
Karsten Samaschke

inklusive DVD-ROM mit

Microsoft Visual Studio 2010 Express

ASP.NET 4.0 mit Visual C# 2010

Leistungsfähige Webapplikationen programmieren

ADDISON-WESLEY

DVD

3

Spracheinführung C# 4.0

Zur optimalen Nutzung der Möglichkeiten, die Ihnen mit ASP.NET 4.0 zur Verfügung stehen, sollten Sie nicht nur die reinen ASP.NET-Programmierelemente verwenden. Verknüpfen Sie doch ASP.NET mit einer Programmiersprache. Diese wird Ihnen bei der Entwicklung wertvolle Hilfe leisten können.

Die wesentlichen Standardsprachen sind C# (sprich C sharp), eine Weiterentwicklung von Microsoft, welche die Vorteile von C++ und Visual Basic vereinen soll, und Visual Basic. Sie haben aber auch die Möglichkeit, andere Sprachen wie C++ (eine objektorientierte Weiterentwicklung der Programmiersprache C, die ihre Wurzeln schon in den 70er Jahren hatte) zu verwenden.

Wir werden uns in diesem Kompendium auf C# in der aktuellen Version 4.0 als Programmiersprache beschränken (im weiteren Verlauf werde ich jedoch zur Vereinfachung an vielen Stellen nur noch C# verwenden), da hiermit die wesentlichen Konzepte einfach und umfassend abgedeckt werden können.

3.1 Zur Einführung: Die Geschichte von C#

Im Gegensatz zu Visual Basic ist die Geschichte von C# relativ kurz.

C# wurde 2002 mit der Einführung von .NET und Visual Studio 2002 auf den Markt gebracht. Bei C# handelt es sich um eine typsichere objektorientierte Sprache, welche die Mächtigkeit von C++ mit der Einfachheit von Visual Basic vereinen soll.

Der Begriff C# kommt dabei aus der Musik und soll zum Ausdruck bringen, dass es ähnlich wie der Halbton C# um eine Oktave höher als C ist.

Mit Einführung von Visual Studio 2005 und dem .NET Framework 2.0 erschien im Jahre 2005 die Version 2.0 der Sprache C#. Die Implementierung von Generics war ein wesentlicher Bestandteil dieser neuen Version.

Mit dem .NET Framework 3.5 und Visual Studio 2008 wurde dann die Version 3.5 vorgestellt, die einige Spracherweiterungen, insbesondere im Hinblick auf LINQ (Language Integrated Query), mit sich brachte.

Aktuell sind wir jetzt bei C# 4.0 und dem .NET Framework 4.0. In dieser Version wurden die Möglichkeit der dynamischen Programmierung sowie einige bereits aus VB bekannten Features auch in C# eingeführt.

3.2 Programmierung mit dem Visual Web Developer

Bevor wir uns mit den eigentlichen Programmierelementen beschäftigen, welche die Programmiersprache C# ausmachen, wollen wir Ihnen kurz die besonderen Features vorstellen, die Ihnen zur Verfügung stehen, wenn Sie zur Programmerstellung den Visual Web Developer verwenden.

Im vorhergehenden Kapitel haben Sie die Installation aller ASP.NET-Komponenten kennengelernt. Wir haben uns auf den Visual Web Developer 2010 in der Express Edition konzentriert, da dieser als Freeware zur Verfügung steht. Er liegt diesem Buch außerdem auf der DVD bei.

Mit dem Visual Web Developer erhalten Sie nicht nur alle für die ASP.NET-Entwicklung wesentlichen Werkzeuge, nebenbei werden auch für Ihre C#-Programmierung wesentliche unterstützende Funktionalitäten bereitgestellt.

Sie erhalten Vorlagen, in denen bereits ein Grundgerüst für Ihr Projekt vorgegeben wird, so dass Sie Ihren eigenen Programmcode nur noch ergänzen müssen. Dies wird Ihnen sehr viel Tipparbeit sparen.

Weiterhin wird der Programmcode automatisch formatiert und Schlüsselwörter werden hervorgehoben. Die notwendigen Einrückungen für Schleifen oder Bedingungen werden automatisch vorgenommen und im Hintergrund läuft ein Parser, der automatisch die Syntax prüft und auf eventuelle Fehler hinweist, bevor die Seite ausgeführt wird.

Als weiteres Tool wird zur Unterstützung der Entwicklung ein lokaler Webserver mitgeliefert, der für die Darstellung und Ausführung der kompilierten serverseitigen Programme auf Ihrem lokalen Rechner sorgen kann.

Nachfolgend wollen wir Ihnen die ersten Schritte kurz vorstellen, die Sie durchführen müssen, um in Ihre ASP.NET-Seiten C#-Code einzubinden. Weiterhin werden wir auch auf die Trennung von ASP.NET-Code und reinen C#-Abschnitten eingehen.

In einem späteren Abschnitt, wenn Sie erste Schritte mit C# gemacht haben, werden wir Ihnen dann weitere Features des Visual Web Developers im Detail vorstellen, die Ihnen die Programmierung erleichtern werden.

3.2.1 Erzeugung einer Website

Zunächst machen wir uns einmal die Möglichkeiten des Visual Web Developers zunutze und erzeugen eine einfache ASP.NET-Website.

Dazu haben Sie zwei Möglichkeiten. Sie können einerseits den Link Neue Website... auf der Startseite anklicken oder unter dem Menüpunkt Datei den darunter liegenden Menüpunkt Neue Website aufrufen. Bei beiden Möglichkeiten öffnet sich ein Dialog, den wir hier einmal abgebildet haben.

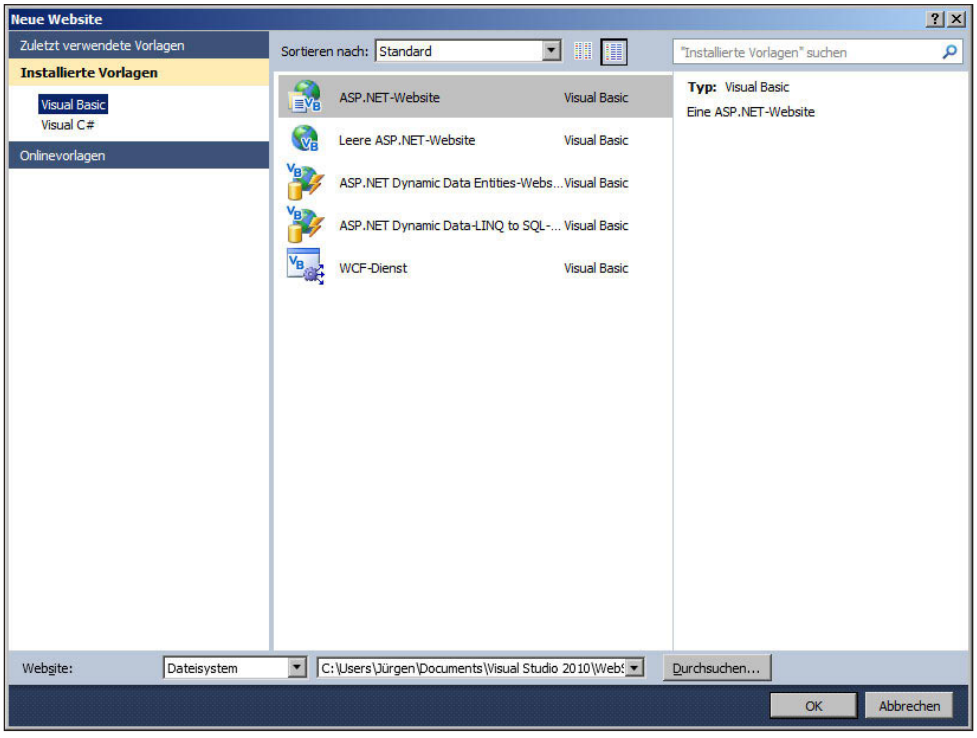


Abbildung 3.1: Erstellen einer neuen Website, Schritt 1

Unter Installierte Vorlage wird hier die Standardsprache für die Webseite festgelegt. Als Standardwert ist hier der Wert Visual Basic vorgegeben, ändern Sie diesen auf C#. Sie können den Speicherort (im unteren Bereich des Dialogs) der Webseite sowohl über das Dateisystem als auch über eine URL (mit dem Feld http) oder die Angabe einer FTP-Adresse (mit dem Feld ftp) festlegen. Wenn Sie eine URL festlegen, können Sie entweder auf einen lokal installierten IIS referenzieren oder aber auch auf einen Remote-Webserver, solange er ebenfalls IIS verwendet. Auch über FTP können Sie eine Remote-Website definieren und an diese Dateien weitergeben.

Welche Art der Speicherung Sie verwenden, hängt vom Gesamtumfeld ab, in dem Sie sich bewegen. Wenn Sie in einer Gruppe entwickeln und auf einem gemeinsamen Server testen wollen, bietet sich eine Remote-Website an. Wenn Sie keinen IIS installieren und nur lokale Tests auf Ihrem Rechner machen wollen sowie keine IIS-Features nutzen, ist die Speicherung im Dateisystem eventuell besser geeignet.

Wir verwenden hier einen Ort im Dateisystem. Sie haben unter C# eine Reihe von Möglichkeiten, wählen Sie hier zunächst einmal die ASP.NET-Website aus und klicken Sie dann den ok-Button. Ihr Frontend hat nun mehr oder weniger das nachfolgende Aussehen:

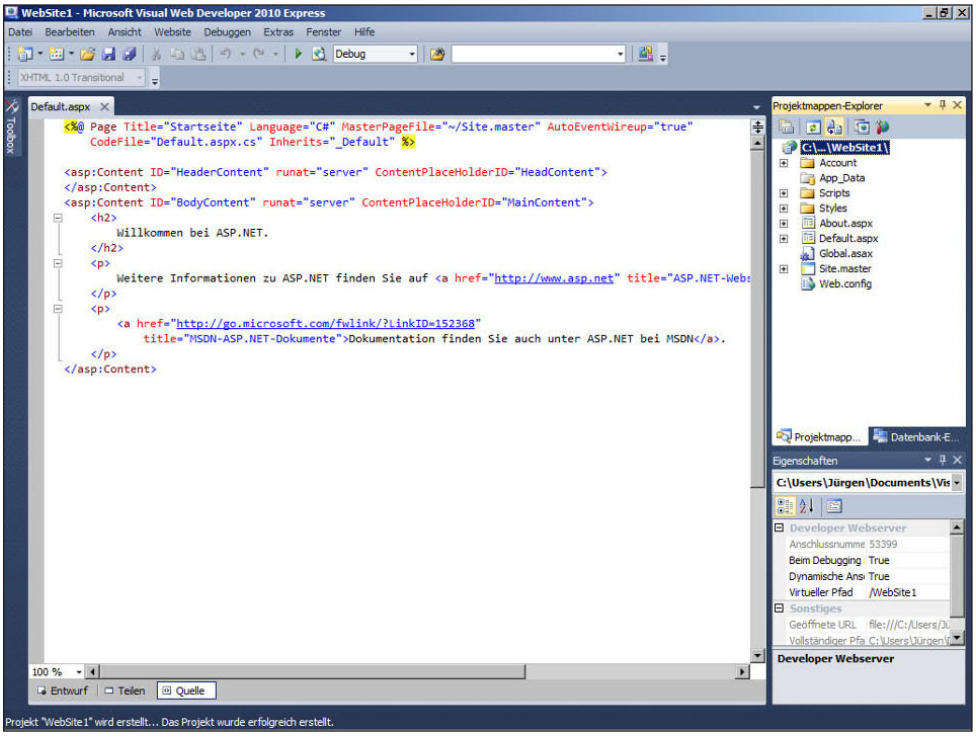


Abbildung 3.2: Frontend nach Erstellung der neuen Webseite (Schritt 2)

Neben der bereits geöffneten Seite *Default.aspx* wurden noch eine Reihe weiterer Dateien und Verzeichnisse erstellt. Die Dateistruktur sieht in etwa wie folgt aus:

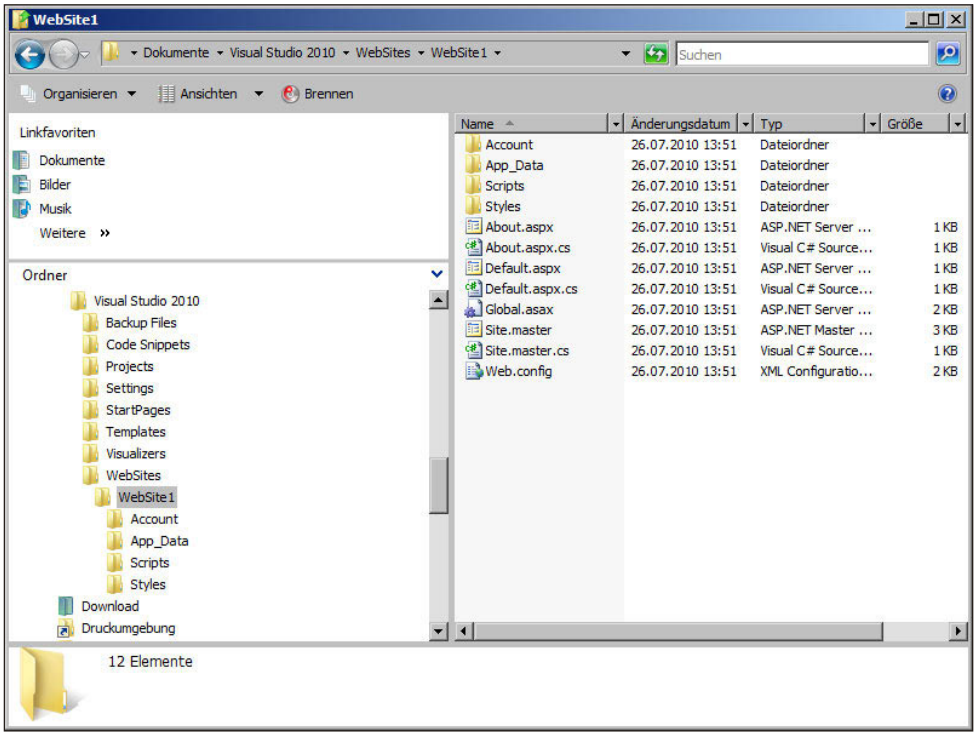


Abbildung 3.3: Angelegte Dateistruktur nach Erstellung einer neuen Website

Uns interessiert nachfolgend die Seite *Default.aspx*. Sie können den C#-Code auch vollständig von den ASP.NET-Elementen abtrennen und diesen in der korrespondierenden Webseite *Default.aspx.cs* ablegen. Wir beschränken uns in diesem Kapitel auf die Verwendung von C#-Code innerhalb einer ASP-Seite.

Mit diesen **.aspx*-Seiten können Sie Ihre ersten Schritte mit der Programmiersprache C# durchführen.

3.2.2 Das obligatorische »Hello World«

Bevor wir uns auf die wesentlichen Programmierelemente von C# konzentrieren, sollten Sie die Programmierumgebung und die einfache selbstständige Erstellung von Webseiten besser kennenlernen. Als Teil des Visual Web Developers wird auch ein lokaler Webserver mit ausgeliefert, der auf Ihrem Rechner ausgeführt wird. Dadurch können Sie die Ergebnisse Ihrer Arbeit ohne großen Aufwand quasi im Entstehen betrachten und kontrollieren.

Das Programm

Um Ihr erstes Programm in C# zu schreiben, löschen Sie als Erstes den gesamten vorbereiteten Code und ersetzen ihn durch die nachfolgenden Zeilen:

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
      Response.Write("Hello World");
  }
</script>
```

Listing 3.1: Das erste C#- und ASP.NET-Programm (Hello_World.aspx)

Speichern Sie diesen Programmtext nun unter dem Namen *Hello_World.aspx* ab. Die Validierungswarnung werden wir zu diesem Zeitpunkt nicht beachten. Sie finden diesen Quelltext übrigens wie alle anderen Programme auch auf der beiliegenden DVD.

Dies stellt den ersten Quelltext dar, den Sie erstellt haben. Nachfolgend wollen wir Ihnen noch kurz erläutern, was dieser Programmcode bedeutet.

Die erste Zeile ist die sogenannte Page-Direktive. In ihr wird festgelegt, dass die für diese Seite verwendete Programmiersprache C# ist (`Language="C#"`).

Die nächste und die letzte Zeile bedeuten, dass der von ihnen eingeschlossene Teil ein Skript darstellt (aufgrund der Page-Direktive in der Sprache C#). Als Programmtext wird eine Funktion mit dem Namen `Page_Load()` erstellt. Das Schlüsselwort `void` besagt, dass diese Funktion keinen Rückgabewert besitzt. Der Inhalt der Funktion ist innerhalb der geschweiften Klammern, die den Funktionsblock darstellen, gekapselt.

Der Inhalt dieser Prozedur besteht aus einer einzigen Zeile: Die Ausgabe im Browser wird durch die ASP.NET-Methode `Response.Write()` ausgeführt, die den in Anführungszeichen gesetzten Text erzeugt.

Die Ausgabe

Wie lässt sich nun eine Ausgabe erzeugen bzw. das Ergebnis betrachten? Hierzu haben Sie ein weiteres Tool im Visual Web Developer, das Ihnen die Arbeit erleichtert: Sie finden es unter Debuggen, wie im nachfolgenden Bild beschrieben.

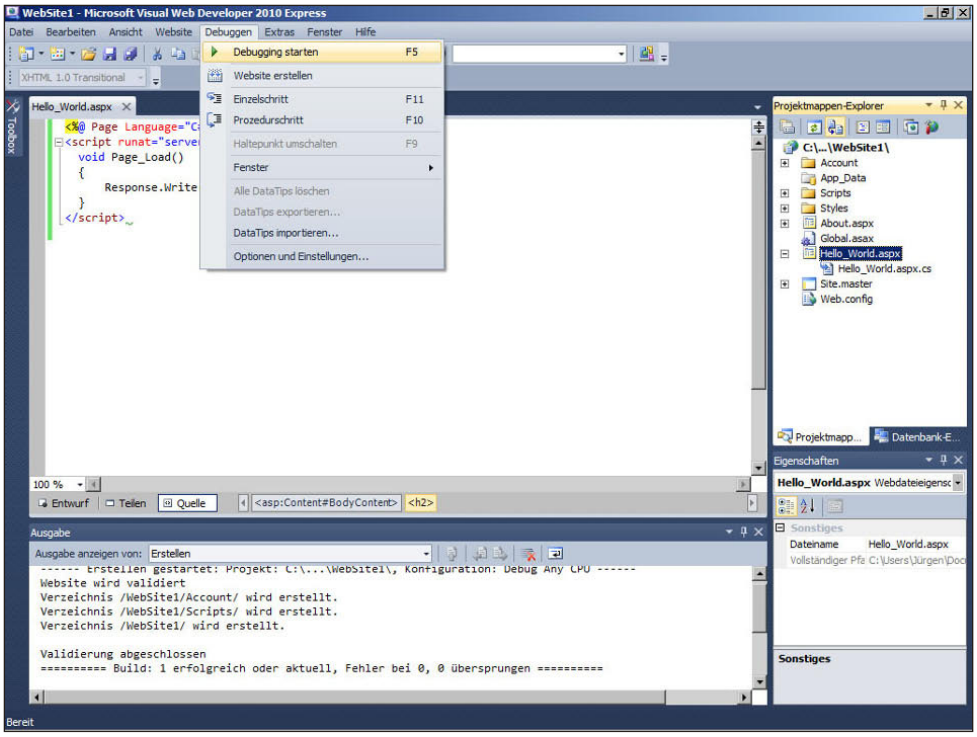


Abbildung 3.4: Start der Ausgabe des »Hello World«-Programms

Wenn Sie dies durchführen, wird ein Browser gestartet, auf dem Sie die Ausgabe betrachten können. Dies wird in Abbildung 3.6 dargestellt.

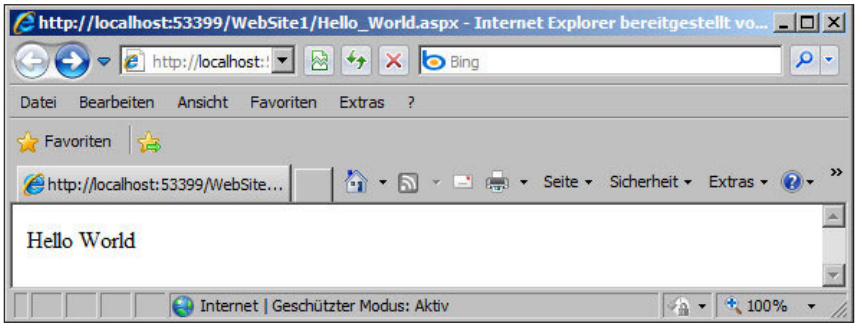


Abbildung 3.5: Ausgabe von Hello World

Dasselbe Ergebnis hätten Sie übrigens auch erhalten, wenn Sie F5 gedrückt hätten.

Üblicherweise wird für die Anzeige der übersetzten Programme der Internet Explorer als Browser gestartet. Die im Browser angezeigte URL können Sie aber auch einfach, wie wir es hier vorgeführt haben, durch einen anderen Browser darstellen lassen. Die erzeugten Programme sind also grundsätzlich unabhängig vom eingesetzten Browser.

Was ist sonst noch passiert?

Mit dem Drücken auf den Debuggen starten F5-Knopf ist nicht nur der Webbrowser mit der anzuzeigenden Seite geladen worden. Bevor dies geschehen ist, ist im Hintergrund der ASP.NET Development Server gestartet worden, den Sie mit dem Visual Web Developer gemeinsam installiert haben und der in den Standardeinstellungen als Webserver zur Verfügung steht. Dass dieser Service gestartet wurde, erkennen Sie in der Taskleiste.

Sie können sich die Details mit einem Rechtsklick ansehen:

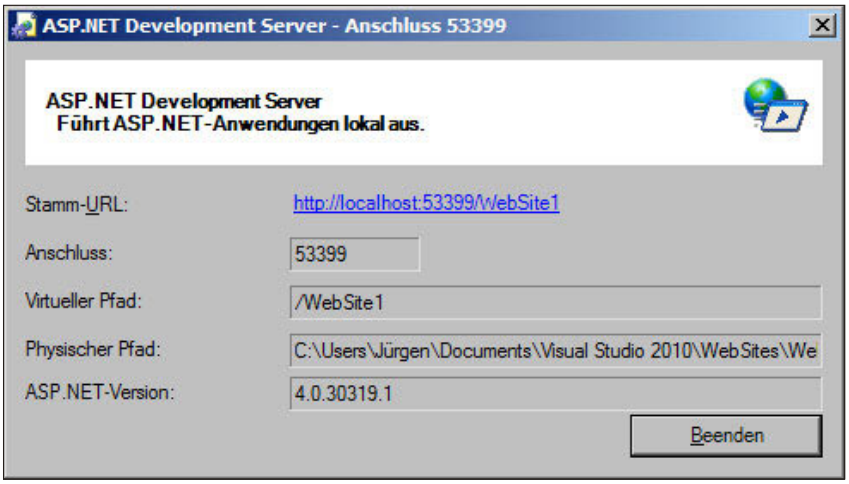


Abbildung 3.6: Anzeige des ASP.NET Development Servers

Dieser Service steht Ihnen allerdings nur lokal auf Ihrem Entwicklungsrechner zur Verfügung.

Weiterhin ist Ihr Programm im Hintergrund übersetzt worden. Es ist ausführbarer Code erzeugt worden und es hat eine Überprüfung Ihres Codes stattgefunden.

Sie können sich diese Codeprüfung selbst ansehen, wenn Sie im unteren Bildschirmabschnitt im Ausgabefenster den Wert für den Menüpunkt Ausgabe anzeigen von: auf Erstellen setzen. Ihr Ergebnis müsste in etwa so aussehen, wie es das nachfolgende Bild auch zeigt.

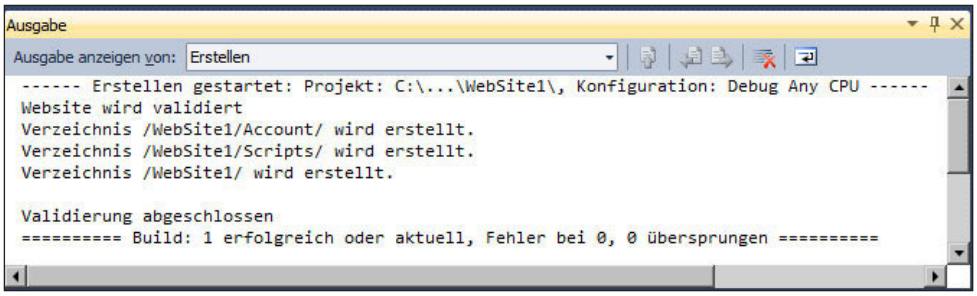


Abbildung 3.7: Ergebnis der Ausgabe des Erstellungsverlaufs von »Hello World.aspx«

Nun kennen Sie das Handwerkszeug, um Ihre ersten ASP.NET-Programme mit C# selbst schreiben und überprüfen zu können.

3.3 Grundbegriffe von Datentypen bis zu Schleifen

Im nachfolgenden Abschnitt werden wir Ihnen die wesentlichen Schlüsselwörter und Konzepte der Programmierung vorstellen sowie auf die Besonderheiten eingehen, die C# von anderen Programmiersprachen unterscheidet.

Die grundlegenden Konzepte einer Programmiersprache setzen wir hierbei voraus, so dass, falls Sie ein absoluter Programmieranfänger sind, Ihnen die Darstellung eventuell zu kompakt sein könnte. Sie werden genug ausführliche Spracheinführungen in der einschlägigen Fachliteratur finden können.

3.3.1 Standarddatentypen

Um Daten im Computer vorzuhalten und manipulieren zu können, werden üblicherweise Variablen definiert, die in unterschiedlichen Datentypen als Standard bereitgestellt werden. Sie haben auch die Möglichkeit, eigene Datentypen selbst zu definieren.

Grundsätzlich müssen Sie Variablen vor der ersten Verwendung deklarieren, das heißt, Sie müssen festlegen, welchen Namen die von Ihnen festzulegende Variable hat und welcher Datentyp dieser Variablen zugeordnet ist. Eine Variable definieren Sie einfach, indem Sie den gewünschten Datentyp gefolgt vom Variablennamen hinschreiben. Eine Deklaration von Variablen hat also die nachfolgende Struktur:

```
Datentyp variablenname;
```

Nach der Deklaration von Variablennamen und zugehörigem Datentyp können Sie (auch als Teil der Deklaration) per Zuweisung dieser Variable einfach Werte zuordnen. Bei lokalen Variablen müssen Sie in C# vor dem Abrufen der Variablen den Wert initialisieren.

Daher bietet es sich an, einfach strukturierte Variablen bei ihrer Deklaration gleich mit einem benötigten Wert zu versehen.

```
Datentyp variablenname = wert;
```

ACHTUNG

Beachten Sie, dass der Wert einer lokalen Variablen in C# ohne eine vorher durchgeführte Zuweisung im Gegensatz zu Visual Basic nicht automatisch vorinitialisiert wird.

Seit C# 3.0 ist auch eine *implizite Typisierung* von lokalen Variablen möglich. Das bedeutet, Sie können bei der Definition der Variablen auf den Datentyp verzichten. Dies geschieht mit dem Schlüsselwort `var`. Allerdings funktioniert das nur, wenn der Variablen sofort ein Wert zugewiesen wird und wenn die Variable nur einen lokalen Gültigkeitsbereich besitzt.

```
var variablenname = wert
```

Der Compiler erkennt aufgrund des zugewiesenen Werts automatisch den Datentyp.

ACHTUNG

Bei der impliziten Typisierung ist ein nachträgliches Ändern des Datentyps nicht mehr möglich.

Auch IntelliSense unterstützt diese Art der Variablendefinition, wie Sie in Abbildung 3.8 sehen. In dem gezeigten Fall werden sämtliche Methoden und Eigenschaften des Datentyps `string` vorgeschlagen.

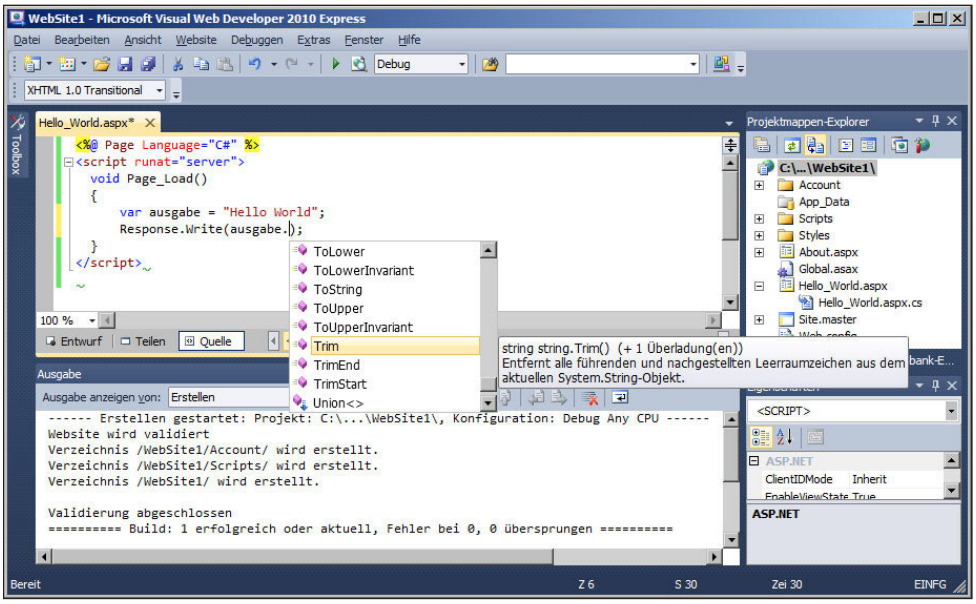


Abbildung 3.8: IntelliSense bei impliziter Typisierung

ACHTUNG

Verwenden Sie jedoch implizite Typisierung nicht grundsätzlich, denn dies kann Ihren Programmcode sehr schnell unleserlich machen. Dieses Sprachmerkmal wurde im Hinblick auf LINQ (mehr dazu in Kapitel 16) eingefügt und sollte eigentlich nur für diese Zwecke in Kombination mit den dafür anderen neuen Sprachmitteln eingesetzt werden.

Die grundlegenden möglichen Datentypen werden wir Ihnen nachfolgend vorstellen.

Unterschiedliche Datentypen für Variablen wurden unter anderem deswegen eingeführt, da es sich als erheblich effizienter herausgestellt hat, unterschiedliche Manipulierungswerkzeuge nur

für bestimmte Datentypen zuzulassen; sowohl die Effizienz bei der Speicherung als auch bei der Verarbeitung wird über diese Standarddatentypen deutlich gesteigert.

Es macht beispielsweise keinen Sinn, eine Multiplikationsoperation für Zeichenketten zu definieren. Andererseits können Integer-Rechenoperationen und Fließkommaberechnungen prozessoroptimiert durchgeführt werden.

Zahlentypen

Für die Speicherung von Zahlen und Operationen mit diesen gibt es eine breite Anzahl unterschiedlicher Datentypen. Es gibt beispielsweise Datentypen für die Speicherung von Bytes (dies sind ganze Zahlen von 0 bis 255). Für eine solche Zahl ist auch nur ein Byte Speicherplatz zur Speicherung notwendig. Der Wertebereich von Bytes ist somit auch nur auf 256 Werte beschränkt.

Wenn Sie ganze Zahlen mit einem größeren Wertebereich als dem Wertebereich eines Bytes speichern wollen, stehen Ihnen dafür mehrere weitere Integer-(Ganzzahlen-)Datentypen wie `Short` (2 Byte), `Integer` (4 Byte) oder `Long` (8 Byte) zur Verfügung. In .NET 4.0 wurde für sehr große Ganzzahlen ein neuer Datentyp `BigInteger` eingeführt, um die bislang bestehenden Limitationen zu überbrücken. Dieser neue Datentyp befindet sich dabei im Namespace `System.Numerics`, der jedoch erst nach dem Hinzufügen eines Verweises auf die gleichnamige Bibliothek zur Verfügung steht.

Wollen Sie Zahlen mit Nachkommastellen (sogenannte Gleitkommazahlen) verwenden, so gibt es auch hierfür Datentypen, die eine Darstellung und optimierte Rechenoperationen mit diesen Zahlentypen ermöglichen.

Aus Speicherplatzgründen stehen Ihnen unterschiedliche Datentypen mit verschiedener Genauigkeit oder auch einem verschieden großen Wertebereich zur Verfügung. Als der häufigste sei hier der Datentyp `Double` genannt.

In der nachfolgenden Tabelle haben wir für Sie alle standardmäßig in C# vordefinierten Standarddatentypen für Zahlen aufgelistet.

Bezeichnung	Wertebereich von	Wertebereich bis	Größe in Bit
`byte`	0	255	8
`short`	–32768	32767	16
`Int`	–2.147.483.648	2.147.483.647	32
`long`	–9.223.372.036. 854.775.808	9.223.372.036. 854.775.807	64
`float/ single`	–3,402823*10^38	3,402823*10^38	32
`double`	–1,79769313486232 *10^308	1,79769313486232 *10^308	64
`decimal`	–79.228.162.514. 264.337.593.543. 950.335	79.228.162.514.264. 337.593.543.950.335	96

Tabelle 3.1: Standard-Zahlendatentypen

Der Datentyp `decimal` stellt bereits eine Besonderheit dar, da Sie festlegen können, wie viele Nachkommastellen Ihr Zahlenwert haben soll. Nach dieser Festlegung wird ohne Rundungen bis auf diese Nachkommastellen gerechnet. Aus diesem Grund bietet sich dieser Datentyp für die Werte an, die zwar Nachkommastellen besitzen, aber eine geringe Anzahl von Nachkommastellen haben (wie beispielsweise bei Geldbeträgen).

Sie haben die Möglichkeit, weitere Datentypen zu definieren oder bereits im .NET Framework definierte Zahlentypen zu verwenden. Diese stellen allerdings keine Standard-Zahlentypen dar. So gibt es Integer-Zahlentypen, deren Wertebereiche grundsätzlich positiv sind, sogenannte Unsigned Integers (`uint`, `ulong`, `ushort`). Außerdem steht Ihnen noch ein Datentyp der Länge Byte zur Verfügung, der mit einem Vorzeichen versehen ist (`sbyte`).

Zeichentypen und Datentypen für Zeichen und Zeichenketten

Für die Verwendung von Zeichen und Zeichenketten (also Aneinanderreihungen von Zeichen) stehen ebenfalls Standarddatentypen zur Verfügung.

Der Bezeichner für den Datentyp für Zeichen lautet `char` und der Datentyp für Zeichenketten lautet `string`.

Der Wertebereich für `char` ist dabei 16 Bit und bezieht sich auf die Unicode-Zeichentabelle des Systems (also 2 Byte pro Zeichen).

Der Datentyp `string` stellt einfach eine sequenzielle Liste von miteinander verknüpften einzelnen Zeichen vom Datentyp `char` dar. Auf dieser Basis gibt es viele Möglichkeiten des Vergleichs und der Manipulation, die wir Ihnen in einem späteren Abschnitt noch näher erklären werden.

Wahrheitswerte

Ein weiteres Format soll die Lesbarkeit sogenannter logischer Ausdrücke erleichtern. Eigentlich stellt dieses Format ein Byte zur Verfügung, in das nur zwei Werte gespeichert werden können. Einer der Werte steht für »Wahr« und der andere für »Falsch«.

Der Bezeichner für diesen Datentyp ist `bool`.

Repräsentation von Datentypen im Speicher

Nachdem wir nun eine Reihe von Datentypen kennengelernt haben, hier ein kleiner Ausflug dahin, wie die Repräsentation der Datentypen im Speicher erfolgen kann.

Es gibt hierbei zwei grundlegende Wege: Der eine Weg ist die direkte Abbildung von Daten im Speicher, wie es bei allen einfachen und kleinen Datentypen, wie `int` und `double`, aber auch bei `bool` der Fall ist. Auch der Datumstyp `DateTime` und ein einzelnes Zeichen `char` werden direkt im Speicher abgebildet. Man spricht hierbei von sogenannten Wertetypen.

Die zweite Art, Datentypen zu repräsentieren, erfolgt über Referenzen auf die eigentlichen Daten. Dies ist bei Datentypen, bei denen die tatsächliche Größe variabel oder unbestimmt ist und sehr groß werden kann, der Fall. Zu diesen Datentypen gehören `string`, aber auch `Arrays` und die benutzerdefinierten Datentypen. Diese werden Referenztypen genannt.

Konvertierung von Datentypen

In der Praxis werden Sie häufig Variablen von einem Datentyp in einen anderen umwandeln müssen. Das .NET Framework stellt hier eine Klasse zur Verfügung, die Ihnen eben diese Konvertierungen ermöglicht. Diese Methoden sind Teil der Klasse `System.Convert`.

Nachfolgend wollen wir uns genauer mit den Methoden der Klasse `System.Convert` beschäftigen, da diese sprachunabhängig sind und diese Klasse Teil der .NET-Basisklassenbibliothek ist.

Sie können mit diesen Methoden die in der nachfolgenden Tabelle aufgelisteten Datentypen ineinander umwandeln.

Falls der umzuwandelnde Wert außerhalb des Wertebereichs des Zieldatentyps liegt, tritt ein Laufzeitfehler (eine Ausnahme vom Typ `OverflowExeption)` auf, der per Ausnahmebehandlung abgefangen werden kann. Falls eine Umwandlung in den Zieldatentyp unmöglich ist, tritt eine Ausnahme vom Typ `InvalidCastException` oder `FormatException` auf. Auf Ausnahmen und ihre Behandlung gehen wir später noch detaillierter ein.

Zieldatentyp	Methode	Kommentar
`bool`	`erg= Convert.ToBoolean (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Bei Strings sind dies die Werte »True« und »False«. Einige Datentypen wie Variablen vom Typ `DateTime` lassen sich nicht in `boolean` umwandeln und erzeugen eine `InvalidCastException`.
`char`	`erg= Convert.ToChar (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Bei der Konvertierung von einem String in ein Zeichen wird nur das erste Zeichen des Strings umgewandelt.
`sbyte`	`erg= Convert.ToSByte (wert)`	Zahlenwerte können sich zwischen –128 und 127 bewegen. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`byte`	`erg= Convert.ToByte (wert)`	Zahlenwerte können sich zwischen 0 und 255 bewegen. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`short`	`erg= Convert.ToInt16 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`int`	`erg= Convert.ToInt32 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`long`	`erg= Convert.ToInt64 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps. Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`ushort`	`erg= Convert.ToUInt16 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.

Zieldatentyp	Methode	Kommentar
`uint`	`erg= Convert.ToUInt32 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`ulong`	`erg= Convert.ToUInt64 (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps (nur positive Zahlen). Falls der Wert Nachkommastellen besitzt, werden diese gerundet.
`float` `single`	`erg= Convert.ToSingle (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`double`	`erg= Convert.ToDouble (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`decimal`	`erg= Convert.ToDecimal (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`DateTime`	`erg= Convert.ToDateTime (wert)`	Erlaubter Wertebereich ist der des Zieldatentyps.
`string`	`erg= Convert.ToString (wert)`	Je nach umzuwandelndem Datentyp wird ein String zurückgeliefert, der den booleschen Wert, ein nach lokalen Einstellungen des Servers generiertes Datum oder einen eine Zahl repräsentierenden String enthält.

Tabelle 3.2: Datentypkonvertierung

Um also einen Wert von `boolean` in `int` umzuwandeln, können Sie beispielsweise die nachfolgenden Zeilen schreiben (wir geben außerdem die ermittelten Werte noch zusätzlich aus).

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int varInt;
    bool varBool;
    varBool = false;
    varInt = Convert.ToInt32(varBool);
    Response.Write("Wert von VarBool: ");
    Response.Write(varBool);
    Response.Write("<br>Ausgabe von VarInt: ");
    Response.Write(varInt);
    varBool = true;
    varInt = Convert.ToInt32(varBool);
    Response.Write("<br>Wert von VarBool: ");
    Response.Write(varBool);
    Response.Write("<br>Ausgabe von VarInt: ");
    Response.Write(varInt);
  }
</script>
```

Listing 3.2: Typkonvertierung von Datentypen (TypKonv.aspx)

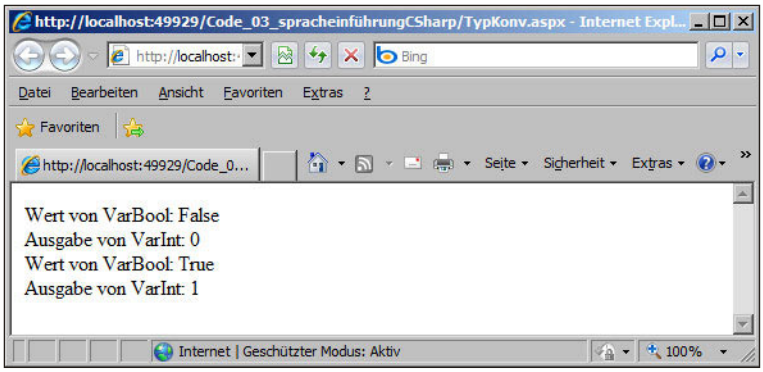


Abbildung 3.9: Datentypkonvertierung

HINWEIS

Bei der Konvertierung von Datentypen sollten Sie beachten, dass es Situationen geben kann, in denen Informationen bei der Konvertierung verloren gehen könnten.

Relativ unproblematische Konvertierungen sind diejenigen, die von einem kleineren zu einem umfangreicheren Datentyp durchgeführt werden. Eine Konvertierung von `short` zu `int` ist ein solches Beispiel. Diese Konvertierungen werden auch implizit durchgeführt.

Wandeln Sie Datentypen von einem Datentyp mit einem großen Wertebereich in einen Datentyp mit einem kleineren Wertebereich um, so ist dies möglich, es besteht aber die Gefahr, dass die Daten nicht verarbeitet werden können.

Natürlich ist es unproblematisch, den Wert einer `int`-Variablen in einen `short`-Wert umzuwandeln, solange der Wert der Variablen sich im erlaubten Wertebereich für `short` bewegt. Um solche Umwandlungen durchführen zu können, müssen Sie also zum Zeitpunkt der Umwandlung bereits eine Reihe von Informationen über den Inhalt der Variablen haben.

Kommen wir nun zur Problembehandlung bei der Datentypkonvertierung: Die Datenkonvertierung in C# wird durch drei Ausnahmeklassen unterstützt, die Sie auch im Nachhinein abfangen können.

Mit der Ausnahme `OverflowException` fangen Sie ab, dass Sie zu große Werte in Werte umwandeln, die vom Zieldatentyp nicht dargestellt werden können.

Die Ausnahme `InvalidCastException` fängt ab, wenn Sie bei Umwandlungen von `float` oder `double` nach `Decimal` Probleme haben. Die ursprüngliche Zahl lässt sich nicht als Dezimalzahl darstellen, der Ursprungswert ist unendlich oder er kann durch `decimal` nicht dargestellt werden.

Ein anderer Fall, in dem die Ausnahme `FormatException` auftritt, ist, wenn eine explizite Konvertierung nicht durchgeführt werden kann, weil für diese Konvertierung kein Datentyp definiert ist. Beispielsweise wenn Sie einen `string` nach `bool` umwandeln wollen und der String keinen booleschen Wert beschreibt.

Im nachfolgenden Beispiel werden Ihnen Typkonvertierungen und die Ausnahmebehandlung bei diesen Konvertierungen demonstriert.

Wir haben hier die Form der strukturierten Ausnahmebehandlung mittels der Kontrollstruktur von `try ...catch` ausgewählt. Diese Kontrollstruktur und weitere werden wir später noch genauer betrachten.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int Var1;
    byte Var2 = 0;
    Var1 = 1233;
    try
    {
      Var2 = Convert.ToByte(Var1);
    }
    catch (OverflowException ex)
    {
       Response.Write("Der Wert von Var1 ist groesser als Byte");
       Response.Write("<br>daher wird er nicht zugewiesen");
    }
    Response.Write("<br>Var1: ")
    Response.Write(Var1)
    Response.Write("<br>Var2: ")
    Response.Write(Var2)
  }
</script>
```

Listing 3.3: Typkonvertierung mit strukturierter Ausnahmebehandlung (TypkonvError.aspx)

Wir machen in diesem Programm eine Zuweisung zu einem Wert, der eigentlich kein Byte-Datentyp sein kann, da der zulässige Wertebereich überschritten wäre. Der Versuch der Zuweisung erzeugt eine Ausnahme, die wir entsprechend abgefangen haben. Statt der Zuweisung geben wir einen Fehlertext aus. Um zu zeigen, dass die Zuweisung tatsächlich nicht stattgefunden hat, geben wir die Werte der beiden Variablen im Nachhinein aus. Die Variable `Var2` wurde dabei mit 0 vorinitialisiert, da ansonsten der Compiler einen Fehler meldet, da nicht sichergestellt ist, dass tatsächlich eine Zuweisung erfolgt.

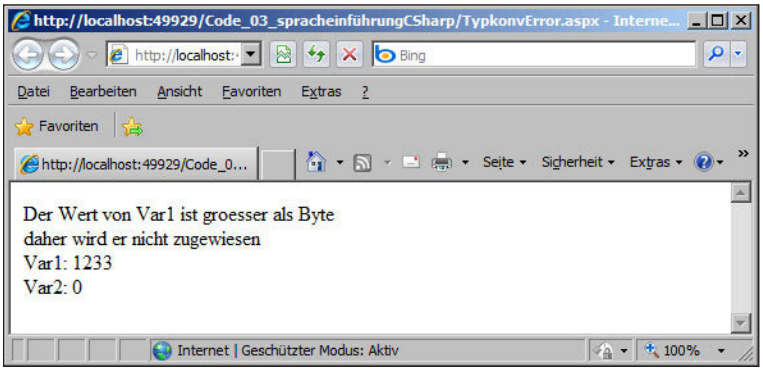


Abbildung 3.10: Datentypkonvertierung mit einfacher Ausnahmebehandlung

Wir werden uns des Themas Fehler- und Ausnahmebehandlung im Absatz 3.7 noch einmal ausführlich annehmen.

3.3.2 Operatoren

Operatoren stellen grundlegende Werkzeuge zur Manipulation von Dateninhalten dar. Diese lassen sich auf die vorgestellten Standarddatentypen anwenden, genauso wie auf die später vorgestellten Datentypen, die im .NET Framework zur Verfügung stehen.

Operatoren zur Zahlenmanipulation

Zahlen lassen sich durch die vier Grundrechenarten sowie die Potenzierung manipulieren. Bei der Division stehen Ihnen neben einem einfachen Divisionsoperator, der den Bruchteil einer Zahl ermittelt, Operatoren zur Verfügung, die Ihnen eine Integer-Division mit Restermittlung ermöglichen. Die nachfolgende Tabelle listet die wesentlichen Operatoren zur Zahlenmanipulation auf.

Für die Verwendung dieser Operatoren gibt es noch eine weitere Abwandlung, die (ähnlich wie es in C üblich ist) das Ergebnis gleich dem links vom Operator stehenden Operanden zuweist.

Für die Ermittlung eines Rests bei der Integer-Division ist kein eigenständiger Zuweisungsoperator vorgesehen.

Diese Zuweisungsoperatoren sind in der zweiten Spalte der Tabelle aufgelistet.

Operator	Zuweisungsoperator	Funktion
+	+=	Addition
-	-=	Subtraktion oder Vorzeichen
*	*=	Multiplikation
/	/=	Division. Handelt es sich bei beiden Operanden um Integerwerte, so wird eine Integer-Division durchgeführt.
%		Modulo: Rest einer Division (4 % 3 = 1)
%=		Modulo-Division

Tabelle 3.3: Operatoren zur Manipulation von Zahlen

Vergleichsoperatoren

Vergleichsoperatoren liefern als Ergebnis einen booleschen Wert, der als Kriterium für Verzweigungen oder für das Verlassen von Schleifenabläufen verwendet werden kann.

Nachfolgend sind die booleschen Operatoren, die in C# Verwendung finden, kurz aufgelistet und ihre Funktion beschrieben.

Operator	Funktion
==	Gleich
!=	Ungleich
<=	Kleiner oder gleich
>=	Größer oder gleich
<	Kleiner als
>	Größer als
!=	Prüfung, dass zwei Objektreferenzierungen nicht auf das gleiche Objekt verweisen
is	Prüfung, ob zwei Objektreferenzierungen auf das gleiche Objekt verweisen

Tabelle 3.4: Operatoren zum Vergleich

Auch hinter diesen Vergleichsoperatoren verbirgt sich nichts zusätzlich Besonderes, was Sie nicht aus anderen Programmiersprachen bereits kennen würden.

Logische Operatoren

Logische Operatoren stellen auf Binärebene eine wichtige Möglichkeit zur Datenmanipulation dar. Auch für Verzweigungen, die auf der Basis von mehreren verknüpften Bedingungen entstehen, sind logische Operatoren von großer Wichtigkeit.

Sie werden Und- und Oder-Verknüpfungen kennen (eine Und-Verknüpfung ist beispielsweise nur dann wahr, wenn beide Teilbedingungen wahr sind).

Exotischer sind Exklusives-Oder-Verknüpfungen oder etwa die Prüfung, ob eine Operation den logischen Wert wahr oder falsch ergeben hat. Die Not-Verknüpfung besitzt lediglich einen Operanden und invertiert seinen Wert.

Grundsätzlich stecken hinter diesen logischen Operatoren aber keine besonderen Geheimnisse, Sie müssen einfach die Ergebnisse der jeweiligen Operation kennen und entsprechend anwenden.

In der nachfolgenden Tabelle haben wir für Sie die wichtigsten logischen Operatoren beschrieben, also um welche Art von Operator es sich handelt, und die zugehörigen Wahrheitswerte aufgelistet.

Operator	Beschreibung	Operand 1	Operand 2	Ergebnis
&	Und-Verknüpfung	Falsch	Wahr	Falsch
		Falsch	Falsch	Falsch
		Wahr	Falsch	Falsch
		Wahr	Wahr	Wahr
|	Oder-Verknüpfung	Falsch	Falsch	Falsch
		Falsch	Wahr	Wahr

Operator	Beschreibung	Operand 1	Operand 2	Ergebnis
		Wahr	Falsch	Wahr
		Wahr	Wahr	Wahr
!	Nicht	Wahr		Falsch
		Falsch		Wahr
^	Exklusives Oder	Falsch	Falsch	Falsch
		Falsch	Wahr	Wahr
		Wahr	Falsch	Wahr
		Wahr	Wahr	Falsch
==	Äquivalent die Umkehrung von Xor	Falsch	Falsch	Wahr
		Falsch	Wahr	Falsch
		Wahr	Falsch	Falsch
		Wahr	Wahr	Wahr
&&	Kurzgeschlossene Und-Verknüpfung	Falsch	*Nicht bewertet*	Falsch
		Wahr	Wahr	Wahr
		Wahr	Falsch	Falsch
||	Kurzgeschlossene Oder-Verknüpfung	Wahr	*Nicht bewertet*	Wahr
		Falsch	Wahr	Wahr
		Falsch	Falsch	Falsch

Tabelle 3.5: Logische Operatoren und die ihnen zugeordneten Wahrheitswerte

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    string Var1 = "a";
    string Var3 = "1";
    Response.Write("Logik mit und ohne &&:<br>");
    Response.Write("Var 1 = a und Var3 = 1 ergibt:<br>");
    int Var1gewandelt;
    if ((Int32.TryParse(Var1, out Var1gewandelt) &&
         Convert.ToInt32(Var1) != 1))
    {
      Response.Write("<br>&&:True: Var1 ist Zahl und ungleich 1 ");
    }
    else
    {
```

```
  Response.Write(
     "<br>&&:False: Var1 ist keine Zahl und ungleich 1 ");
}
try
{
  if ((Int32.TryParse(Var1, out Var1gewandelt) &
       Convert.ToInt32(Var1) != 1))
  {
    Response.Write("<br>&:True: Var1 ist Zahl und ungleich 1");
  }
  else
  {
    Response.Write(
       "<br>&:False: Var1 ist keine Zahl und ungleich 1 ");
  }
}
catch (Exception e)
{
  Response.Write("<br>Var1 ist keine Zahl und nicht umwandelbar ");
  Response.Write("daher Cast-Fehler bei &<br>");
}
if (Int32.TryParse(Var1, out Var1gewandelt) &&
    Convert.ToInt32(Var1) == 1)
{
  Response.Write("<br>&&:True: Var1 ist Zahl und = 1");
}
else
{
  Response.Write("<br>&&:False: Var1 ist keine Zahl und = 1 ");
}
try
{
  if (Int32.TryParse(Var1, out Var1gewandelt) &
      Convert.ToInt32(Var1) == 1)
  {
    Response.Write("<br>&:True: Var1 ist Zahl und = 1");
  }
  else
  {
    Response.Write("<br>&:False: Var1 ist keine Zahl und = 1");
  }
}
catch (Exception e)
{
  Response.Write("<br>Var1 ist keine Zahl und nicht umwandelbar ");
  Response.Write("daher Cast-Fehler bei &<br>");
}
if (Int32.TryParse(Var3, out Var1gewandelt) &&
    Convert.ToInt32(Var3) != 1)
{
  Response.Write("<br>&&:True: Var3 ist Zahl und ungleich 1");
}
else
{
```

```
      Response.Write(
        "<br>&&:False: Var3 ist keine Zahl und ungleich 1 ");
    }
    try
    {
      if (Int32.TryParse(Var3, out Var1gewandelt) &
          Convert.ToInt32(Var3) != 1)
      {
        Response.Write("<br>&:True: Var3 ist Zahl und ungleich 1");
      }
      else
      {
        Response.Write(
          "<br>&:False: Var3 ist keine Zahl und ungleich 1");
      }
    }
    catch (Exception e)
    {
      Response.Write("<br>Var3 ist keine Zahl und nicht umwandelbar");
      Response.Write("<br>daher Cast-Fehler bei &");
    }
    if (Int32.TryParse(Var3, out Var1gewandelt) &&
        Convert.ToInt32(Var3) == 1)
    {
      Response.Write("<br>&&:True: Var3 ist Zahl und = 1");
    }
    else
    {
      Response.Write("<br>&&:False: Var3 ist keine Zahl und = 1");
    }
    try
    {
      if (Int32.TryParse(Var3, out Var1gewandelt) &
          Convert.ToInt32(Var3) == 1)
      {
        Response.Write("<br>&:True: Var3 ist Zahl und = 1");
      }
      else
      {
        Response.Write("<br>&:False: Var3 ist keine Zahl und = 1");
      }
    }
    catch (Exception e)
    {
      Response.Write("<br>Var3 ist keine Zahl und nicht umwandelbar");
      Response.Write("<br>daher Cast-Fehler bei &");
    }
  }
</script>
```

Listing 3.4: Logik mit & und && (Logik.aspx)

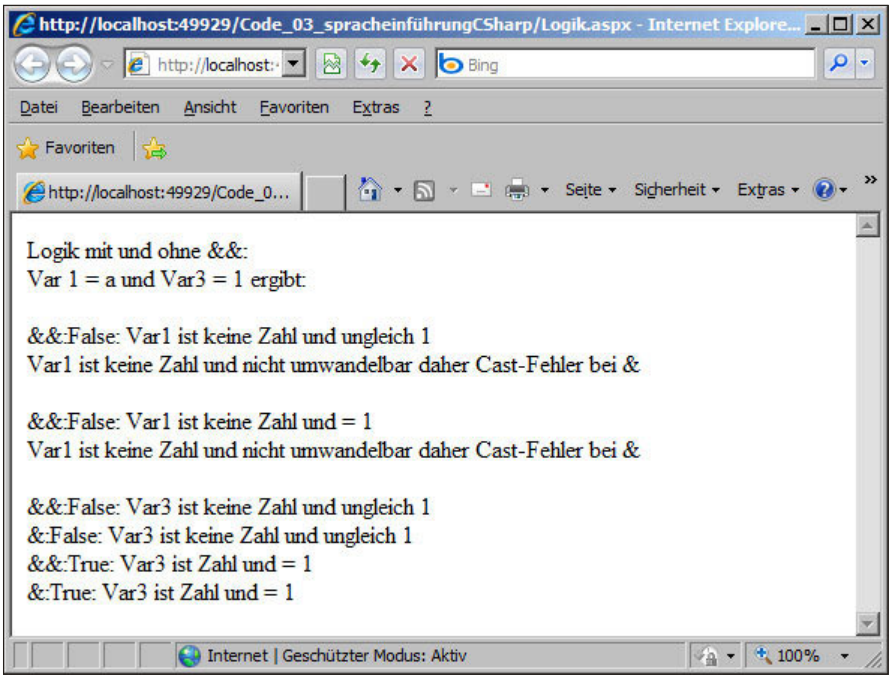


Abbildung 3.11: Logik mit &- und &&-Operator

Ternärer Operator

Mit dem `?:`-Operator steht in C# ein ternärer Operator zur Verfügung. Bei einer Abfrage mit `?:` wird einer von zwei möglichen Werten zurückgegeben. Im nachfolgenden Beispiel wird überprüft, ob eine Variable `i` einen Wert größer als 5 besitzt. Ist das der Fall, wird das zweite Argument zurückgegeben, ansonsten das dritte Argument.

```
string s  = i>5 ? "groß": "klein";
```

Sonstige Operatoren

Neben den oben beschriebenen mathematischen und logischen Operatoren gibt es noch eine Reihe weiterer Operatoren, die eine gewisse Bedeutung haben. Diese sind nachfolgend im Einzelnen aufgelistet und kurz beschrieben.

Zunächst betrachten wir zwei Operatoren, die nah an Assembler angelehnt sind, die sogenannten Bitshift-Operatoren, mit denen in einer Variablen die Inhalte bitweise nach links oder rechts verschoben werden. Bei einem Integerwert entspricht die Verschiebung um eine Stelle einer Multiplikation mit zwei (bei der Verschiebung nach links) oder einer Division durch zwei (bei der Verschiebung nach rechts), unter der Voraussetzung, dass das Ergebnis noch im gültigen Bereich liegt. Ansonsten ist das Ergebnis negativ.

In der nachfolgenden Grafik haben wir anhand eines Byte-Werts die Wirkungsweise der Bitshift-Verschiebung illustriert.

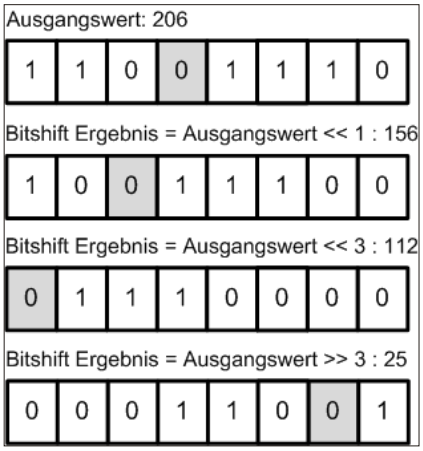


Abbildung 3.12: Illustration von Bitshift-Operationen

Diese Operatoren sind nur auf Integer-Variablen anwendbar, der zweite Operand legt fest, um wie viele Bits die Verschiebung zu erfolgen hat.

Einen weiteren einfachen, aber sehr nützlichen Operator stellt derjenige dar, mit dem Zeichenketten miteinander verknüpft werden. Dieser wird durch + repräsentiert.

Operator	Funktion	Beispiel
<<	Bitshift nach links	`Erg = Wert << 3`
>>	Bitshift nach rechts	`Erg = Wert >> 2`
+	Verknüpfung von Zeichenketten	`A = B + C`

Tabelle 3.6: Sonstige Operatoren

Wir sollten zum Abschluss noch ein paar Worte über die Hierarchien von Operationen und Klammerung verlieren.

Grundsätzlich gilt in der Hierarchie der Ausführung, dass eine Operation höherer Ordnung vor einer Operation niederer Ordnung ausgeführt wird. Punktrechnung geht also vor Strichrechnung. Durch die Verwendung von Klammern, die Sie beliebig schachteln können, können Sie diese Hierarchie, wie in der Schulmathematik gelernt, aufheben.

Die Verwendung von Klammern empfiehlt sich auch, wenn Sie Rechenoperatoren mit logischen Operatoren mischen wollen, da es die Lesbarkeit einer Formel erheblich erhöhen kann.

3.3.3 Strukturierte Datentypen

Wie mit dem Standarddatentyp `string` schon angedeutet, werden Sie häufig grundlegende Datentypen miteinander verknüpfen wollen, um die benötigten Daten in einfachen Strukturen vorzuhalten. Somit erschaffen Sie neuartige Klassen von Datentypen.

Nachfolgend werden wir Ihnen einige grundlegende strukturierte Datentypen vorstellen.

Sie basieren auf den Möglichkeiten des .NET Frameworks und werden in gleichartiger Form damit auch von anderen Sprachen, die sich dieses Frameworks bedienen, verwendet. Durch die Definition dieser Klassen werden gleich die Möglichkeiten zur Manipulation der Datentypen mitgeliefert.

Arrays

Arrays sind eine Möglichkeit, beispielsweise die Standarddatentypen, die wir bereits kennengelernt haben, zu Wertegruppen zusammenzufassen. Hierbei werden die Daten in einer Art Liste zusammengefasst und können quasi über einen Index ausgewählt, manipuliert und angezeigt werden.

Die Deklaration und die Initialisierung eines Arrays erfolgen, wie Sie es auch schon bei einfachen Variablentypen gelernt haben, durch den vor den Arraynamen gestellten Datentyp gefolgt von einem eckigen Klammernpaar:

```
string[] Arg1;
```

Vor der ersten Verwendung muss das Array jedoch initialisiert werden. Dies geschieht mit folgender Anweisung:

```
Arg1 = new string[5];
```

Wir haben in diesem Beispiel also ein Array mit fünf Elementen vom Datentyp `string` deklariert. Die Definition und Initialisierung hätten natürlich auch kompakter in einer einzigen Zeile geschrieben werden:

```
string[] Arg1 = new string[5];
```

Das erste Element ist `Arg1[0]`. Einzelne Elemente innerhalb des Arrays können nun direkt zugewiesen werden:

```
Arg1[2] = "Dritter Wert";
```

Nun können Sie entweder auf einzelne Elemente des Arrays zugreifen, diese Elemente verändern, austauschen oder löschen.

Wie bei normalen Variablendefinitionen können Sie auch bei Arrays bereits bei der Definition die Werte der einzelnen Variablen initialisieren.

```
string[] Arg1 = new string[5] {"A", "B", "C", "D", "E"};
```

Auf eine Größenangabe des Arrays hätten Sie hier verzichten können, da die Größe aufgrund der Anzahl der Initialwerte ermittelt wird. Noch kompakter wäre folgende Schreibweise gewesen, bei der nur die Werte zugewiesen werden:

```
string[] Arg1 = {"A", "B", "C", "D", "E"};
```

Eine weitere Besonderheit bei Arrays stellt die Möglichkeit dar, mehrdimensionale Datenfelder zu erzeugen. Die Deklaration eines solchen Datentyps geschieht durch eine kommagetrennte Angabe der Größe der jeweiligen Dimension. Hier ein kleines Beispiel zur Veranschaulichung:

```
string[,,] Arg1 = new string[2,2,2];
```

Mit dieser Deklaration haben wir ein dreidimensionales Objekt erzeugt, in dem Zeichenketten abgelegt sind. Die maximale »Breite«, »Höhe« und die »Tiefe« sind angegeben. Sie können in diesem Datenarray acht Elemente (2*2*2 Elemente) ablegen.

Dieser Datentyp stellt bereits einen Typ dar, der als Teil des .NET Frameworks bereitgestellt wird.

Es gibt einige weitere Sprachelemente, welche die Handhabung eines Arrays erleichtern. Mittels der Methode `GetLowerBound()` können Sie für die jeweils angegebene Dimension die mögliche Untergrenze ermitteln. Mittels `GetUpperBound()` ermitteln Sie die Obergrenze der jeweiligen Dimension.

Datumstypen

Datum und Uhrzeit stellen einen strukturierten Datentyp dar. Die Elemente dieses strukturierten Datentyps lassen sich auf unterschiedliche Weise ausgeben und anzeigen. Zum einen kann die Anzeige (ähnlich wie auch schon bei Gleitkommazahlen, wo in Deutschland das Dezimalzeichen ein Komma ist, in angelsächsischen Ländern aber ein Punkt) länderspezifisch sein, zum anderen können Sie Daten auf unterschiedliche Weise darstellen (Sie können Wochenendtage mit angeben oder weglassen, Monatsbezeichnungen in verschiedener Weise ausgeben oder Ähnliches).

Eine Datumsanzeige kann in langer Form mit ausgeschriebenem Monatsnamen und Wochentag erfolgen oder auch in kurzer Form, als einfache Aneinanderreihung von Zahlen. Sie können die Informationen genauer halten oder auch Details weglassen (wie die Uhrzeit oder die Sekunden einer Uhrzeit).

In der nachfolgenden Tabelle ist einmal beispielhaft eine Reihe unterschiedlicher möglicher Datumsformate ein und desselben Datums aufgelistet:

Datum	Uhrzeit
25.2.2008	1:23
25. Februar 2008	1:23:45
25/02/08	01:23
25. Feb. 08	1h 23 min 45 sec

Tabelle 3.7: Verschiedene Darstellungen von Datums- und Uhrzeitformaten

In C# wird als Standarddatentyp für ein Datum und eine Uhrzeit der Datentyp `DateTime` zur Verfügung gestellt. Wenn Sie einer Variablen vom Datentyp `DateTime` einen Wert direkt zuweisen wollen, so müssen Sie dazu eine neue Instanz einer `DateTime`-Struktur erstellen:

```
DateTime Geburtsdatum = new DateTime(1967,12,24);
```

In der Darstellung und der Auswertung des Datentyps `DateTime` gibt es eine Reihe von Funktionen, mit denen Sie beispielsweise das aktuelle Systemdatum und die aktuelle Systemzeit ermitteln können. Sie können auch einzelne Elemente dieses Datentyps mit verschiedenen Funktionen manipulieren oder auswerten.

Das .NET Framework stellt dabei die `DateTime`-Struktur zur Verfügung. Mit dieser Struktur erhalten Sie eine ganze Reihe von Methoden, um Daten oder Teile von Daten zu setzen, auszuwerten und in vielen möglichen unterschiedlichen Formen auch anzuzeigen.

In der nachfolgenden Tabelle sind einige wesentliche Eigenschaften zur Ermittlung von Elementen der `DateTime`-Struktur aufgelistet.

Eigenschaft	Beschreibung
`DateTime.Now`	Liefert die aktuelle Systemzeit und das Systemdatum.
`DateTime.Today`	Liefert das Systemdatum.
`DateTime.TimeOfDay`	Liefert die aktuelle Systemzeit.
`DateTime.Ticks`	Liefert die Anzahl an 100 Nanosekunden-Intervallen, die seit dem 1.1.0001 vergangen sind.
`DateTime.Second`	Liefert die Sekunde eines vorgegebenen Datums als Zahl.
`DateTime.Minute`	Liefert die Minute eines vorgegebenen Datums als Zahl.
`DateTime.Hour`	Liefert die Stunde eines vorgegebenen Datums als Zahl.
`DateTime.Month`	Liefert den Monat eines vorgegebenen Datums als Zahl.
`DateTime.Year`	Liefert das Jahr eines vorgegebenen Datums als Zahl.
`DateTime.DayOfWeek`	Liefert den Namen des Wochentags eines Datums.

Tabelle 3.8: Ermitteln von Datum, Uhrzeit und Elementen davon

Sie haben neben den Möglichkeiten, verschiedene Eigenschaften auszulesen, auch die Möglichkeit, an `DateTime`-Strukturen mit verschiedenen Methoden Manipulationen vorzunehmen. In der nachfolgenden Tabelle haben wir einige davon aufgelistet und erläutert.

Methode	Beschreibung
`DateTime.Parse()`	Mittels dieser statischen Methode können Sie eine Zeichenkette in eine Struktur vom Typ `DateTime` umwandeln.

`DateTime.Add()`
`DateTime.AddTicks()`
`DateTime.AddSeconds()`
`DateTime.AddMinutes()`
`DateTime.AddHours()`
`DateTime.AddDays()`
`DateTime.AddMonths()`
`DateTime.AddYears()`
`DateTime.Subtract()`
	Diese Gruppe von Methoden können Sie dazu verwenden, um Daten innerhalb der `DateTime`-Struktur zu verändern, indem Sie Zeiteinheiten hinzufügen oder (durch Hinzufügen negativer Werte oder Verwendung von `DateTime.Subtract`) vermindern. Falls Sie außerhalb des zugelassenen Wertebereichs der `DateTime` gelangen sollten, wird eine `ArgumentOutOfRangeException` erzeugt.

`DateTime.ToString()`
`DateTime.ToLongDateString()`
`DateTime.ToLongTimeString()`
`DateTime.ToShortDateString()`
`DateTime.ToShortTimeString()`
	Mittels dieser Methoden wandeln Sie den Wert einer `DateTime`-Struktur in einen String um. Der String wird je nach gewählter Methode formatiert.

`DateTime.Compare()`	Mit dieser statischen Methode vergleichen Sie zwei `DateTime`-Strukturen miteinander. Das Ergebnis ist ein Integerwert, der wie folgt zu interpretieren ist: Wert < 0: Der erste `DateTime`-Wert ist kleiner als der zweite `DateTime`-Wert. Wert = 0: Beide `DateTime`-Werte sind gleich. Wert > 0: Der erste `DateTime`-Wert ist größer als der zweite `DateTime`-Wert.
`DateTime.IsLeapYear()`	Statische Methode zur Ermittlung, ob das übergebene Jahr als Schaltjahr definiert ist.

Tabelle 3.9: Manipulation und Vergleich von Datum, Uhrzeit und Elementen davon

Weiterhin sind mathematische Operatoren für Addition und Subtraktion für die `DateTime`-Struktur definiert. Ebenso können Sie auch die Vergleichsoperatoren verwenden.

Wenn Sie Additionen und Subtraktionen durchführen, ist das Ergebnis ein Wert in der Struktur `TimeSpan`. Diese Struktur ist grundsätzlich ähnlich aufgebaut wie die Struktur `DateTime`. Ein wesentlicher Unterschied liegt darin, dass die Struktur `DateTime` einen Zeitpunkt repräsentiert, die Struktur `TimeSpan` aber einen Zeitraum.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    DateTime Arg1 = new System.DateTime();
    DateTime Arg2 = new System.DateTime();
    TimeSpan Arg3 = new System.TimeSpan();
    Arg1 = System.DateTime.Parse("24.01.1969 05:32:12");
    Arg2 = System.DateTime.Now;
    Response.Write("Einige Datumsausgaben:");
    Response.Write("<br>Datum: ");
    Response.Write(Arg1.ToLongDateString());
    Response.Write("<br>Jetzt: ");
    Response.Write(Arg2.ToLongDateString());
    Response.Write("<br>Stunde: ");
```

```
    Response.Write(Arg1.Hour.ToString());
    Response.Write("<br>Minute: ");
    Response.Write(Arg1.Minute.ToString());
    Response.Write("<br>Sekunde: ");
    Response.Write(Arg1.Second.ToString());
    Response.Write("<br>Jahr: ");
    Response.Write(Arg1.Year.ToString());
    Response.Write("<br>Monat: ");
    Response.Write(Arg1.Month.ToString());
    Response.Write("<br>Tag: ");
    Response.Write(Arg1.Day.ToString());
    Response.Write("<br>Datum: ");
    Response.Write(Arg1.ToString());
    Response.Write(
       "<br>Zeitdifferenz zwischen Jetzt und Datum (als TimeSpan):");
    Arg3 = Arg2 - Arg1;
    Response.Write(Arg3.ToString());
  }
</script>
```

Listing 3.5: Anzeige verschiedener Datumsformate (DateTime.aspx)

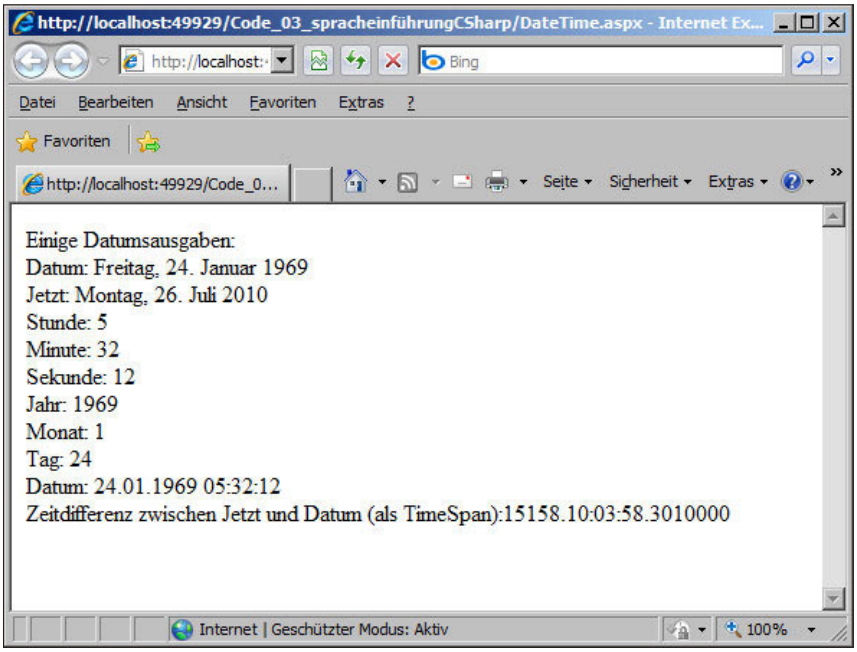


Abbildung 3.13: Ausgabe verschiedener Datumsformate und Elemente

3.3.4 Wertetyp- und Referenztypsemantik

Wie bereits erwähnt, unterscheiden sich Variablen in zwei Gruppen, die Wertetypen und die Referenztypen. Bei Wertetypen handelt es sich dabei um die folgenden Datentypen:

» byte

» short

» int

» long

» ushort

» uin

» ulong

» float

» double

» decimal

» DateTime

» char

» bool

» und sonstige Strukturen

Bei den Wertetypen wird der Wert der Variablen direkt an dieser Speicherstelle im Stack gespeichert.

Zu den Referenztypen gehören folgende Typen:

» string

» und sonstige andere als Klassen implementierte komplexe Objekte

Bei den Referenztypen wird an der Speicherstelle im Stack auf eine Adresse im Heap referenziert. Referenztypen werden somit wie Zeiger auf einen anderen Speicherbereich implementiert.

Doch Wertetypen und Referenztypen unterscheiden sich nicht nur in der Art der Speicherung, sondern auch in der Art und Weise, wie Daten zugewiesen werden.

Wertetypsemantik

Bei der Zuweisung eines Wertetyps wird eine Kopie der entsprechenden Variablen gemacht. Wird die zugewiesene Variable manipuliert, hat das keine Auswirkungen auf die Originalvariable.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int Arg1= 7;
    int Arg2 = Arg1;
    Arg2 = 9;
    Response.Write("Arg1: " + Arg1.ToString() + "<br>");
    Response.Write("Arg2: " + Arg2.ToString());
  }
</script>
```

Listing 3.6: Beispiel für Wertetypsemantik (WerteTypSemantik.aspx)

Im Beispiel in Listing 3.6 wird eine Variable `Arg1` vom Typ `int` definiert und dabei der Wert 7 zugewiesen. Im nächsten Schritt wird eine zweite Integer-Variable `Arg2` definiert und ihr wird der Wert von `Arg1` zugewiesen. Der Wert von `Arg1` wird aufgrund der Wertetypsemantik in die Speicherzelle `Arg2` kopiert. Anschließend wird der Wert von `Arg2` auf 9 gesetzt. Wenn beide Variablen danach ausgegeben werden, dann haben sie auch unterschiedliche Werte, einmal den Wert 7 für `Arg1` und 9 für `Arg2`. Sie sehen, dass bei der Zuweisung `Arg2 = Arg1` eine Kopie des Werts durchgeführt wurde.

Referenztypsemantik

Bei der Zuweisung eines Referenztyps hingegen wird keine Kopie der entsprechenden Variablen gemacht. Vielmehr wird die Adresse im Heap, die auf das tatsächliche Objekt zeigt, kopiert und somit zeigen beide Variablen auf denselben Speicherbereich. Wird die zugewiesene Variable manipuliert, hat das Auswirkungen auf die Originalvariable.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    Person Arg1 = new Person();
    Arg1.Name = "Christian";
    Person Arg2 = Arg1;
    Arg2.Name = "Tobias";
    Response.Write(Arg1.Name.ToString() + "<br>");
    Response.Write(Arg2.Name.ToString());
  }
</script>
```

Listing 3.7: Beispiel für Referenztypsemantik (ReferenzTypSemantik.aspx)

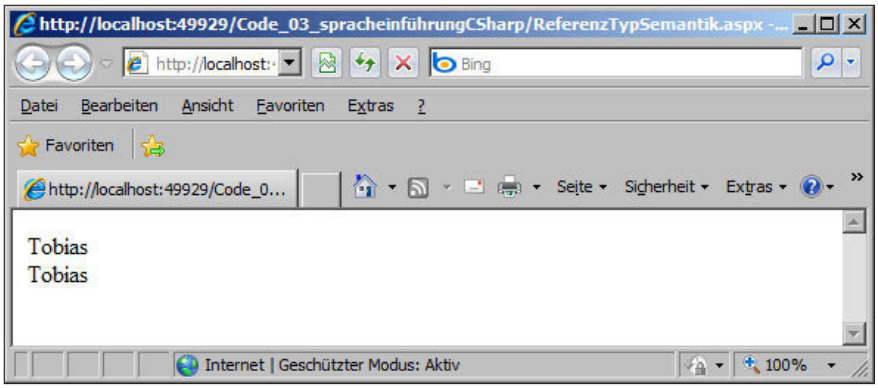


Abbildung 3.14: Ausgabe Referenztypsemantik

Im Beispiel in Listing 3.7 wird eine Variable `Arg1` vom Typ `Person` (ohne der Objektorientierung vorgreifen zu wollen, sehen Sie die Definition dieses Typs in Listing 3.8) definiert und instanziiert. Dann wird der Eigenschaft `Name` der Wert »Christian« zugewiesen. Im nächsten Schritt wird eine zweite Personenvariable `Arg2` definiert und ihr wird `Arg1` zugewiesen. Dabei wird die Adresse von `Arg1` aufgrund der Referenztypsemantik in die Speicherzelle `Arg2` kopiert. Anschließend wird die Name-Eigenschaft von `Arg2` auf »Tobias« gesetzt. Wenn von beiden Variablen danach die Name-Eigenschaft ausgegeben wird, dann haben sie identische Werte, nämlich »Tobias«. Sie sehen, dass bei der Zuweisung `Arg2 = Arg1` eine Kopie der Adresse und nicht des Werts durchgeführt wurde.

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

Listing 3.8: Listing 3.8: Definition einer Klasse Person (Person.cs)

ACHTUNG

Obwohl es sich beim Datentyp `string` um einen Referenztyp handelt, verhält er sich wie ein Wertetyp. `string` ist somit ein Referenztyp mit Wertetypsemantik. Die Ausnahme wurde von den Framework-Entwicklern gemacht, um die Verarbeitung dieses doch wesentlichen Datentyps intuitiver zu gewährleisten.

Nullable Typen

Nicht lokale Wertetypen werden bei der Definition mit einem Standardwert versehen, sofern Sie nicht direkt einen Initialwert zuweisen. Für sämtliche Zahlenwerte ist das der Wert 0, für den Datentyp `bool` der Wert `false` und für den Datentyp `DateTime` der 01.01.0001.

Das bedeutet, dass Wertetypen niemals den Wert `null` annehmen können. Gerade bei Datenbankanwendungen ist es jedoch sinnvoll, Spalten, die den Wert `null` (nicht belegt) besitzen dürfen, nicht mit einem Initialwert zu versorgen. Stellen Sie sich vor, in einer beliebigen Tabelle gibt es ein Feld Geburtsdatum. Wenn der Wert nicht bekannt ist, wird eben nichts eingetragen. Der Wert in der Datenbank ist `null` und somit weiß jeder, dass diese Information nicht bekannt ist. Wird

jedoch der Initialwert eingetragen, hätten diese Personen stattdessen am 01.01.0001 Geburtstag. Peinlich für denjenigen, der in seiner Anwendung eine Serienbrieffunktionalität besitzt, die den Personen dann zum 2010. Geburtstag gratuliert.

Deswegen wurden bereits in der Version 2.0 des Frameworks sogenannte *Nullable Types* eingeführt. Das bedeutet, dass Wertetypen jeweils ein Nullable-Äquivalent besitzen, das auch den Wert `null` annehmen kann.

Einen **Nullable Integer** definieren Sie dabei wie folgt:

```
int? Arg1;
```

3.3.5 Kontrollstrukturen und Schleifen

Kontrollstrukturen und Schleifen stellen zwei grundlegende Elemente einer Programmiersprache dar.

Bei einer Kontrollstruktur wird auf der Basis von vorher ermittelten Daten eine Entscheidung für den weiteren Programmablauf getroffen.

Durch Schleifen wird sichergestellt, dass bestimmte Abschnitte mehrfach zu durchlaufen sind, bis ein bestimmter Status erreicht ist.

Weiterhin gibt es noch den Sprungbefehl als primitive Variante einer Schleife. Was ist ein Sprungbefehl? Ein Programm ist üblicherweise ein Block, der aus mehreren Programmzeilen besteht. Ein Sprungbefehl veranlasst das Programm, zu einer bestimmten Stelle zu springen und die Ausführung von dieser Stelle aus weiter fortzuführen.

Diese wichtigen Elemente der Programmierung wollen wir Ihnen in den nächsten Abschnitten näher bringen.

Im Wesentlichen werden von C# zwei grundlegende Arten von Kontrollstrukturen zur Verfügung gestellt. Es gibt sie in unterschiedlichen Varianten, aber diese Abwandlungen dienen dem Zweck, die Lesbarkeit Ihres Programmcodes zu erhöhen, denn Sie können die durch die Variante erzielte Besonderheit auch auf einem anderen Weg mit einer anderen der bereitstehenden Kontrollstrukturen erreichen.

Die Kontrollstruktur if ... else

Die Abfrage eines Zustands – sei es der Inhalt einer Variablen, das Abfragen eines bestimmten Systemstatus oder der Vergleich zweier Datenstrukturen – und die anschließende Weiterverarbeitung aufgrund des Ergebnisses dieser Prüfung stellen ein wesentliches Element für die Erstellung von Programmen dar.

Die Syntax für diese Statusabfragen lautet in C#:

```
if (Bedingung)
{
  Anweisung(en)
}
```

Sie können Ihre Anweisungen um einen alternativen Anweisungsblock erweitern. Es wird dann entweder der eine oder der andere Block ausgeführt und anschließend mit dem normalen Programmcode fortgefahren. Hierfür ergänzen Sie Ihre `if`-Bedingung um einen `else`-Block.

Die Syntax für diesen Bedingungsblock sieht dann wie folgt aus.

```
if (Bedingung)
{
  Anweisung(en)
}
else
{
  Anweisung(en)
}
```

Wenn Sie den Visual Web Developer als Editor verwenden, geschieht ein Einrücken zum optischen Absetzen des vom Bedingungsblock begrenzten Programmcodes automatisch. Wenn Sie einen Editor verwenden, mit dem dies nicht möglich ist, sollten Sie aus Gründen der Übersichtlichkeit eine solche Strukturierung manuell vornehmen. Sollte der Visual Web Developer diese automatische Formatierung einmal nicht durchführen, dies geschieht zumeist, wenn ein Compilerfehler vorliegt, dann können Sie die automatische Formatierung mit der Tastenkombination STRG+K, STRG+D durchführen lassen (Voraussetzung ist natürlich, dass keine Compilerfehler im Code vorhanden sind).

Noch ein kleiner Tipp am Rande, wenn Sie nach einem C#-Schlüsselwort die Taste ↹ zweimal drücken, dann wird im Visual Web Developer sofort der gesamte Block für dieses Schlüsselwort automatisch angelegt.

Dies gilt auch für die weiteren Elemente, die Ihren Programmcode strukturieren.

```
< %@ Page Language="C#" %>
<script runat="server">
void Page_Load()
  {
    Random ZufallsObj = new Random();
    double Zufall1;
    double Zufall2;
    Zufall1 = ZufallsObj.Next();
    Zufall2 = ZufallsObj.Next();
    Response.Write("If...Else");
    Response.Write(" <br> ");
    Response.Write("Zufallszahl1: ");
    Response.Write(Zufall1.ToString());
    Response.Write(" <br> ");
    Response.Write("Zufallszahl2: ");
    Response.Write(Zufall2.ToString());
    Response.Write(" <br> ");
    if (Zufall1 > Zufall2)
    {
      Response.Write("Erste Zufallszahl grösser");
      Response.Write(" <br> ");
    }
    else
    {
      Response.Write("Zweite Zufallszahl grösser");
```

```
      Response.Write(" <br> ");
    }
  }
</script>
```

Listing 3.9: Eine Bedingungsabfrage mit `if ... else` und generierten Zufallszahlen (IfElse.aspx)

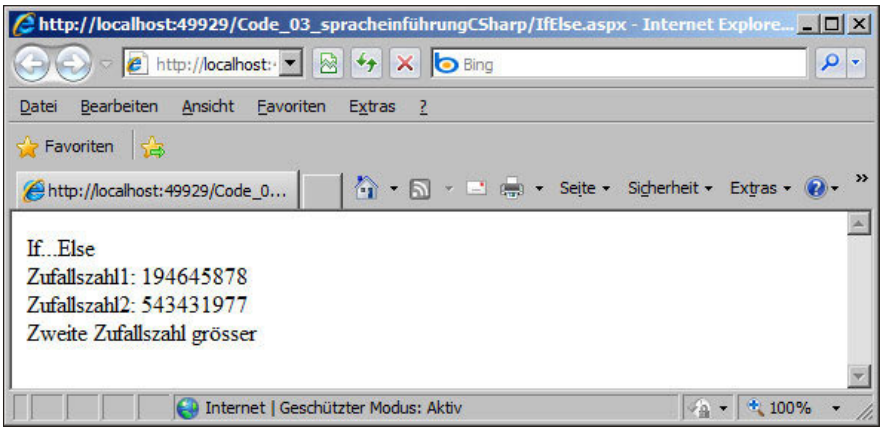


Abbildung 3.15: Eine Bedingungsabfrage mit If ... Else und generierten Zufallszahlen

Noch eine kleine ergänzende Anmerkung: In diesem Beispiel haben wir die `Random`-Klasse des .NET Frameworks verwendet, um zwei Zufallszahlen zu erzeugen, die verglichen wurden. Wir haben zunächst ein `Random`-Objekt angelegt und mittels der Methode `Random.Next()` dann die Zufallszahlenerzeugung durchgeführt.

Sie können zwischen dem `if`- und dem `else`-Block optional noch weitere `else if`-Blöcke einbauen.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    Random ZufallsObj = new Random();
    double Zufall1 = ZufallsObj.NextDouble();
    Response.Write("If...ElseIf...Else");
    Response.Write(" <br> ");
    Response.Write("Zufallszahl1: ");
    Response.Write(Zufall1.ToString());
    Response.Write(" <br> ");
    if (Zufall1 < 0.3)
    {
      Response.Write("kleine Zufallszahl");
      Response.Write(" <br> ");
    }
    else if (Zufall1 < 0.6)
    {
        Response.Write("mittlere Zufallszahl");
```

```
        Response.Write(" <br> ");
    }
    else
    {
        Response.Write("große Zufallszahl");
        Response.Write(" <br> ");
    }
  }
</script>
```

Listing 3.10: Eine Bedingungsabfrage mit If-ElseIf-Else (IfElseIfElse.aspx)

Die Verwendung mehrerer `else if`-Blöcke macht Ihren Code jedoch sehr schnell übersichtlich. Die bessere Alternative dazu sehen Sie im folgenden Abschnitt.

Die Kontrollstruktur switch

Eine weitere Kontrollstruktur, mit der Sie einen ganzen Satz von Zuständen prüfen und bearbeiten können, bilden die nachfolgenden Bedingungsblöcke.

Mit der Kontrollstruktur `switch` haben Sie die Möglichkeit, quasi eine Reihe von `if...else if... else`-Abfragen gebündelt ausführen zu lassen. Die Syntax für diese Kontrollstruktur ist wie folgt:

```
switch (Wert)
{
  case Ergebnis1:
    Anweisungen
    break;
  case Ergebnis2
    Anweisungen
    break;
  case Ergebnis3
    Anweisungen
    break;
  default:
    Anweisungen
    break;
}
```

Der `default`-Zweig ist auch in diesem Fall optional. Das Schlüsselwort `break` verhindert ein Durchfallen in den nächsten Zweig.

Wenn Sie also den Inhalt einer Variablen auf verschiedene Werte überprüfen und auf Basis dieser unterschiedlichen Werte Anweisungsketten ausführen wollen, können Sie dies mit der `switch`-Kontrollstruktur entsprechend durchführen.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int Arg1 = Convert.ToInt32(System.DateTime.Now.DayOfWeek);
    Response.Write("Der Wochentag ist: ");
    Response.Write("<br> ");
    switch (Arg1)
```

```
    {
       case 1:
         Response.Write("Montag");
         Response.Write("<br> ");
         break;
       case 2:
         Response.Write("Dienstag");
         Response.Write("<br> ");
         break;
       case 3:
         Response.Write("Mittwoch");
         Response.Write("<br> ");
         break;
       case 4:
         Response.Write("Donnerstag");
         Response.Write("<br> ");
         break;
       case 5:
         Response.Write("Freitag");
         Response.Write("<br> ");
         break;
       case 6:
         Response.Write("Samstag - Wochenende!");
         Response.Write("<br> ");
         break;
       case 0:
         Response.Write("Sonntag - Wochenende!");
         Response.Write("<br> ");
         break;
    }
  }
</script>
```

Listing 3.11: Die Kontrollstruktur Switch ermittelt den aktuellen Wochentag (Switch.aspx).

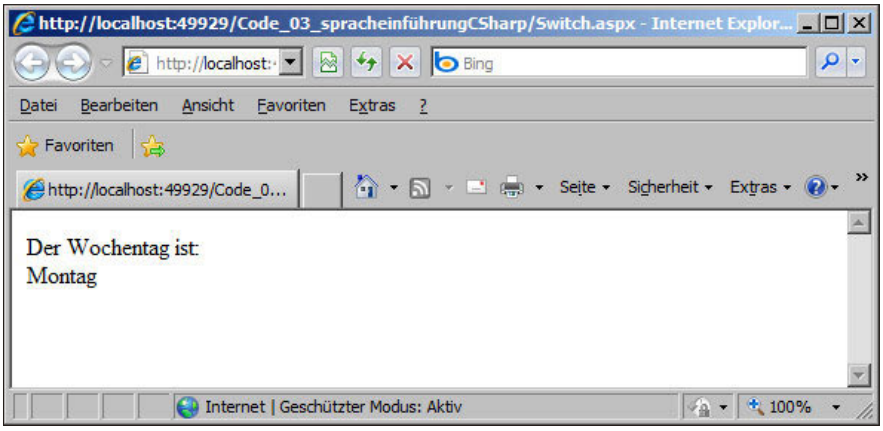


Abbildung 3.16: Die Kontrollstruktur Switch ermittelt den aktuellen Wochentag (Switch.aspx).

Schleifen und unbedingte Sprungbefehle

In der Programmierung haben Sie immer wieder Abschnitte, die mehrfach durchlaufen werden müssen, oder Abschnitte, die strukturell gleichartig sind und sich nur von den Inhalten der Variablen unterscheiden. Als prozedurale Programmierelemente stellt Ihnen C# hierfür verschiedene Schleifenstrukturen und Sprungbefehle zur Verfügung.

Der Goto-Befehl als unbedingter Sprung

Der `goto`-Befehl ist ein sehr mächtiger, aber auch primitiver Befehl. Im Prinzip wird zur Verwendung dieses Sprungbefehls eine Stelle im Programmcode markiert. Zu dieser Markierung springen Sie dann, indem Sie den Befehl `goto Sprungmarke` aufrufen.

Die Gefahr bei der Verwendung des `goto`-Sprungbefehls liegt darin, dass er Programme sehr unübersichtlich und schwer lesbar machen kann (Stichwort Spaghetticode). Der von Ihnen erstellte Code wird schon bald nach Fertigstellung vermutlich nicht einmal für Sie nachvollziehbar sein.

Ihnen stehen unter C# eine ganze Reihe alternativer Programmierelemente zur Verfügung, die der Verwendung des `goto`-Befehls vorzuziehen sind.

Die for-Schleife

Ein besseres Mittel für die Programmierung von Schleifen ist beispielsweise die `for`-Schleife, die einen Zähler mitführt, der sicherstellen kann, dass die Schleife wieder verlassen werden kann, wenn dieser Zähler einen bestimmten Wert erreicht hat.

Die Syntax einer `for`-Schleife ist die nachfolgende:

```
for (Variableninitialisierung; Bedingung; Inkrement)
{
  Anweisungen
}
```

Anbei ein Beispiel für einen Schleifenkopf, der eine Schleife zehnmal wiederholen lässt:

```
for (int i = 1; i < 11; i++)
```

Die `for`-Schleife können Sie übrigens vorzeitig verlassen, indem Sie im Anweisungsblock die Anweisung `break` einbauen.

Schleifen können Sie beliebig ineinander schachteln. Ein Überkreuzen von zwei Schleifen ist – logisch nachvollziehbar – aber nicht möglich.

```
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load()
  {
    int Arg1 = 0;
    Response.Write("Schleife mit for");
    Response.Write(" <br> ");
    for (double Arg2 = 2; Arg2 < 40.1; Arg2+=3.4)
      {
      Arg1 = Arg1 + 1;
```

```
        Response.Write("Argument in Durchlauf ");
        Response.Write(Arg1.ToString());
        Response.Write(": ");
        Response.Write(Arg2.ToString());
        Response.Write(" <br> ");
        if (Arg1 > 10)
        {
            break;
        }
      }
      Response.Write("Programm Fertig ");
    }
</script>
```

Listing 3.12: Eine for-Schleife (For.aspx)

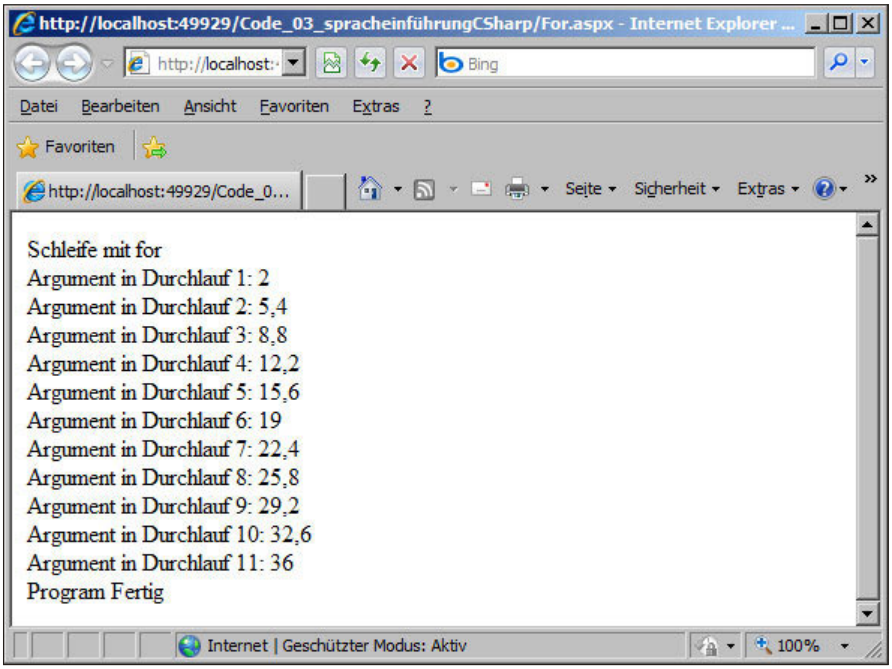


Abbildung 3.17: Eine For-Schleife (For.aspx)

Die foreach-Schleife

Sehr ähnlich zur `for`-Schleife verhält sich die `foreach`-Schleife. Bei dieser Art von Schleife durchlaufen Sie aufzählbare Elemente wie zum Beispiel Arrays oder Collections (Auflistungsobjekte). Deswegen brauchen Sie auch keine Schleifenzähler mitzugeben, da diese aufzählbaren Elemente vom ersten bis zum letzten Eintrag automatisch durchlaufen werden.

Die Syntax einer `foreach`-Schleife ist die nachfolgende:

```
foreach (Datentyp Element in Liste)
{
  Anweisungen
}
```

Die `foreach`-Schleife können Sie übrigens genauso wie die `for`-Schleife mit `break` vorzeitig verlassen.

Auch `foreach`-Schleifen können Sie beliebig ineinander schachteln, jedoch nicht überkreuzen.

```
<%@ Page Language="C#" %>
<script runat="server">
void Page_Load()
  {
    int[] Arg1 = {1 , 2, 3, 4, 5};
    Response.Write("Schleife mit foreach");
    Response.Write(" <br> ");
    foreach (int Arg2 in Arg1)
    {
      Response.Write("Argument in Durchlauf ");
      Response.Write(Arg2.ToString() + "<br>");
    }

    Response.Write("Program Fertig ");
  }
</script>
```

Listing 3.13: Eine foreach-Schleife (ForEach.aspx)

Die While-Schleife

Mit der `While`-Schleife haben Sie eine weitere Möglichkeit, eine Schleifenstruktur aufzubauen. Hier können Sie bei jedem Schleifendurchlauf eine Bedingung auf ihren Wahrheitswert überprüfen.

Die Syntax einer `While`-Schleife ist nachfolgend beschrieben.

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

In diesem Falle wird der Wahrheitswert der Bedingung am Anfang der Schleife (kopfgesteuerte Schleife) überprüft.

Alternativ haben Sie die Möglichkeit, die Überprüfung der Bedingung an das Ende der Schleife zu stellen (fußgesteuerte Schleife). Dann stellt sich die Syntax der Schleife in der Form `Do...While` wie folgt dar:

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

Diese Schleife wird jetzt so lange durchlaufen, wie die `Bedingung` den logischen Wert Wahr besitzt, mindestens aber einmal.

Auch bei der `while`-Schleife haben Sie die Möglichkeit, diese mit dem »Ausstiegsbefehl« `break` vorzeitig zu verlassen. Eine typische Schleifenstruktur hätte also die folgende Form:

```
while (Bedingung)
{
  Anweisungen
  if (Bedingung2)
  {
    break;
  }
  Anweisungen
}
```

Nachfolgend auch hier noch einmal ein Listing mit der zugehörigen Bildschirmausgabe.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    string Ket = "Test";
    string CatKet = "";
    string KetErg = "TestTestTestT";

    Response.Write("Eine Schleife mit While <br>");
    Response.Write("KetErg:");
    Response.Write(KetErg);
    while (CatKet != KetErg)
    {
      Response.Write("<br>CatKet sieht jetzt so aus:");
      Response.Write(CatKet);
      CatKet = CatKet + Ket;
      if (CatKet.Length > KetErg.Length)
      {
        Response.Write("<br>Konnte KetErg nicht bauen <br>");
        break;
      }
    }
    Response.Write("Abschlusswert von CatKet:");
    Response.Write(CatKet);
    Response.Write("<br>Teil 1 Fertig <br>");
    CatKet = "";
    do
    {
      Response.Write("<br>CatKet sieht jetzt so aus:");
      Response.Write(CatKet);
      CatKet = CatKet + Ket;
      if (CatKet.Length > KetErg.Length)
      {
        Response.Write("<br>Konnte KetErg nicht bauen <br>");
        break;
      }
```

```
    } while (CatKet != KetErg);

    Response.Write("Abschlusswert von CatKet:");
    Response.Write(CatKet);
    Response.Write("<br>Teil 2 Fertig <br>");
  }
</script>
```

Listing 3.14: Die While-Schleife (*While.aspx*)

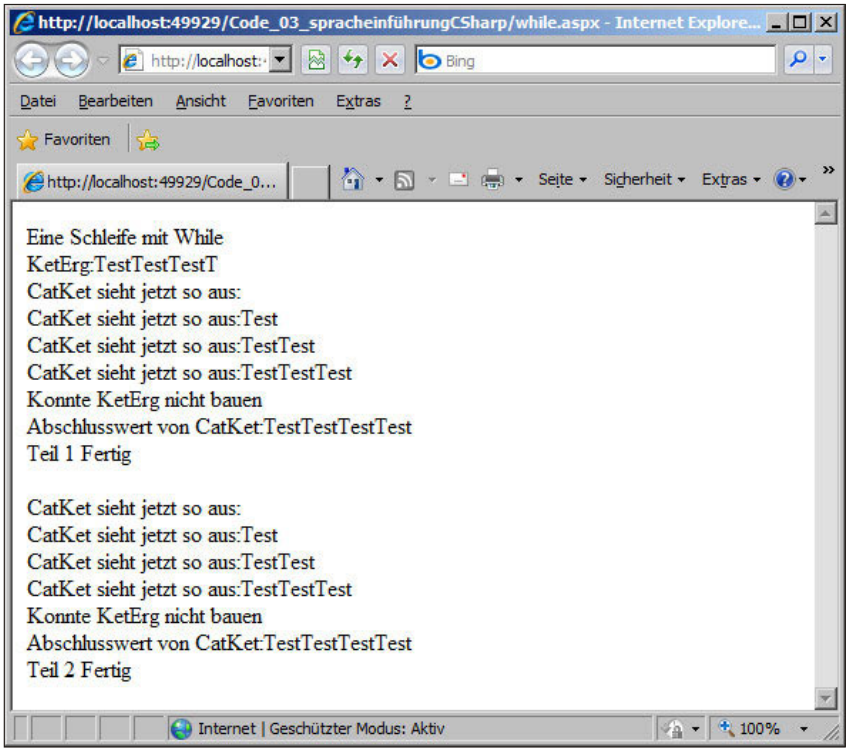


Abbildung 3.18: Die While-Schleife

3.4 Programmelemente und Programmebenen

Nachdem Sie nun verschiedene grundlegende Programmelemente für die C#-Programmierung kennengelernt haben, ist es an der Zeit, Ihnen einen Überblick über die Strukturierung von Programmen zu geben.

Sie haben die Möglichkeit, diverse Programmelemente zusammenzufassen und in Dateien abzuspeichern.

Grundsätzlich können Sie diese Dateien über sogenannte Assemblies in Ihren zu übersetzenden Quellcode importieren lassen und auf diese Weise verschiedene Codeelemente zusammenstellen. Um die Programmelemente möglichst einfach zusammenstellen zu können, bietet C# ein ganzes Reich von Programmelementen an, die Ihren Code weiter modular strukturieren.

Hier sind neben prozeduralen Programmelementen auch die objektorientierten Strukturen zu nennen. Zunächst möchten wir Ihnen Funktionen und Prozeduren als prozedurale Programmelemente vorstellen. In einem zweiten Schritt werden wir dann die erweiterten Elemente der objektorientierten Programmierung ergänzen, die auch auf Funktionen anwendbar sind.

3.4.1 Funktionen

Mit Funktionen erhalten Sie die Möglichkeit, Programmabschnitte zu gliedern und diese einzelnen Gliederungselemente in andere Programme zu übernehmen, einfach indem Sie die Funktion übernehmen. Sie müssen bei der Erstellung von Funktionen, die Sie in andere Programme übernehmen wollen, einige Dinge berücksichtigen, auf die wir nachfolgend eingehen werden.

Mit der objektorientierten Programmierung haben Methoden wesentliche Aufgaben von Funktionen übernommen, in ihrer Programmierung sind beide Elemente sehr ähnlich, so dass sich Funktionen einfach in Methoden umwandeln lassen, mit denen eine noch bessere Wiederverwendbarkeit ermöglicht wird.

Programmierung von Funktionen

Funktionen sind kleine Unterprogramme, die über den von Ihnen vergebenen Funktionsnamen aufgerufen werden und ein Funktionsergebnis als Rückgabewert zurückliefern. Sie haben die Möglichkeit, eine Liste von Parametern zu definieren, die an diese Funktion übergeben werden, mit denen die Funktion dann arbeiten kann.

In einer Funktion haben Sie die Möglichkeit, lokale Variablen einzusetzen. Dies sind Variablen, die im globalen Programmumfeld (außerhalb der Funktion) unbekannt sind und nicht verwendet werden. Sie sollten es vermeiden, den Gültigkeitsbereich von Variablen zu groß zu wählen und ihn im Wesentlichen auf diese lokalen Variablen innerhalb von Funktionen zu beschränken. Ansonsten wäre die Wiederverwendbarkeit des Codes stark eingeschränkt.

ACHTUNG

Eine lokale Variable, die innerhalb eines Blocks (Schleife, Kontrollstruktur etc.) definiert wird, ist auch nur innerhalb dieses Blocks gültig, man spricht dann von sogenannten blockweiten Variablen. Außerhalb des Blocks sind diese Variablen nicht zugänglich, eine Verwendung führt zu einem Compiler-Fehler.

In der Konsequenz heißt das: Verwenden Sie möglichst lokal deklarierte Variablen. Verzichten Sie so weit wie möglich auf Variablen, die übergreifend über verschiedene Programmabschnitte bekannt sein müssen.

Wenn Sie von einer aufrufenden Ebene Daten an eine Funktion weitergeben wollen, bieten sich hierfür die in einer Funktion deklarierbaren Übergabeparameter an. Hierbei erfolgt eine Wertübergabe an die Übergabeparameter, die innerhalb der Funktion wie lokale Variablen zu verwen-

den sind. Außerhalb der Funktion sind diese deklarierten Übergabeparameter wie auch die sonstigen dort deklarierten lokalen Variablen nicht gültig.

Sie haben die Möglichkeit, für jeden Parameter festzulegen, ob er durch die Funktion verändert werden kann oder ob eben dies nicht möglich ist. Standardmäßig werden Parameter *ByValue* (Übergabe einer Kopie des Wertes, keine Manipulation möglich) übergeben, die Alternative wäre *ByReference*. Wenn Sie einen Parameter als Referenz übergeben wollen, so setzen Sie das Schlüsselwort `ref` (Übergabe der Referenz auf die Variable, Manipulation möglich) vor die Variable. In C# müssen Sie dass sowohl beim Funktionskopf als auch bei allen Aufrufen machen.

Das Schlüsselwort `out` besitzt eine sehr ähnliche Bedeutung wie `ref`, nur dass bei `out` die Variable übergeben wird, die nicht vorinitialisiert werden muss.

Funktionen ohne Rückgabewert (in Visual Basic würden Sie von Prozeduren sprechen) werden mit dem Rückgabewert `void` definiert.

Eine Funktion in C# wird durch die nachfolgende Syntax aufgerufen:

```
void Funktionsname([ref|out] Datentyp Parameter, [ref|out] Datentyp Parameter2 , ...)
{
  Anweisungen
}
```

Sie können beliebig viele oder auch keine Parameter deklarieren und übergeben.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    string Arg1 = "Testtext";
    int Arg2 = 0;
    string Arg3 = "Hallo";
    Response.Write("Aufruf einer void-Funktion<br>");
    Response.Write("Einige Variablen vor dem Funktionsaufruf:");
    Response.Write("<br>Arg1:");
    Response.Write(Arg1);
    Response.Write("<br>Arg2:");
    Response.Write(Arg2.ToString());
    Response.Wite("<br>Arg3:");
    Response.Write(Arg3);
    Response.Write(" <br> ");
    Prozedur(Arg1, ref Arg3);
    Response.Write("<br>Variablen nach void-Funktionsaufruf:");
    Response.Write("<br>Arg1:");
    Response.Write(Arg1);
    Response.Write("<br>Arg2:");
    Response.Write(Arg2.ToString());
    Response.Write("<br>Arg3:");
    Response.Write(Arg3);
    Response.Write(" <br> ");
    Response.Write("Programm Fertig ");
  }
```

```
  void Prozedur(string Arg2, ref string Arg3)
  {
    Response.Write("<br>An Funktion übergeben:");
    Response.Write(" <br>Arg2:");
    Response.Write(Arg2);
    Response.Write(" <br>Arg3:");
    Response.Write(Arg3);
    Arg2 = "Hallo2";
    Arg3 = "Hallo3";
    Response.Write("<br>Neuzuweisung der Variablen <br>");
    Response.Write("Arg2:");
    Response.Write(Arg2);
    Response.Write("<br>Arg3:");
    Response.Write(Arg3);
    Response.Write("<br>Funktion Ende<br>");
  }
</script>
```

Listing 3.14: Eine Funktion wird aufgerufen (voidFunction.aspx).

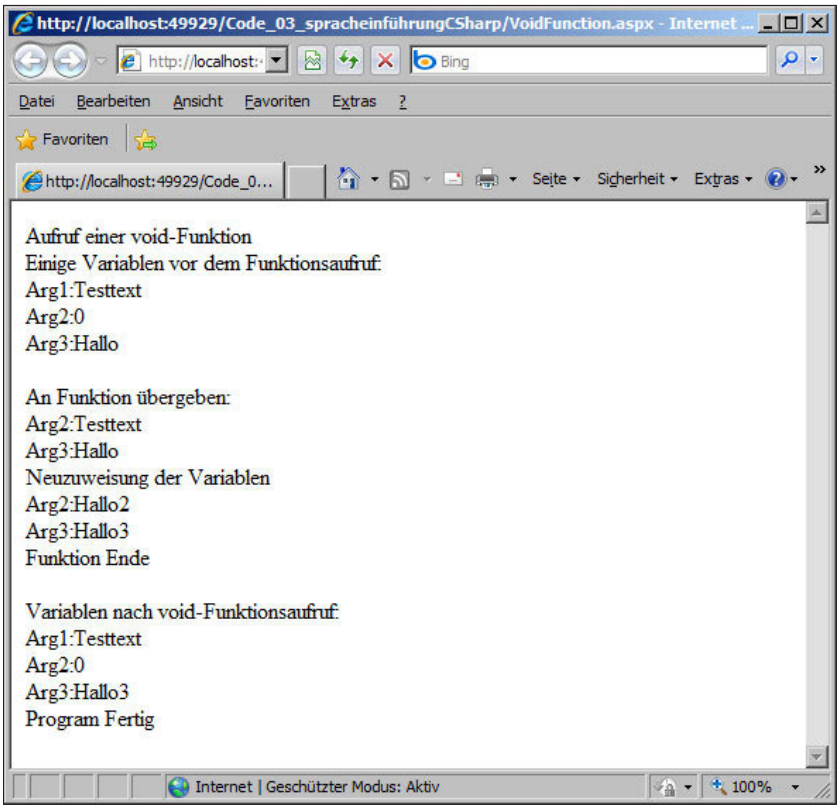


Abbildung 3.19: Eine Funktion wird aufgerufen.

Weiterhin haben Sie die Möglichkeit, eine Funktion auch vorzeitig über `Return` zu verlassen. Dies ist beispielsweise sinnvoll, wenn eine bestimmte Bedingung bereits vor Ende der Funktion eingetreten ist.

HINWEIS

Sie können per Schlüsselwort noch definieren, welchen Gültigkeitsbereich die Funktion besitzt. Die entsprechenden Schlüsselwörter und Bedeutungen sind im nachfolgenden Abschnitt zur Objektorientierung erklärt. In Tabelle 3.10 finden Sie diese Informationen zusammengefasst.

Kommen wir nun zu Funktionen mit Rückgabewerten. Da der Rückgabewert auch ein strukturierter Datentyp sein kann, haben Sie die Möglichkeit, innerhalb von Funktionen komplexe Datenstrukturen aufzubauen und an den aufrufenden Programmabschnitt zurückzuliefern.

Eine Funktion in C# kann also beispielsweise die folgende Syntax haben:

```
Rückgabedatentyp  Funktionsname(Datentyp Parameterliste,…)
{
  Anweisungen
  return Rückgabewert;
}
```

Die Verwendung und Deklaration der Übergabeparameter erfolgen analog zu void-Funktionen. Sie müssen nur statt des Schlüsselworts `void` den Datentyp, den die Funktion besitzt, definieren.

In dem obigen Beispiel wird explizit über das Schlüsselwort `return` sowie einem Wert der Rückgabewert angegeben und unmittelbar die Funktion verlassen.

Die Funktion wird im Hauptprogramm aufgerufen:

```
Ergebnis = Funktionsname(Parameter1,…)
```

Nun noch ein vollständiges Programm:

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int Erg = 0;
    Response.Write("Aufruf einer Funktion");
    Response.Write(" die eine ganzzahlige Zufallszahl berechnet.<br> ");
    Response.Write(
       "Die Obergrenze (9) wird als Parameter mitgegeben.<br>");
    Erg = Zufallsinteger(9);
    Response.Write(" Folgende Zahl wurde ermittelt: ");
    Response.Write(Erg);
  }

  int Zufallsinteger(int Grenze)
  {
    Random r = new Random();
    return r.Next(Grenze);
  }
</script>
```

Listing 3.15: Die Verwendung einer Funktion (Function.aspx)

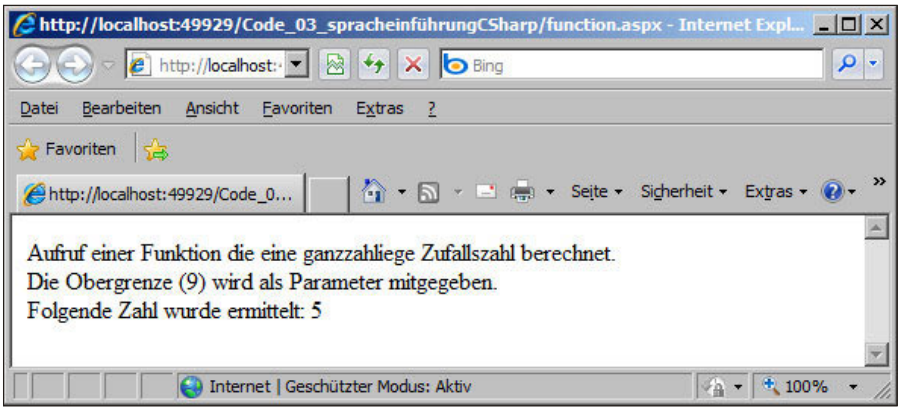


Abbildung 3.20: Die Verwendung einer Funktion

Grundsätzlich ist es sinnvoll, sowohl in Funktionen als auch in Prozeduren ausschließlich mit lokalen Variablen zu arbeiten und Datenübergaben nur über Parameterlisten und übergebene Datenstrukturen durchzuführen.

Diese Vorgehensweise ermöglicht Ihnen eine erheblich einfachere Wiederverwendung von Funktionen in anderen Programmen.

HINWEIS

Auch bei Funktionen können Sie durch vorangestellte Schlüsselwörter wie `public` oder `private` die Zugriffsberechtigungen steuern. Im nachfolgenden Abschnitt gehen wir genauer darauf ein. Die Schlüsselwörter und ihre Bedeutung sind in Tabelle 3.10 zusammengefasst.

3.4.2 Objektorientierung

Mit der Objektorientierung werden bestimmte Programmelemente gekapselt und mit Eigenschaften und Logiken (Methoden) versehen. Diese können in anderen Programmabschnitten weiter verwendet werden.

In gewisser Weise stellen sie damit einen weiteren Schritt nach Funktionen dar, um die Wiederverwendbarkeit von Code zu verbessern.

Klassen

Klassen stellen die Baupläne von den daraus erstellten Objekten dar. In einer Klasse werden also alle wesentlichen Elemente festgelegt, die das daraus hervorgehende Objekt beschreiben. Das Objekt kann dabei ein konkretes Programm sein, aber auch der Repräsentant einer Datenstruktur, die definiert wird.

Nachfolgend stellen wir Ihnen die wesentlichen Elemente vor, die es bei der Definition von Klassen zu beachten gibt.

Die Zugriffsrechte auf und die Sichtbarkeit von Programmcode bilden ein wesentliches Konzept der objektorientierten Programmierung. Dieses Konzept wird mit dem Schlagwort Kapselung von Code beschrieben.

Grundsätzlich stellt die Kapselung von Programmabschnitten auch außerhalb der objektorientierten Programmierung ein wesentliches Strukturmittel dar, das die Lesbarkeit und die Wiederverwendbarkeit Ihrer Programmblöcke vereinfachen oder gar erst ermöglichen kann.

Sie fassen mittels Kapselung verschiedene Elemente von Methoden und Eigenschaften zu einer Gruppe zusammen. Teil dieser Gruppe können durchaus auch eine oder mehrere andere Klassen sein. Diese Elemente befinden sich damit in einer definierten Klasse und bilden nach außen eine Einheit. Sie können beispielsweise auch steuern, ob Informationen und Abläufe dieser Einheit von außen frei zugänglich sind oder Einschränkungen unterliegen oder ob Sie gar jeden Zugriff von außen verwehren wollen.

In der nachfolgenden Tabelle sind die unterschiedlichen Zugriffsebenen kurz aufgelistet und erklärt.

Zugriffsbezeichner	Bedeutung
`public`	Dieses Klassenelement ist öffentlich zugänglich, es gibt keinerlei Zugriffsbeschränkungen nach außen.
`private`	Dieses Klassenelement kann nur innerhalb des Kontextes, in dem es deklariert wurde, verwendet werden. Dies bedeutet, dass Sie innerhalb einzelner Module, in denen die Deklaration erfolgt ist, freien Zugriff haben, aber nicht außerhalb.
`protected`	Die Elemente dieser Klasse können nur aus der Klasse, innerhalb derer sie deklariert wurden, verwendet werden oder in einer daraus abgeleiteten (vererbten) Klasse.
`internal`	Die Elemente der Klasse sind nur in derselben Assembly oder demselben Modul zugänglich. Dieses stellt die Standarddeklaration dar, wenn kein Bezeichner angegeben wurde.
`protected internal`	Die Elemente der Klasse sind nur aus der Klasse, in der sie deklariert wurden, zugänglich (oder aus daraus abgeleiteten Klassen) oder aus demselben Modul. Es stellt die Vereinigungsmenge der `internal`- und der `protected`-Zugriffsberechtigungen dar.

Tabelle 3.10: Zugriffsberechtigungsebenen von Klassen, Methoden, Eigenschaften und Ähnlichem

HINWEIS

Diese Zugriffsberechtigungsebenen können Sie übrigens auch bei Funktionen verwenden, die damit innerhalb einer Klasse mit den entsprechenden Berechtigungsstrukturen ausgestattet werden. Dies ist eine Erweiterung des klassischen prozeduralen Gedankens. Auch Funktionen sind damit objektorientierte Elemente. Sie sind die Methoden des objektorientierten Universums.

Ein weiteres Konzept stellt die Vererbung von Code, Eigenschaften oder Aktionen dar. Die Definition einer Klasse kann auf der Definition einer bereits vorhandenen Klasse aufbauen. Die neue Klasse übernimmt hierbei die Merkmale der Basisklasse. Die Übernahme der Merkmale der Basisklasse bezeichnet man als Vererbung.

Die neue Klasse ist aus der Basisklasse abgeleitet worden. Sie kann zusätzliche Elemente enthalten oder auch ein bereits bestehendes Element durch ein anderes Element ersetzen. Dieses Verhalten nennt man Überschreiben.

Wir werden diese Elemente in den nachfolgenden Abschnitten zu Methoden und Eigenschaften noch kennenlernen.

Die Nutzung der Vererbung bietet sich an, wenn es Klassen gibt, die konzeptionell eine Spezialisierung einer Basisklasse darstellen.

Schlüsselwort	Bedeutung
:	Mit dieser Anweisung wird die Basisklasse angegeben, aus der die aktuelle Klasse abgeleitet wird. (Class2:Class1)
`abstract`	Diese Klasse ist abstrakt, und die Elemente der Klasse müssen durch vererbte (abgeleitete) Klassen implementiert sein. Die damit gekennzeichnete Klasse ist also zwingend als Basisklasse gekennzeichnet, von der andere Klassen abzuleiten sind.
`sealed`	Diese Klasse darf nicht vererbt (abgeleitet) werden. Diese Klasse ist damit also nicht als Basisklasse verwendbar.

Tabelle 3.11: Schlüsselwörter, die im Zusammenhang mit Vererbung stehen

Kommen wir noch einmal auf das Überschreiben von Elementen einer Klasse zurück. Es existieren eine Reihe von Schlüsselwörtern, die klassifizieren, ob die Elemente überschreibbar sind oder nicht. Diese haben wir in der nachfolgenden Tabelle aufgeführt.

Außerdem haben wir noch die Überladung mit aufgelistet, obwohl es dafür in C# gar kein Schlüsselwort gibt. Überladung ist die Technik, eine Programmeinheit (z. B. eine Methode) mehrfach mit demselben Namen innerhalb einer Klasse zu erstellen. Die Methoden sollten sich nicht in der bereitgestellten Funktionalität unterscheiden, sondern diese Funktionalität auf Parameter mit unterschiedlichen Datentypen anwenden, die korrekte Methode wird anhand der übergebenen Datentypen ausgewählt. Sie können außerdem auch die Anzahl der Parameter zusätzlich variieren.

Bevor wir nun endgültig zur angekündigten Tabelle kommen, noch ein paar Worte zum sechsten Schlüsselwort, das sich dort wieder findet. Das Schlüsselwort `new` steht nicht nur für die Instanzierung von Klassen, sondern ermöglicht auch Modifikationen abgeleiteter Klassen aus einer Basisklasse.

Schlüsselwort	Bedeutung
`Überladen (kein Schlüsselwort vorhanden)`	Gruppe gleichnamiger Methoden, die sich in Anzahl und Typ der verwendeten Parameter unterscheiden können
`override`	Die deklarierte Methode oder Eigenschaft überschreibt eine gleichnamige Methode oder Eigenschaft aus einer Basisklasse. Die Parameter der neu deklarierten Methode oder Eigenschaft müssen in ihren Datentypen und der Anzahl der Parameter mit denen der überschriebenen Methode oder Eigenschaft übereinstimmen.

Schlüsselwort	Bedeutung
`virtual`	Diese Methode oder Eigenschaft kann durch die Methode oder Eigenschaft einer abgeleiteten Klasse überschrieben werden.
`abstract`	Diese Methode oder Eigenschaft muss von einer Methode oder Eigenschaft in einer abgeleiteten Klasse überschrieben werden, damit diese Klasse erzeugt werden kann. Dies ist nur bei einer abstrakten Basisklasse zulässig. Damit wird sichergestellt, dass bei nicht implementierten Klassen – also Klassen ohne Code – der notwendige Code in der abgeleiteten Klasse existiert.
`sealed`	Diese Methode oder Eigenschaft darf nicht von einer abgeleiteten Methode oder Eigenschaft überschrieben werden. Dies ist die Standardbelegung. Wenn eine überschreibbare Methode oder Eigenschaft nicht erneut überschrieben werden können soll, dann verwenden Sie dieses Schlüsselwort. In der Basisklasse ist es nicht notwendig.
`new`	Neudefinition von Elementen der Basisklasse. Dieser Vorgang wird auch Verschattung genannt. Hierbei wird Polymorphie ausgeschaltet.

Tabelle 3.12: Überladung, Überschreibung und Verschattung

Die Deklaration einer Klasse erfolgt mit dem `class`-Schlüsselwort. In einer Klasse werden, wie schon erwähnt, bestimmte Programmabschnitte (oder Funktionen) deklariert, es können Datentypen deklariert, gruppiert und bereitgestellt werden.

Kurz: Die Struktur eines Codeblocks wird hier festgelegt. Formal müssen Sie einige weitere Punkte bei der Deklaration einer Klasse und ihrer Elemente in Hinblick auf deren Methoden und Eigenschaften beachten.

Eine Klasse wird wie nachfolgend beschrieben deklariert:

```
[Bezeichner] class Klassenname [:Basisklasse][: Interfacename]
{
  [Deklarationen und Unterroutinen]
}
```

Innerhalb einer Klasse können Sie nun eine Reihe von Unterroutinen bauen.

Mit diesen Rahmenparametern können Sie nun Klassen und ganze Bibliotheken von Klassen deklarieren.

Strukturierung von Klassen

Wenn Sie eine Reihe von Klassen erstellt haben, stellt sich schnell die Frage nach weiteren Strukturierungsmöglichkeiten dieser Klassen. In diesem Zusammenhang steht Ihnen als Strukturierungselement der `Namespace` zur Verfügung, in dem Sie mehrere Klassen zusammenfassen können.

Auch einen `Namespace` können Sie weiter hierarchisch gliedern, so dass Sie auch mehrere Namespaces (und Klassen) zu einem neuen `Namespace` zusammenfassen können. Es entstehen dabei Strukturen, die in etwa wie folgt dargestellt aussehen:

```
namespace Ebene1
{
  namespace Ebene2
  {
    class Klassenname
    {
    }
  }
  class Klassenname
  {
  }
}

namespace Ebene1
{
  class Klassenname
  {
  }
}
```

Auf diese Weise ist auch das .NET Framework organisiert.

Erzeugung von Objekten

Die objektorientierte Programmierung ermöglicht es Ihnen, dynamisch Objekte zu erzeugen, die nach der Verwendung dynamisch auch wieder freigegeben und verworfen werden. Die Erzeugung erfolgt auf der Basis der in der Klassendeklaration festgelegten Struktur. Die Deklarationen in einer Klasse stellen ja den Bauplan des Hauses dar, das durch die konkreten Elemente (z. B. eine gemauerte Treppe, die im Bauplan durch die Skizze einer Treppe symbolisch angelegt ist), also durch Objekte, in der Realität des Programmablaufs erzeugt wird.

Das Erzeugen von Objekten erfolgt dabei durch den Aufruf eines Konstruktor-Schlüsselworts. In der aktuellen Version von C# werden die Instanziierung im Speicher und die Erzeugung gemeinsam über das Schlüsselwort `new` durchgeführt.

Sie deklarieren also ein Objekt und geben ihm die Struktur der zu verwendenden Klasse und erzeugen anschließend dieses Objekt.

```
Beispielklasse Beispiel;
Beispiel = new Beispielklasse()
```

Schon haben Sie das Objekt `Beispiel`, das alle Methoden und Eigenschaften der `Beispielklasse` beinhaltet.

Der Konstruktor kann dabei auch überladen werden. Wenn Sie eine Anforderung haben, dass bestimmte Eigenschaften bereits bei der Objekterzeugung initialisiert werden müssen oder sollen, dann können Sie dies sehr einfach mit einem Konstruktor machen.

Schauen Sie sich die Definition der Klasse `Person` im Listing 3.8 noch einmal an. Diese Klasse besteht aus einer Eigenschaft `Name`. Wenn jetzt die Anforderung besteht, dass der Name bereits bei der Objekterzeugung mit einem Wert versehen wird, dann können Sie einen Konstruktor definieren, an den der Wert für den Namen übergeben wird.

```
Person(string personenName)
{
    Name = personenName
}
```

Die Instanziierung der Personenklasse könnte dann wie folgt aussehen:

```
Person p = new Person("Christian")
```

Hier wird der Wert »Christian« an den Konstruktor übergeben und damit die `Name`-Eigenschaft mit diesem Wert initialisiert.

Auf diese Art und Weise können Sie sehr einfach Ihr Objekt mit Werten vorinitialisieren.

Wenn Sie in Ihrer Klasse keinen Konstruktor selbst anlegen, wird vom Compiler standardmäßig ein Standardkonstruktor angelegt.

Mit C# 3.0 wurde eine neue zusätzliche Möglichkeit gegeben, Werte bequem zu initialisieren. Wenn Sie eine Klasse mit sehr vielen Eigenschaften haben und Sie wollen eine Möglichkeit haben, diese Werte in jeglicher Kombination vorzuinitialisieren, dann müssten Sie für jede Kombination einen entsprechenden Konstruktor definieren. Der Aufwand hierfür ist enorm. Deswegen bietet C# nun eine vereinfachte Objektinitialisierung.

Ohne entsprechende Konstruktoren anzulegen, können Sie die Initialwerte der Eigenschaften bei der Objekterzeugung in geschweiften Klammern angeben.

```
Person p = new Person() {Name = "Christian"}
```

Über diese Syntax können Sie kommagetrennt auch mehrere Eigenschaften in beliebiger Kombination initialisieren.

Methoden

Methoden stellen die Algorithmen innerhalb einer Klasse dar. Sie sind also die Berechnungsfunktionen einer Klasse, oder einfacher ausgedrückt: Mithilfe von Methoden ermöglichen Sie es Ihren Klassen, die Ausführung von Aktionen zu definieren.

Die Deklaration von Methoden ist also die Deklaration von Funktionen innerhalb einer Klasse.

Damit können Sie sich an den bei Funktionen bereits erläuterten Elementen einfach orientieren.

Felder und Eigenschaften

Felder einer Klasse sind die innerhalb dieser Klasse deklarierten Variablen. Über die Zugriffssteuerung, die Sie auch auf die Deklaration von Variablen innerhalb einer Klasse anwenden können (Sie erinnern sich: Tabelle 3.10), können Sie den Zugriff auf die Variablen bestimmen.

Mit Eigenschaften werden Informationen, die in Klassen verwendet werden sollen, bezeichnet. Um eine Eigenschaft zu erstellen, müssen Sie diese deklarieren. Sie müssen einen Namen für die Eigenschaft festlegen und einen Datentyp zuweisen, der von der Eigenschaft zurückgegeben werden soll.

```
[Bezeichner] Datentyp Eigenschaftsname
{  [get
   {
     Anweisungen
   }]

   [set
   {
     Anweisungen
   }]
}
```

Im Prinzip deklarieren Sie eine Property ähnlich wie eine Funktion. Sie können auch bei Propertys Parameterlisten mit übergeben, Sie können die Zugriffssteuerung festlegen, alles, wie Sie es von Funktionen bereits kennen.

Nun ist der Rahmen der Eigenschaft festgelegt. Hierfür gibt es zwei Anweisungsblöcke in der Eigenschaft. Innerhalb des einen Anweisungsblocks werden Eigenschaftswerte ermittelt:

```
get
{
  Anweisungen
}
```

Das Setzen der Eigenschaften wird durch den Abschnitt

```
set
{
  Anweisungen
}
```

vorgenommen. Das Ermitteln und das Setzen von Eigenschaften wird hierbei im Code konsequent aufgetrennt. Auf den `get`-Block wird zugegriffen, wenn Sie innerhalb eines Ausdrucks auf die Eigenschaft zugreifen. Der `set`-Block wird ausgeführt, wenn Sie der Eigenschaft einen Wert zuweisen wollen.

Falls Sie Eigenschaften so anlegen wollen, dass sie entweder nur gelesen oder nur geschrieben werden können, dann lassen Sie den entsprechenden `Set`- oder `Get`-Block einfach weg.

Standardmäßig können Sie allerdings sowohl das Lesen als auch das Schreiben von Werten durchführen.

In C# 3.0 ist auch eine kompakte Schreibweise für Propertys möglich, wie Sie es in Listing 3.8 sehen. Hier benötigen Sie nur noch folgende Syntax:

```
public string Name { get; set; }
```

NEU

Um das Anlegen einer Property möglichst komfortabel zu gestalten, müssen Sie im Codeeditor nur noch das Schlüsselwort `prop` angeben und danach zweimal die ↹-Taste drücken. Es wird Ihnen automatisch das gerade erwähnte kompakte Codegerüst angelegt.

Entwicklung von Klassen

Die Klasse, die wir in diesem Abschnitt deklarieren, hat eine Methode und einige Eigenschaften, um Ihnen die Entwicklung einer Klasse noch einmal konkret näher zu bringen.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    int Erg;
    LottoZahl Zahl = new LottoZahl();
    SuperZahl Zahl2 = new SuperZahl();
    Spiel77 Zahl3 = new Spiel77();
    Erg = Zahl.ZufallsInt();
    Response.Write("Aufruf der Klasse LottoZahl<br>");
    Response.Write("Erg hat den Wert: ");
    Response.Write(Erg.ToString());
    Erg = Zahl2.ZufallsInt();
    Response.Write("<br>Aufruf der Klasse SuperZahl <br>");
    Response.Write("Erg hat den Wert: ");
    Response.Write(Erg.ToString());
    Response.Write(" <br> ");
    Erg = Zahl2.ZufallsInt(0, 999999);
    Response.Write("Aufruf der Klasse SuperZahl mit BasisMethode");
    Response.Write("<br>Erg hat den Wert: ");
    Response.Write(Erg.ToString());
    Erg = Zahl3.ZufallsWert;
    Response.Write("<br>Aufruf der Klasse Spiel77 <br>");
    Response.Write("Erg hat den Wert: ");
    Response.Write(Erg.ToString());
  }

  public class ZufallsZahl
  {
    public int ZufallsInt(int Unten, int Oben)
    {
      Random r = new Random();
      return r.Next(Unten, Oben);
    }
  }

  public class LottoZahl : ZufallsZahl
  {
    public int ZufallsInt()
    {
      Random r = new Random();
      return r.Next(1, 49);
    }
  }

  public class SuperZahl : ZufallsZahl
  {
    public int ZufallsInt()
    {
```

```
      Random r = new Random();
      return  r.Next(1, 9);
    }
  }

  public class Spiel77 : ZufallsZahl
  {
    public int ZufallsWert
    {
      get { return ZufallsInt(0, 9999999); }
    }
  }
</script>
```

Listing 3.16: Eine Klasse mit einer Eigenschaft wird deklariert (Class.aspx).

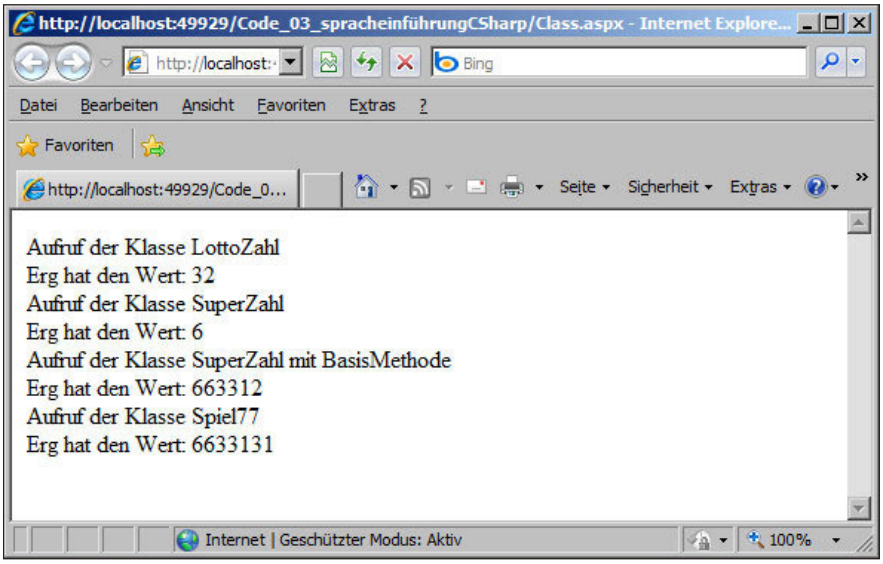


Abbildung 3.21: Ergebnis der Lottozahlen

In diesem kleinen Programm ist eine Basisklasse deklariert worden, die eine Methode enthält. Weiterhin haben wir drei Klassen definiert, die aus dieser Basisklasse entstanden sind.

Wir haben in einer Klasse die Basismethode durch Überladung verändert. Wir haben sie in der zweiten Klasse durch Überladen um eine gleichartige Methode mit weniger Parametern ergänzt. In der dritten Klasse haben wir nur eine Eigenschaft hinzugefügt, die ein ganz spezieller Methodenaufruf der Basisklasse ist.

Die Funktionsweise Überladung ist durch den doppelten Aufruf der Methode einmal in der schreibgeschützten Eigenschaft und zum anderen direkt im erzeugten Objekt demonstriert worden.

3.4.3 Zusammenstellung von Bibliotheken, Einbindung von Namespaces und externen Objekten

Sie haben in C# die Möglichkeit, Ihre Objekte und Klassenbibliotheken in einzelne Dateien abzuspeichern und auf diese Art für die verschiedenen Projekte nur einzelne Elemente zu verwenden.

Bei der Neuerstellung eines Codeblocks werden automatisch bereits erste Standardelemente für ein Programm vorgegeben.

In einem anderen Programmblock können Sie dann diese Bibliotheken importieren und damit die dort deklarierten Elemente zur Verfügung stellen.

Den Import einer Bibliothek führen Sie über die Verwendung der Compiler-Direktiven durch. Neben der direkten Verwendung im Kommandozeilenmodus können Sie diese Direktiven auch direkt im Visual Web Developer angeben. Auf die Verwendung des Kommandozeilenmodus gehen wir in einem späteren Abschnitt noch einmal gesondert ein.

Eine Bibliothek können Sie außerdem unter dem Menüpunkt WEBSITE und dem Unterpunkt VERWEIS HINZUFÜGEN ergänzen. Sie erhalten ein Untermenü, in dem unter anderem alle .NET-Komponenten aufgelistet sind. Das nachfolgende Bild zeigt hiervon einen kleinen Ausschnitt.

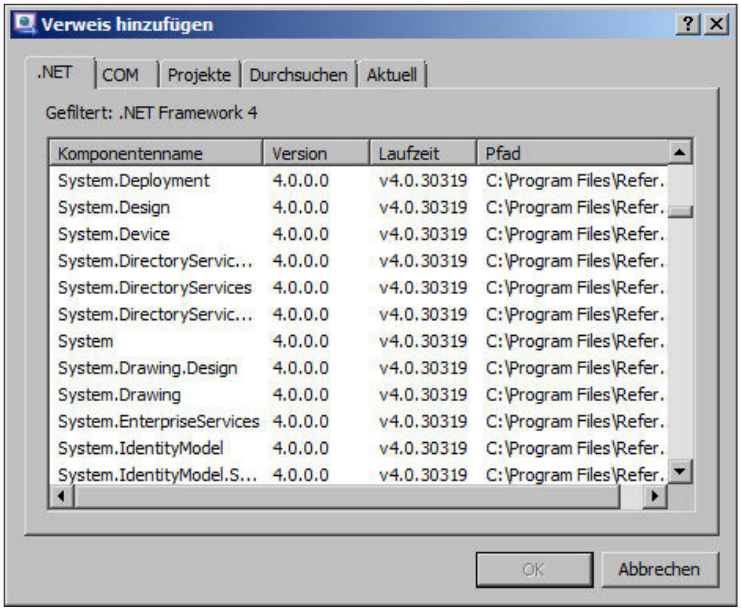


Abbildung 3.22: Ein kleiner Ausschnitt von Verweisen

Jetzt geht es noch darum, wie Sie Klassen und deren Funktionalitäten von zusätzlich referenzierten Bibliotheken nutzen können.

Im Quelltext importieren Sie dann den Namespace mit dem Schlüsselwort `using`.

```
using System.Numerics;
```

Wenn Sie einen Namespace importiert haben, brauchen Sie den Namespace nicht mehr zur Verwendung der Klasse explizit voranzustellen.

Diese Zeile importiert also genau diesen beschriebenen Namespace zur Verwendung im Quelltext. Danach können Sie im Quelltext die darin deklarierten Klassen mit allen Methoden und Eigenschaften verwenden.

3.5 Basisfunktionen des .NET Frameworks

Es gibt eine ganze Reihe von Elementen, die als Teil des .NET Frameworks bereitgestellt werden. Diesen Elementen, die in Basisklassen implementiert sind, wollen wir uns nun eingehender widmen.

Die Funktionalitäten, auf die wir nachfolgend eingehen, sind Teil der Klassenbibliothek des .NET Frameworks. Wir haben einige der Module, wo es geeignet erschien, inhaltlich in einem Abschnitt zusammengefasst.

3.5.1 Standardfunktionen und Methoden zur Stringmanipulation

Das Anzeigen, Umwandeln, Ergänzen und Manipulieren von Zeichenketten stellt einen Bereich der Programmierung dar, den Sie bei Ihren Programmen sicher häufig brauchen werden.

Als Erstes möchten wir kurz die Möglichkeiten der Stringmanipulation erläutern. Wir fassen die vorgestellten Eigenschaften und Methoden abschließend jeweils in einer kleinen Tabelle noch einmal stichpunktartig zusammen. Die folgende Zeichenkette dient als Basis für die jeweiligen Manipulationsziele (der String wird mit je drei Leerzeichen begonnen und abgeschlossen).

```
string Bstring = "   abc def GHI jkl   ";
```

Wenn Sie bestimmte Abschnitte aus einem String herausschneiden wollen, so können Sie die Methode `String.Remove()` verwenden. Als Funktionsergebnis wird jeweils der entsprechend manipulierte String zurückgeliefert. Ein Parameter ist die Anzahl der betroffenen Zeichen. Alternativ geben Sie vor diesem Parameter einen zweiten Parameter an, der das Startzeichen, ab dem die Manipulation beginnen soll, kennzeichnet.

Beispiele:

» `Erg= Bstring.Remove(0, 8);`
liefert als Ergebnis: `"ef GHI jkl   "`

» `Erg= Bstring.Remove(8);`
liefert als Ergebnis: `"   abc d"`

» `Erg= Bstring.Remove(5, 8);`
liefert als Ergebnis: `"   abI jkl   "`

Auch das Beschneiden von Strings um Leerzeichen am Anfang und am Ende der Zeichenkette ist sehr hilfreich und wird durch die Methoden `String.Trim()` (schneidet an beiden Enden der Zeichenkette), `String.TrimEnd()` (schneidet am rechten Ende der Zeichenkette) und `String.TrimStart()` (schneidet am linken Ende der Zeichenkette) unterstützt.

Beispiele:

» `Erg = Bstring.TrimStart();`
liefert als Ergebnis: "`abc def GHI jkl  `"

» `Erg = Bstring.TrimEnd();`
liefert: "`   abc def GHI jkl`"

» `Erg = Bstring.Trim();`
liefert als Ergebnis: "`abc def GHI jkl`"

Wenn Sie Informationen über eine Zeichenkette erhalten wollen, so können Sie beispielsweise mit der Eigenschaft `String.Length` die Länge einer Zeichenkette ermitteln. Die Frage, ob in einer Zeichenkette ein bestimmter String vorkommt, können Sie mit der Methode `String.Contains()` klären (hierbei geben Sie als Parameter den zu suchenden Substring an). Wenn Sie die Positionen der von Ihnen gesuchten Zeichenkette ermitteln wollen, können Sie dies mit der Methode `String.IndexOf()` und, falls Sie die Position rückwärts suchen wollen, mit `String.LastIndexOf()` ermitteln. Hierbei gibt es verschiedene überladene Methoden, mittels derer Sie den Suchbereich (Startposition der Suche, Anzahl der zu durchsuchenden Zeichen im String) festlegen können. Wollen Sie eine Zeichenkette in einem String ersetzen, so steht Ihnen `String.Replace()` zur Verfügung.

Beispiele:

» `int Erg = Bstring.Length`
liefert als Ergebnis: `21`

» `int Erg = Bstring.IndexOf(" ")`
liefert als Ergebnis: `1`

» `int Erg = Bstring.LastIndexOf("")`
liefert als Ergebnis: `21`

» `string Erg = Bstring.Replace ("c def GHI ", " ")`
liefert als Ergebnis den String: "`   ab jkl   `"

Abschließend wollen wir auch noch auf die Möglichkeiten von Umwandlungen in Groß- oder Kleinbuchstaben in einem String oder der Aufspaltung von einem String in mehrere Zeichenketten (bzw. des Zusammenfügens mehrerer Zeichenketten in einen String) mit oder ohne trennendes Kennzeichen eingehen.

Die Buchstaben in einer Zeichenkette lassen sich einfach über `String.ToLower()` in Kleinbuchstaben und `String.ToUpper()` in Großbuchstaben umwandeln. Sonderzeichen und Ziffern sind hiervon übrigens nicht berührt. Sehr umfassende Möglichkeiten zur Stringformatierung stehen Ihnen mit der Methode `String.Format()` zur Verfügung.

Das Auftrennen und das Zusammenfügen von Zeichenketten erfolgen über die Methoden `String.Split()` und `String.Join()`.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    string Arg1 = "   abc def GHI jkl   ";
    string Arg3 = Arg1.Trim();
    int Arg4 =0;
    Response.Write("Einige Stringmanipulationen<br>");
    Response.Write("Trim: ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg3 = Arg1.Remove(0, 10);
    Response.Write("<br>Remove(0, 10): ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg3 = Arg1.Remove(10);
    Response.Write("<br>Remove(10): ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg3 = Arg1.Remove(5, 5);
    Response.Write("<br>Remove(5,5): ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg3 = Arg1.ToLower();
    Response.Write("<br>ToLower: ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg3 = Arg1.ToUpper();
    Response.Write("<br>ToUpper: ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg3);
    Arg4 = Arg1.Length;
    Response.Write("<br>Length: ");
    Response.Write(Arg1);
    Response.Write("<br>");
    Response.Write(Arg4);
    Response.Write("<br>");
  }
</script>
```

Listing 3.17: Strings manipulieren (StringManipulation.aspx)

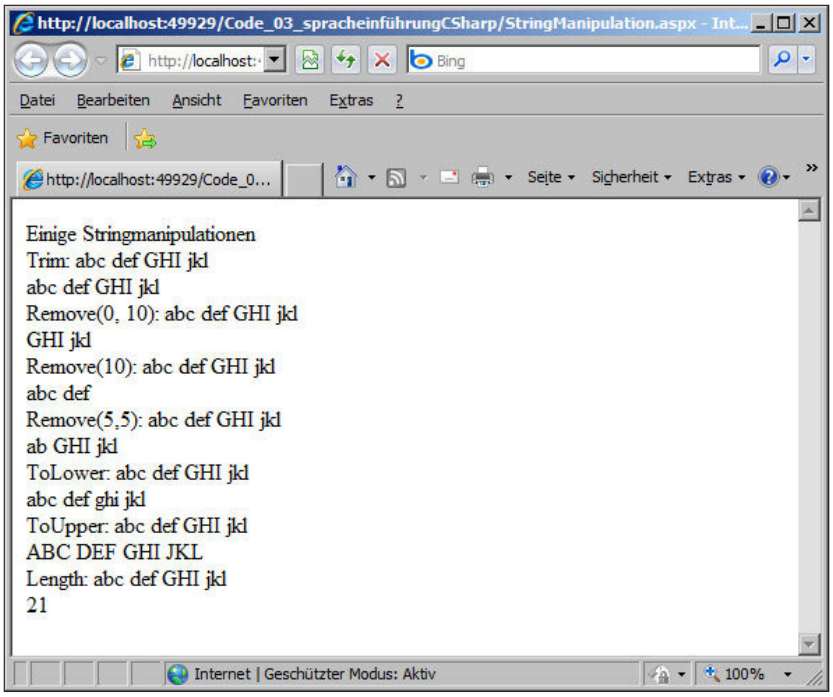


Abbildung 3.23: Strings manipulieren

In anderen Szenarien haben Sie die Möglichkeit, durch die Verwendung zusätzlicher Parameter diese Methoden weitaus umfangreicher nutzen zu können.

3.5.2 Andere nützliche Methoden und Funktionen

Es existieren eine ganze Reihe weiterer nützlicher Methoden innerhalb des .NET Frameworks.

Generierung von Zufallszahlen

Als sprachunabhängiges Element der Zufallszahlengenerierung ist die Verwendung der `Random`-Klasse vorgesehen. Wir haben diese bereits in den Beispielen für die `if ... else`-Kontrollstruktur und die `while`-Schleife verwendet.

Nachfolgend noch ein paar kurze weitere Erläuterungen zu den als Teil der `Random`-Klasse zur Verfügung stehenden Methoden:

Methode	Beschreibung
`Random.Next()`	Es wird eine ganzzahlige positive Zufallszahl zurückgeliefert. Der mögliche Wertebereich ist der Wertebereich eines Int32-Werts. Sie können einen oder zwei Parameter angeben, ein einzelner Parameter definiert eine obere Grenze und zwei Parameter eine untere und eine obere Grenze für den zulässigen Wertebereich. `Random.Next(4,8)` würde also Zufallszahlen zwischen 4 und 8 zurückliefern.

Methode	Beschreibung
`Random.NextBytes()`	Mit dieser Methode lassen sich alle Elemente eines Byte-Arrays mit verschiedenen Zufallszahlen belegen.
`Random.NextDouble()`	Mit dieser Methode wird eine Zufallszahl erzeugt, die zwischen 0 und 1 liegt und den Datentyp `Double` hat.

Tabelle 3.13: Methoden zur Erzeugung von Zufallszahlen

Für Beispiele verweisen wir auf die bereits oben erwähnten Programmbeispiele.

Mathematische Methoden

Mathematische Funktionen werden durch Methoden aus der statischen Klasse `Math` bereitgestellt.

Nachfolgend geben wir Ihnen einen kurzen Überblick über diese Klasse.

Als grobe Einteilung lassen sich die mathematischen Methoden von den Funktionalitäten her in trigonometrische Methoden und andere (nicht trigonometrische) Methoden einteilen.

Trigonometrische Methoden und allgemeine mathematische Funktionen

Für Operationen der Trigonometrie stellt C# alle klassischen Funktionen in Form von Methoden bereit. Die Berechnung von Sinus, Cosinus, Tangens und Arcustanges ist mit diesen Methoden kein Problem.

Mit der Methode `Math.Exp()` lässt sich die Exponentialfunktion abbilden, `Math.Log()` ermittelt den natürlichen Logarithmus, `Math.Log10()` den Logarithmus auf der Basis 10 und `Math.Sqrt()` beispielsweise die Quadratwurzel.

Die Funktionalitäten und die Verwendung der Trigonometrie-Methoden und der Methoden für Exponentialfunktion und die anderen allgemeinen Funktionen entnehmen Sie dem nachfolgenden Beispielprogramm.

CODE

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    double Arg =1;
    double Erg;
    float Erg2;
    double pi = 3.14159263;
    Response.Write("Einige trigonometrische ");
    Response.Write("Funktionen: <br> ");

    Erg = Math.Sin(Arg);
    Response.Write("Sinus von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");
```

CODE

```
    Erg = Math.Cos(Arg);
    Response.Write("Cosinus von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");

    Erg = Math.Tan(Arg);
    Response.Write("Tangens von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");

    Erg = Math.Atan(Arg);
    Response.Write("Arcustangens von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");

    Erg = Math.Exp(Arg);
    Response.Write("Exponentialfunktion von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");

    Arg = 2;
    Erg = Math.Sqrt(Arg);
    Erg2 = (float)Math.Sqrt(Arg);
    Response.Write("Quadratwurzel von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write(" (Double);");
    Response.Write(Erg2.ToString());
    Response.Write(" (Float)<br>");
  }
</script>
```

Listing 3.18: Trigonometrische und mathematische Funktionen (*Trigon.aspx*)

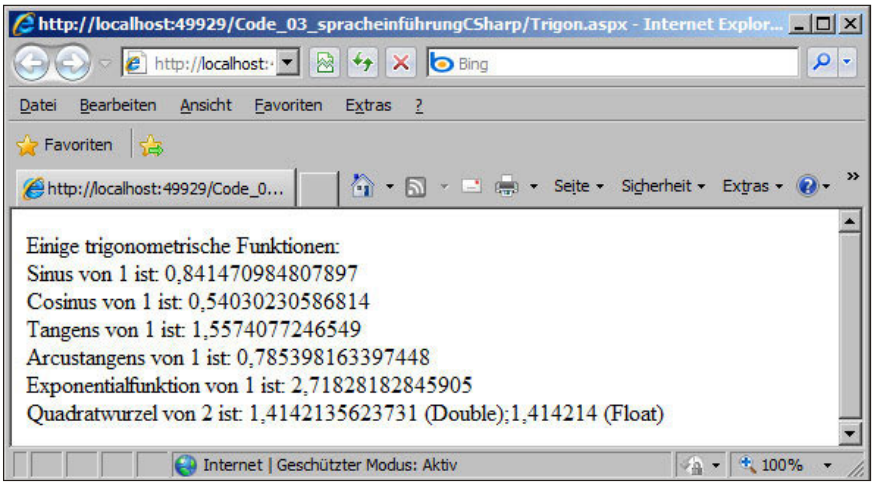


Abbildung 3.24: Trigonometrische und mathematische Funktionen

Anpassungen von Zahlen

Die Methoden `Math.Abs()`, `Math.Sign()` und `Math.Round()` lassen sich am besten damit beschreiben, dass mit ihnen Anpassungen an Zahlen vorgenommen werden können.

Die erste Methode `Math.Abs()` ermittelt den Absolutwert einer Zahl – das bedeutet, negative Zahlen werden eliminiert. `Math.Sign()` ermittelt das Vorzeichen einer Zahl. Die dritte Methode `Math.Round()` führt eine Rundung einer reellen Zahl durch. Auch wenn der zurückgegebene Datentyp derselbe ist wie der der ursprünglichen Zahl, werden doch die Nachkommastellen abgeschnitten.

Auch hier werden die Funktionalitäten dieser drei Methoden an einem Beispiel kurz demonstriert.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    double Arg = -1.2345;
    double Arg2 = 1.2345;
    double Erg;
    Response.Write("Einige mathematische Funktionen: <br>");
    Erg = Math.Abs(Arg);
    Response.Write("Abs von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");
    Erg = Math.Sign(Arg);
    Response.Write("Signum von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");
```

```
    Erg = Math.Sign(Arg2);
    Response.Write("Signum von ");
    Response.Write(Arg2.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");
    Erg = Math.Round(Arg);
    Response.Write("Round von ");
    Response.Write(Arg.ToString());
    Response.Write(" ist: ");
    Response.Write(Erg.ToString());
    Response.Write("<br>");
  }
</script>
```

Listing 3.19: Mathematische Methoden (Math.aspx)

Hier die Ausgabe im Browser:

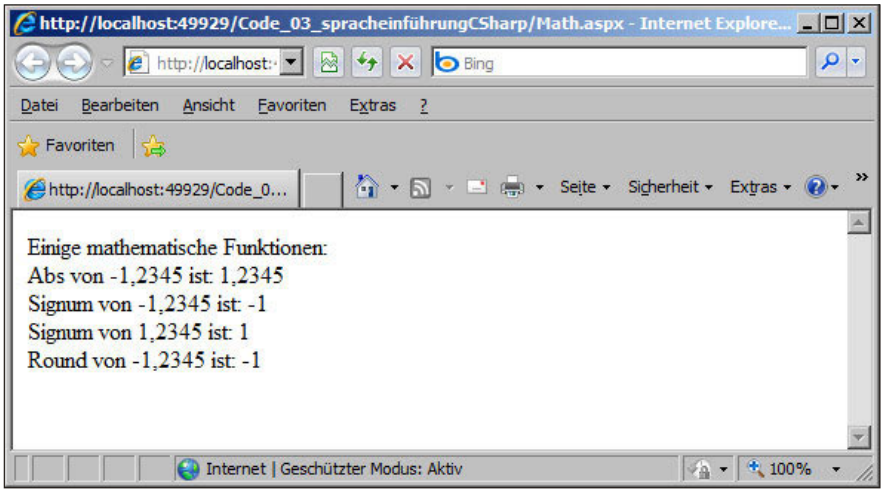


Abbildung 3.25: Mathematische Methoden

Mit der aktuellen Framework-Version 4.0 gibt es jetzt auch eine Unterstützung von komplexen Zahlen. Genauso wie der neue Datentyp `BigInteger` befindet sich die Klasse `Complex` in der Bibliothek `System.Numerics`.

Um eine komplexe Zahl zu definieren und zu instanzieren, gibt es eine eigene Konstruktorüberladung, bei der sowohl der reelle als auch der imaginäre Teil der Zahl angegeben wird.

```
Complex c = new Complex(5,2);
```

3.6 Kompilierung von Programmen

Für die Übersetzung der von Ihnen erstellten Programme haben Sie zwei Möglichkeiten. Die eine haben wir bisher immer verwendet: die Verwendung des in die Entwicklungsumgebung integrierten Compilers.

In diesem Abschnitt wollen wir Ihnen die zweite Variante für die Kompilierung von C#-Programmen vorstellen: die Verwendung des ebenfalls zur Verfügung stehenden Befehlszeilen-Compilers und die damit verbundenen Möglichkeiten.

3.6.1 Aufruf des Befehlszeilen-Compilers

Sie haben grundsätzlich die Möglichkeit, den Befehlszeilen-Compiler aufzurufen. Zum einen steht Ihnen der Befehlszeilen-Compiler innerhalb der Visual Studio 2010-Umgebung (nicht bei der Expressversion) zur Verfügung, zum anderen können Sie ihn auch direkt über die Windows-Eingabeaufforderung aufrufen.

Wir konzentrieren uns hier auf den Aufruf aus der Windows-Eingabeaufforderung, weil in der Visual Studio Express Edition, die hier ja als Arbeitsgrundlage dient, kein Kommandozeilen-Compiler direkt enthalten ist.

Sie öffnen die Eingabeaufforderung, indem Sie im Menü START unter AUSFÜHREN den Befehl CMD eingeben.

INFO

Beachten Sie, dass Sie die Path-Umgebung Ihres Computers so anpassen müssen, dass Sie den Ort des Programms *csc.exe* bekannt machen. Ansonsten müssen Sie immer den vollständigen Pfad angeben. Wenn Sie Visual Studio verwenden, können Sie auch über die dort bereitgestellte Kommandozeile gehen.

Es öffnet sich die Eingabeaufforderung. Hier können Sie nun mit dem Befehl *csc.exe* den C#-Compiler aufrufen.

Dabei haben Sie die Möglichkeit, eine ganze Reihe von Kommandozeilenparametern mitzugeben. Einige der einzelnen Parameter und ihre Bedeutung können Sie der nachfolgenden Tabelle entnehmen.

Parameter	Bedeutung
`/?` `/help`	Zeigt die Compiler-Optionen an. Es wird dabei keine Kompilierung durchgeführt.

`/debug+`
`/debug-`
`/debug:full`
`/debug:pdbonly`
	Sie können die Ausgabe von Debug-Informationen einstellen und abstellen. Außerdem können Sie die Ausgabe auf eine vollständige Ausgabe dieser Informationen einstellen oder auf die pdb-Informationen einschränken.

`/define:`*`Konstante=Wert`*	Der globale Wert für die Konstanten für eine bedingte Kompilierung wird festgesetzt.

Parameter	Bedeutung

`/doc+`
`/doc-`
`/doc:`*`filename`*
	Erstellen einer XML-Dokumentationsdatei für das zu übersetzende File. Alle Kommentare des Files werden in diese Datei übernommen.

`/out:`*`filename`*	Angabe des Ausgabedateinamens
`/target:[exe|winexe|library|module]`	Ausgabetyp der Assembly

Tabelle 3.14: Einige wichtige Optionen des Kommandozeilen-Compilers

3.7 Fehler- und Ausnahmebehandlung in C#

Fehler und Ausnahmen treten in den unterschiedlichsten Varianten auf, wenn Sie ein Programm schreiben.

Die einfachste und am schnellsten zu eliminierende Art von Fehlern sind Syntaxfehler. Sie werden bereits bei der Codeerfassung durch den Editor erkannt und abgefangen, spätestens bei der Aufbereitung und Übersetzung des Programms werden sie endgültig aufgespürt.

Als zweite Art von Fehlern gibt es logische Fehler, in denen Bearbeitungsschritte nicht in einer sinnvollen Reihenfolge durchgeführt werden oder anderes schwierig nachvollziehbares Verhalten erzeugt wird. Diese Fehler sind durch Unterstützung von Werkzeugen schwierig aufzuspüren. Sie lassen sich eigentlich nur durch eine detaillierte Planung der Programmstruktur und späteres ausführliches Testen erkennen und beseitigen.

Als dritte Art von Fehlern gibt es die sogenannten Laufzeitfehler (oder Ausnahmen, die während der Programmabarbeitung auftreten). Für diese Fehlerklasse bietet C# Unterstützung an, die wir uns nachfolgend näher ansehen wollen.

Wir haben bereits bei der Typkonvertierung die Ausnahmebehandlung in C# kurz gestreift, indem wir durch einen strukturierten Fehlerbehandlungsblock das Ergebnis eines Fehlers abgefangen und so einen Programmabbruch vermieden haben.

Fehler und Ausnahmen besitzen dieselbe Ursache, so dass wir nachfolgend die beiden Begriffe als Synonyme verwenden werden.

3.7.1 Strukturierte Fehlerbehandlung

In der strukturierten Ausnahmebehandlung gehen Sie auf den Umstand ein, dass in bestimmten Abschnitten immer unvorhergesehene Dinge passieren können. Ein Beispiel hierfür sind Erfassungen durch einen Benutzer oder der Umgang mit Datenströmen oder der Umgang mit dem Herstellen und Halten einer Verbindung zu einem anderen beteiligten System.

Sie können darauf hoffen, dass bestimmte, nicht geplante Programmzustände nicht auftreten. Stattdessen können Sie aber auch gezielt Programmabschnitte bauen, die dann abgearbeitet werden, wenn ein solcher unvorhergesehener Zustand auftritt. Durch das Erzwingen einer er-

neuten Eingabe von Daten oder der weiteren Abarbeitung eines Programmblocks mit festen Parametern können Sie solche undefinierten Programmzustände umschiffen und vermeiden.

In C# können Sie Codeblöcke mit diesen potenziellen Fehlerquellen durch die `try ... catch`-Kontrollstruktur abfangen und auf einfache Weise handhaben.

Sehen wir uns diese Kontrollstruktur nun einmal genauer an. Sie kennen die Syntax ja bereits aus dem Abschnitt über Kontrollstrukturen.

```
try
{
  Anweisungen
}
catch [Filter]
{
  Anweisungen
}
finally
{
  Anweisungen
}
```

Die Anweisungen, die eventuell eine Ausnahmebehandlung erforderlich machen könnten, schließen sich an das Schlüsselwort `try` an. Falls eine solche Ausnahme auftritt, wird der Abschnitt, der nach dem Schlüsselwort `catch` folgt, ausgeführt. Sie können beliebig viele unterschiedliche `catch`-Blöcke hintereinander setzen. Falls eine Ausnahme auftritt, wird jeder Filter jedes einzelnen `catch`-Blocks geprüft, und falls die im Filter beschriebene Ausnahme auftritt, wird der `catch`-Block ausgeführt. Daraus ergibt sich, dass Sie sinnvollerweise die `catch`-Blöcke in der Art aufbauen sollten, dass die Ausnahmen von Block zu Block allgemeiner gehalten werden.

Optional können Sie noch einen Block anschließen, der durch das Schlüsselwort `finally` abgegrenzt wird. Der daran anschließende Programmblock wird in jedem Fall ausgeführt, egal ob eine Ausnahme aufgetreten ist oder nicht.

ACHTUNG

Sie haben die Möglichkeit, aus einem `catch`-Block wieder in den `try`-Block zurückzuspringen, so dass Sie Programmcode aus diesem Block einfach wiederholen können. Es ist nicht gestattet, aus einem `catch`-Block in einen nachfolgenden neuen `try`-Block zu springen oder von einem `catch`-Block in einen anderen `catch`-Block.

Wir haben die strukturierte Fehlerbehandlung bereits in vorherigen Beispielen eingesetzt.

3.7.2 Die Exception-Klasse des .NET Frameworks

Die `Exception`-Klasse, die wir bereits im vorherigen Abschnitt angesprochen haben, stellt die Basis für die Ausnahmebehandlung mithilfe des .NET Frameworks dar. Daher wollen wir Ihnen in diesem Abschnitt die wesentlichen Eigenschaften und Methoden vorstellen.

Die Eigenschaften der Klasse enthalten hauptsächlich Metainformationen über den aufgetretenen Fehler, der eine genauere Analyse und basierend darauf zielorientierte Lösungsansätze ermöglicht.

In der nachfolgenden Tabelle sind die wichtigsten Eigenschaften aufgelistet und beschrieben.

Eigenschaft	Bedeutung
`Exception.Data`	Mittels Schlüssel-Werte-Paaren werden zusätzliche Informationen über den Fehler zur Auswertung bereitgestellt.
`Exception.HelpLink`	Mittels dieser Eigenschaft ist eine Verknüpfung zu einer Hilfedatei definiert.
`Exception.InnerException`	Der Rückgabewert ist die `Exception`-Instanz, welche die aktuelle Ausnahme ausgelöst hat, falls jemand die ursprüngliche `Exception` weitergeworfen hat. Die `InnerException`-Eigenschaft kann auch `null` sein. Mit dieser Eigenschaft erhalten Sie eine Möglichkeit, den Verlauf von Ausnahmen, der zur aktuellen Ausnahme geführt hat, herzuleiten und zu analysieren.
`Exception.Message`	Diese Eigenschaft gibt eine Textbeschreibung der Ausnahme aus.
`Exception.StackTrace`	Es wird der Stack ausgegeben, auf dem alle Methoden, die gerade ausgeführt werden, protokolliert sind. Damit ist die Identifizierung der Codezeile, die zur Ausnahme geführt hat, meistens leicht möglich.
`Exception.Source`	Der Name der Anwendung oder des Objekts, das den Fehler ausgelöst hat, wird zurückgegeben.
`Exception.TargetSite`	Die Methode, welche die aktuelle Ausnahme ausgelöst hat, wird ermittelt und abgerufen.

Tabelle 3.15: Eigenschaften der Exception-Klasse

3.7.3 Die Erzeugung von Ausnahmen

Nachdem wir uns mit dem Abfangen von Ausnahmen beschäftigt haben, werfen wir noch einen kurzen Blick darauf, wie eine Ausnahme erzeugt werden kann.

Neben den verschiedenen, bereits vorgegebenen möglichen, zur Laufzeit eines Programms auftretenden Fehlern und Ausnahmen haben Sie die Möglichkeit, in C# auch selbstständig eine solche Ausnahme zu erzeugen. C# stellt Ihnen hierfür das Schlüsselwort `throw` bereit.

Sie erzeugen einfach eine Ausnahme innerhalb Ihres Programms mittels der Zeile:

```
throw Ausdruck;
```

Der Ausdruck muss hierbei von `System.Exception` abgeleitet sein.

3.8 C# 4.0-Neuerungen

Mit der Sprachversion 4.0 wurden einige kleine Spracherweiterungen eingeführt, von denen wir in diesem Abschnitt die zwei wichtigsten kurz vorstellen wollen.

3.8.1 Optionale und benannte Parameter

In Visual Basic gibt es ja schon lange optionale und benannte Parameter. In C# war man lange der Meinung, dass durch die Überladung von Methoden auf diese Sprachkonstrukte verzichtet werden kann. Doch vor allem im Bereich der COM-Interoperabilität (zum Beispiel Office Automation) stellte sich heraus, dass C#-Code an dieser Stelle im Vergleich zu Visual Basic wesentlich aufwändiger zu entwickeln war. Deswegen wurden jetzt auch in C# benannte und optionale Parameter eingeführt.

Stellen Sie sich vor, Sie haben folgende Methodendefinition:

```
void SaveDocument(string fileName, bool overwrite, bool readOnly, string password) {}
```

Sie hätten nun bei einem Methodenaufruf jeden einzelnen Parameter angeben müssen oder dementsprechende Überladungsmethoden schreiben müssen (bei Word besitzt die `Save()`-Methode eines Dokuments ca. 20 Parameter).

Da Sie den Dateinamen wohl zwingend brauchen, könnten Sie die anderen drei Parameter als optionale Parameter definieren. Das machen Sie, indem Sie die anderen Parameter einfach mit einem Standardwert definieren.

```
void SaveDocument(string fileName, bool overwrite=true, bool readOnly=false, string pass-
word="") {}
```

Dabei müssen nicht optionale Parameter vor den optionalen Parametern definiert sein und natürlich auch weiterhin übergeben werden.

Die Methode kann jetzt wie folgt aufgerufen werden:

```
SaveDocument("C:\\temp\\test.txt");
```

Auf die Angabe der optionalen Parameter kann verzichtet werden.

Wollen Sie genau einen optionalen Parameter angeben, so müssen Sie alle vor diesem definierten optionalen Parameter angeben, es sei denn, Sie verwenden das Konstrukt der benannten Parameter. Dabei geben Sie vor dem Übergabewert den Namen des entsprechenden Parameters an, wie der folgende Codeausschnitt illustriert:

```
SaveDocument("C:\\temp\\test.txt", password:"test");
```

Dabei ist jetzt auch die Reihenfolge der Parameter nicht mehr entscheidend. Wenn Sie Parameter benennen, können Sie diese in jeglicher beliebigen Reihenfolge übergeben.

Es wäre also auch folgende Syntax zulässig, bei der der nicht optionale Parameter am Ende der Reihenfolge steht.

```
SaveDocument(password:"test", fileName:"C:\\temp\\test.txt");
```

3.8.2 Dynamische Spracherweiterungen

In letzter Zeit gab es immer mehr Sprachen, wie zum Beispiel JavaScript, IronRuby oder IronPython, auf dem Markt, die mit dynamischer Programmierung sehr viel Erfolg hatten. Die Möglichkeiten, die dynamische Sprachen besitzen, wurden jetzt auch in C# 4.0 eingeführt.

Bevor wir uns jedoch mit der Implementierung in C# befassen, sollten wir uns kurz Gedanken machen, wo die Vorteile von dynamischen Sprachen liegen.

Dynamische Sprachen können mit zur Kompilierzeit unbekannten Datentypen wesentlich intuitiver und leichter umgehen als dies mit statischen Sprachen möglich wäre.

Im .NET Framework gibt es mit Reflections auch eine Zauberbox, die mit unbekannten Typen zur Kompilierzeit gut umgehen kann, doch ob der resultierende Code immer intuitiv und gut les- und wartbar ist, mögen wir mal in den Raum stellen.

Sie werden erkennen, dass Sie mit den dynamischen Möglichkeiten auf viele Typcasts verzichten können, was wie bei den eben beschriebenen optionalen Parametern gerade bei der Office-Automation ein wahrer Segen ist.

Es sollte jedoch nicht verschwiegen werden, dass diese Dynamik auch ihren Preis hat. Bei statischen Sprachen muss der verwendete Datentyp bereits zur Kompilierzeit bekannt sein und somit können Tippfehler oder sonstige syntaktische Fehler bereits zur Kompilierzeit erkannt werden. Die Typenüberprüfung bei dynamischen Sprachen erfolgt erst zur Laufzeit, was natürlich in Bezug auf Fehleranfälligkeit ein Nachteil ist. Somit kann ein statischer Compiler natürlich auch Optimierungen am Code ausführen, was in der Regel zu einer bei weitem besseren Laufzeit gegenüber dynamischen Sprachen führt.

C# 4.0 führt dazu einen neuen Datentyp `dynamic` ein, mit dem die dynamischen Möglichkeiten auch in C# Einzug halten.

```
dynamic test = "Dynamisch";
```

Bitte beginnen Sie jetzt nicht, `dynamic` mit `var` zu vergleichen. Es handelt sich hierbei wirklich um zwei verschiedene Paar Stiefel. Die implizite Typisierung, die mit dem Schlüsselwort `var` einhergeht, stellt den entsprechenden Typ bereits zur Kompilierzeit fest und der Compiler erzeugt den fehlenden Code. Die Vorteile sind IntelliSense, ein besseres Laufzeitverhalten und Typchecks bereits zur Kompilierungszeit. Dynamische Sprachen werten dagegen den Ausdruck `dynamic` erst zur Laufzeit aus.

Schauen wir uns einfach ein Beispiel an, um Reflections und `dynamic` miteinander zu vergleichen. Dabei erzeugen wir einmal mittels Reflections und einmal mit dem dynamischen Sprachkonstrukt ein Objekt vom Typ `Random` und rufen dann die Methode `Next()` dieses Objekts auf.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
  {
    //Reflections
    object zufallsGenerator =
           Activator.CreateInstance(Type.GetType("System.Random"));
```

```
    Type objType = zufallsGenerator.GetType();
    var method = objType.GetMethod("Next",System.Type.EmptyTypes);
    Response.Write(method.Invoke(zufallsGenerator, null).ToString() +
             "<br>");
    //Dynamic
    dynamic dynamicGenerator =
            Activator.CreateInstance(Type.GetType("System.Random"));
    Response.Write(dynamicGenerator.Next().ToString());
  }
</script>
```

Listing 3.20: Beispiel für dynamische Programmierung im Vergleich zu Reflections (DynamicVsReflections.aspx)

Wir denken, der Unterschied in der Lesbarkeit ist offensichtlich.

Die Verwendung dieses dynamischen Sprachkonstrukts macht in den folgenden Szenarien Sinn:

» Beim Arbeiten mit COM-Objekten, um viele Typumwandlungen zu vermeiden und somit lesbareren Code zu schreiben

» Bei der Interaktion mit dynamischen Sprachen

» Bei der Arbeit mit Objekten, bei denen sich die Strukturen sehr häufig ändern können, wie zum Beispiel XML-Dokumente

3.9 Support von C# in Visual Web Developer

Kommen wir nun zurück zum Visual Web Developer und den Unterstützungsmöglichkeiten, die Ihnen mit dieser Umgebung bei der Erstellung von C#-Code zur Verfügung gestellt werden.

HINWEIS

Die in diesem Abschnitt beschriebenen Dinge können sich je nach Installation und Konfiguration der Einstellungen des Visual Web Developers auf Ihrem System anders verhalten, als es hier beschrieben ist. Sie sollten durch die hier beschriebenen Punkte dennoch einen Eindruck und Überblick über die Features bekommen können.

Die beschriebenen Punkte basieren auf dem Visual Web Developer direkt nach einer Standardinstallation.

Sie haben vermutlich bereits beim Schreiben Ihrer Programme bemerkt, dass Ihnen Features zur Verfügung stehen, die Informationen über die Verwendung von Befehlen, Datentypen und auch Elemente des .NET Frameworks liefern. Auf diese Hilfsmittel wollen wir nachfolgend noch einmal eingehen.

3.9.1 Ein erster Eindruck von den Möglichkeiten

Um an einigen Beispielen zu zeigen, welche Hilfsmittel Ihnen für die Programmierung bereitgestellt werden, geben Sie einmal das nachfolgende (nicht ausführbare) Programm ein.

```
<%@ Page Language="C#" %>
<script runat="server">
  void Page_Load()
```

```
  {
      Response.Write;
      Response.Writes("Hello World")
  }
</script>
```

Listing 3.21: Fehlerhafter Programmblock zur Demonstration einiger Beispiele des Supports von C# durch den Visual Web Developer

Sie sehen, dass einige der Programmzeilen farblich hervorgehoben sind, um die einzelnen Sprachelemente kenntlich zu machen. C#-Schlüsselwörter werden, sobald sie von den im Hintergrund laufenden Validierungsroutinen erkannt werden, farblich hervorgehoben.

Fehlerhafte Zeilen werden ebenso durch ein Unterringeln hervorgehoben.

`Response.Write;`**:**

Die Methode `Response.Write` benötigt in jeder dem System bekannten Implementierung mindestens ein Argument. Dies wird richtigerweise moniert.

`Response.Write("Hello World")`**:**

Es fehlt das abschließende Semikolon.

3.9.2 Die Features des Visual Web Developer Editors

Nach diesem Ausflug in ein Beispiel, was Sie alles an Unterstützung bei Fehlern erfahren können, lassen Sie uns die Schlüsselfunktionen noch einmal zusammenfassen und einzeln erklären:

» Sie haben Zugriff auf alle Eigenschaften, Events und Methoden des .NET Frameworks sowie ergänzter Namespaces. Dies schließt kurze Erklärungen der jeweiligen Funktionalitäten mit ein.

» Codeabschnitte können wie Kapitel in einer Gliederungsansicht versteckt oder angezeigt werden. Dies kann zum Beispiel auf unterschiedlichen Hierarchieebenen erfolgen. (Beachten Sie dabei die +- und –-Schalter in der linken Leiste neben dem Code.)

» Zeileneinzüge, Tabulatoren, Schlüsselwörter, besondere Methoden und das Verhalten von Drag&Drop für Codeblöcke lassen sich über Optionen weitestgehend frei konfigurieren.

» Sie haben die Möglichkeit, Code-Bruchstücke (sogenannte Codeausschnitte) einzufügen, in denen Sie nur noch kleinere Anpassungen vorzunehmen brauchen. Natürlich können Sie Ihre eigenen Bruchstücke definieren und der bereits vorhandenen Sammlung hinzufügen.

» Befehle werden über die IntelliSense®-Technologie erkannt und automatisch ergänzt.

» Im Editor gibt es einen Bereich für das Debugging zum Setzen von Haltepunkten, einen Bereich für das Festsetzen von Lesezeichen, so dass Sie einfach an beliebige, von Ihnen festgelegte Stellen im Programm springen können.

» Sie können zu jedem definierten Schlüsselwort oder Teil eines Namespace per rechtem Mausklick die zugehörige Definition anzeigen lassen.

Schauen wir uns einige dieser Features nun einmal genauer an.

3.9.3 Code erstellen mit IntelliSense-Unterstützung

IntelliSense ermöglicht Ihnen, ohne über weitere Fenster gehen zu müssen, den Zugriff auf eine Vielzahl von Informationen über die von Ihnen gerade verwendeten Sprachelemente und Codeabschnitte.

Sie haben so direkt per Mausklick eine ganze Reihe von Hintergrundinformationen direkt verfügbar, Sie können Vorschläge zur Codeergänzung ausführen oder den Code automatisch ergänzen lassen.

Die Ergänzungen und Kommentare sehen in etwa so aus wie in Abbildung 3.26 dargestellt.

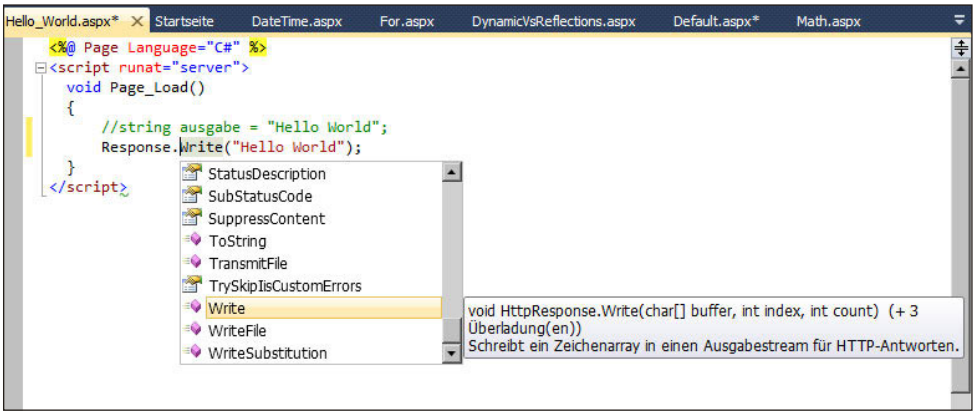


Abbildung 3.26: Code ergänzen mit IntelliSense®

Die Listbox erscheint in dem Augenblick, in dem erkannt wird, dass Sie eine Methode oder Eigenschaften eingeben wollen. Wollen Sie die Liste schon eher sehen, können Sie diese mit Strg+Space einblenden. Sie können durch die gelisteten Methoden und Eigenschaften navigieren, erhalten kurze Erläuterungen und können durch Tab, Strg+Enter oder Doppelklick mit der Maus die Methode vervollständigen.

Wenn Sie Parameter einer Methode ergänzen müssen, haben Sie die Möglichkeit, mittels der Anzeige der einzelnen Parameterelemente eine schnelle (und syntaktisch und semantisch korrekte) Vervollständigung der notwendigen Parameter vorzunehmen. Sie erhalten als Kommentar die notwendigen Hilfen angezeigt. Soweit es möglich ist, verschiedene Parametergruppen anzugeben, erhalten Sie einzelne Auswahlfelder, über die Sie diese Parameterlisten und die zugehörigen ergänzenden Informationen angezeigt bekommen.

Sie können alle diese Optionen variieren und für Ihre Bedürfnisse anpassen. Sie finden die Einstellungen zu C# im Optionsmenü.

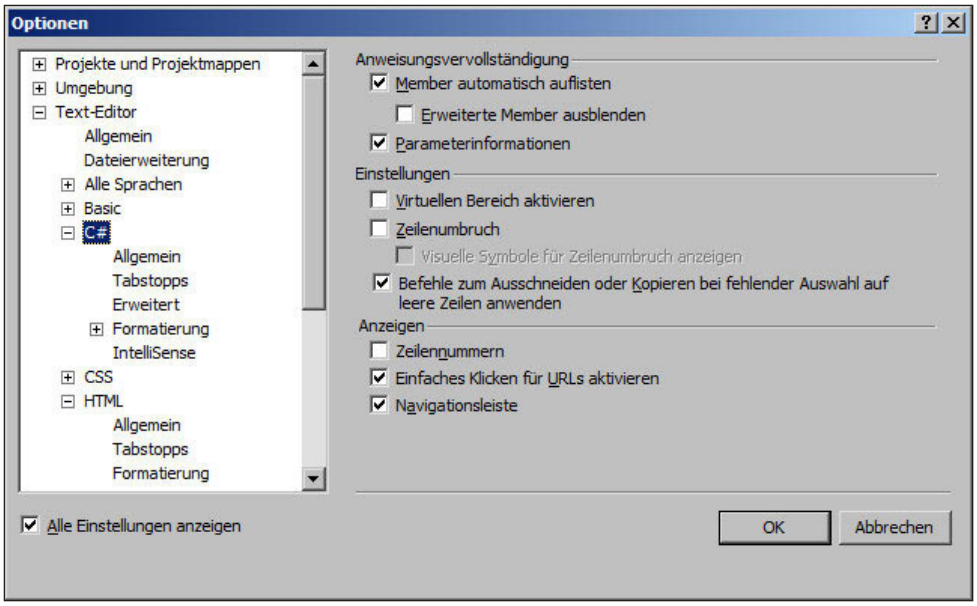


Abbildung 3.27: Optionen für C# im Texteditor des Visual Web Studios Express

Wie Sie in Abbildung 3.26 erkennen können, verdeckt die eingeblendete IntelliSense-Liste den nachstehenden Programmcode. Aus diesem Grunde gibt es seit der Visual Web Developer Version 2008 die Möglichkeit, mit der Strg-Taste die eingeblendete IntelliSense-Liste transparent darzustellen. Durch das Loslassen der Strg-Taste wird die Vorschlagsliste wieder normal dargestellt.

Abbildung 3.28 zeigt eine transparente IntelliSense-Liste.

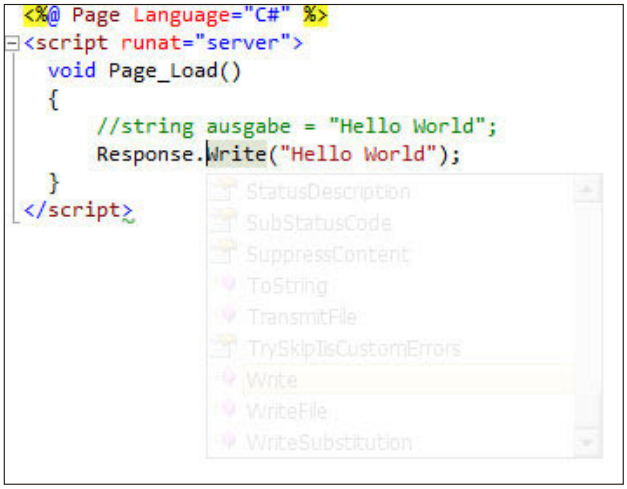


Abbildung 3.28: Transparente IntelliSense-Liste

3.9.4 Neue Features in der Version 2010

Von den vielen Kleinigkeiten, die das Leben in der neuen Visual Web Developer Edition angenehmer machen, wollen wir die aus unserer Sicht wichtigsten Features vorstellen.

Verbessertes IntelliSense

IntelliSense hat in den Vorgängerversionen nur funktioniert, wenn der entsprechende Begriff von Anfang an korrekt geschrieben wurde. Sehr oft hat man jedoch nur einen Teil eines Klassennamens zum Beispiel im Kopf. Das IntelliSense in der aktuellen Version funktioniert auch, wenn der entsprechende Suchbegriff nur partiell angegeben wird.

Abbildung 3.29 zeigt, dass bei der Eingabe von Exce (ich suche nach einer `Exception`) nicht nur Vorschläge in der Liste auftauchen, die mit der entsprechenden Zeichenfolge beginnen, sondern auch solche die diese Zeichenfolge beinhalten.

In Abbildung 3.30 sehen Sie noch eine weitere Verbesserung der IntelliSense-Funktionalität. Klassen-, Methoden und Property-Namen sind laut Konvention von der Schreibweise her Pascal-Casing. Das bedeutet, dass diese Bezeichnungen groß beginnen und bei zusammengesetzten Wörtern der entsprechende Wortanfang auch wieder großgeschrieben wird (z. B. **S**tring**B**uilder). Statt sich durch endlos lange Listen (viele Begriffe beginnen mit String) durchzukämpfen, kann man nun einfach die Anfangsbuchstaben des entsprechenden Bezeichners angeben (bei `StringBuilder` eben `SB`).

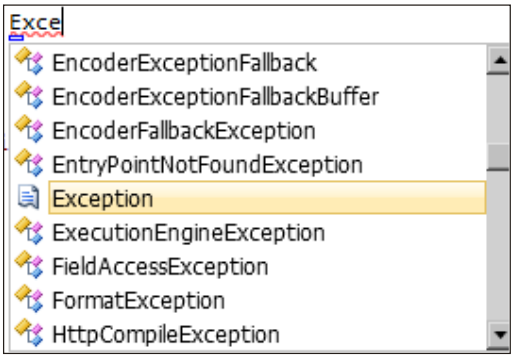


Abbildung 3.29: Verbessertes IntelliSense in Action

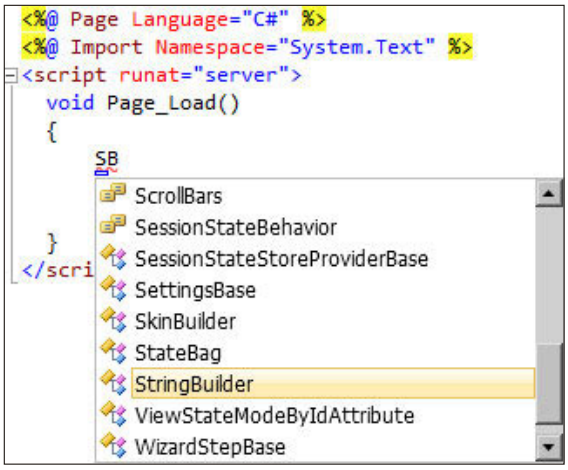


Abbildung 3.30: IntelliSense mit Pascal-Casing

Beachten Sie bitte, dass in diesem Beispiel zuvor auch der Namespace `System.Text`, in dem sich die `StringBuilder`-Klasse befindet, mittels der `Imports`-Direktive importiert wurde.

Zoomfunktionalität

Sie können jetzt im Editor den Code zoomen und somit zum Beispiel für Präsentationen vergrößern. Drücken Sie dazu einfach die Strg-Taste und zoomen Sie mit dem Mausrad, bis die gewünschte Darstellung groß bzw. klein genug ist.

Referenzhervorhebung

Wenn Sie in einer Codedatei eine Methode markieren, werden automatisch alle anderen Vorkommen dieses Methodenaufrufs sowie sofern vorhanden die Definition der Methode innerhalb derselben Codedatei hervorgehoben. Dies illustriert in einem kleinen Beispiel die Abbildung 3.31.

Mit Strg+⇧+↓ bzw. Strg+⇧+↑ können Sie zwischen den Aufrufen hin und her navigieren.

```
<script runat="server">
  void Page_Load()
  {
      HelloWorld();
      HelloWorld();
      //string ausgabe = "Hello World";
      Response.Write("Hello World");
  }

  void HelloWorld()
  {
      Response.Write("Hello World");
  }
</script>
```

Abbildung 3.31: Referenzhervorhebung

Aufrufhierarchie

Wenn Sie auf eine Methode, Eigenschaft oder einen Konstruktor mit der rechten Maustaste klicken, gibt es ein neues Kontextmenü ALLE VERWEISE SUCHEN. Mit dieser Funktion können Sie alle Stellen im Programm anzeigen, wo die entsprechende Methode/Eigenschaft oder der Konstruktor aufgerufen wird.

Mit einem Doppelklick in der Trefferliste wird sofort der entsprechende Aufruf aktiviert.

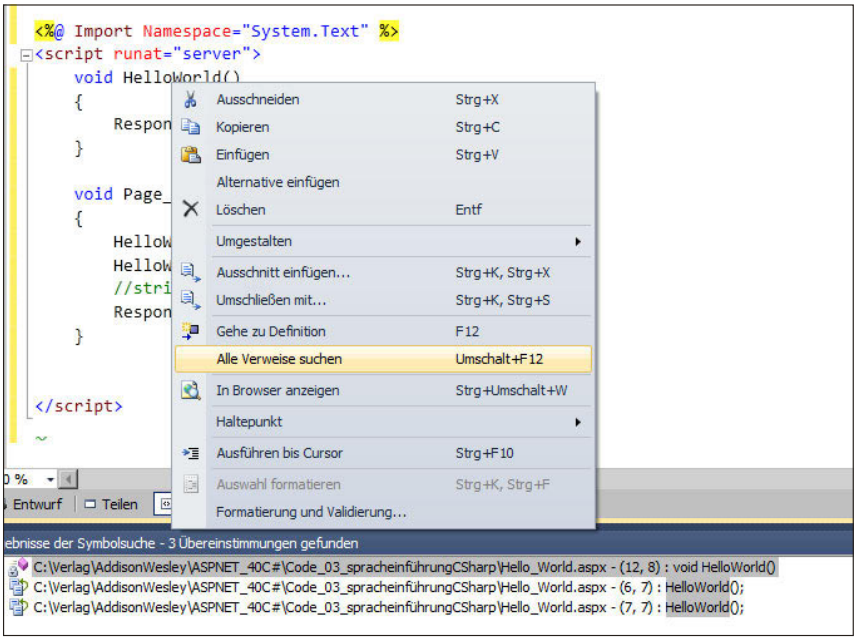


Abbildung 3.32: Alle Aufrufe von HelloWorld in der Aufrufhierarchie

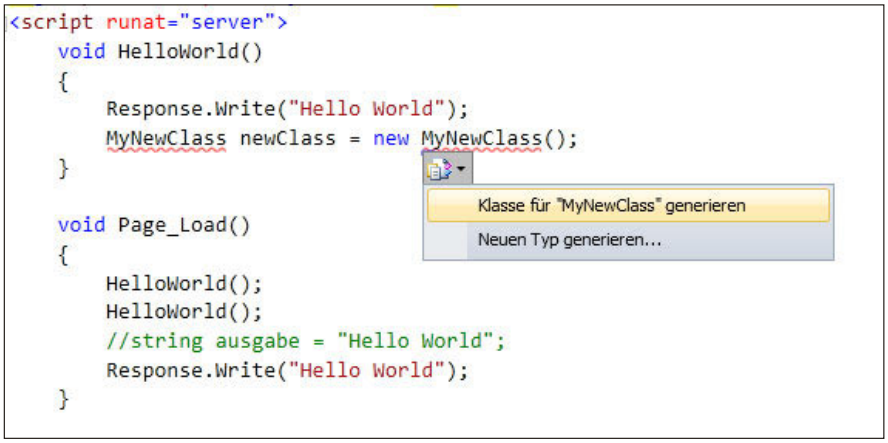
Codegeneration

In den bisherigen Versionen war es möglich, für neue Methoden den entsprechenden Methodenrumpf automatisch anlegen zu lassen. Mit der 2010er Version ist es nun auch möglich, sich Klassen automatisch anlegen zu lassen.

Schreiben Sie einfach dafür den Code so, als würden Sie ein Objekt von einem bestehenden Typ instanzieren, obwohl der entsprechende Typ noch nicht definiert ist. Wenn Sie danach den Klassennamen markieren, ist der erste Buchstabe des Klassennamens unterstrichen, und wenn Sie auf diese Markierung mit der Maus fahren, sehen Sie die Optionen wie in Abbildung 3.33 dargestellt.

Bei der Auswahl von Klasse für Klassenname generieren wird automatisch im *App_Code*-Ordner eine entsprechende Codedatei angelegt.

Bei der Auswahl von Neuen Typ generieren... können Sie die entsprechenden Optionen wie in Abbildung 3.34 dargestellt auswählen, um Einfluss auf die Generierung der Klassendatei zu haben.




```
<script runat="server">
    void HelloWorld()
    {
        Response.Write("Hello World");
        MyNewClass newClass = new MyNewClass();
    }

    void Page_Load()
    {
        HelloWorld();
        HelloWorld();
        //string ausgabe = "Hello World";
        Response.Write("Hello World");
    }
```

Abbildung 3.33: Anlegen einer neuen Klassendatei

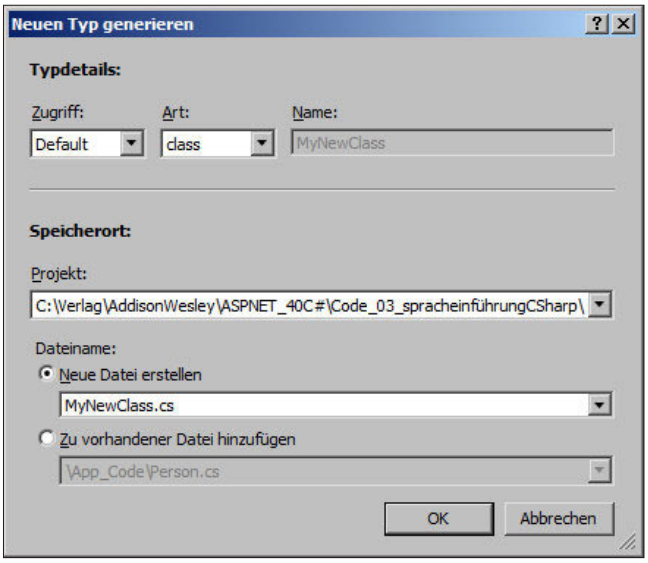


Abbildung 3.34: Dialog bei Auswahl von Neuen Typ generieren

Boxselektion

Mittels der Boxselektion können Sie gleiche Änderungen an mehreren Zeilen Code gleichzeitig vornehmen.

Stellen Sie sich vor, Sie haben innerhalb einer Klasse viele Variablen als `private` definiert und wollen dies auf `internal` ändern. Anstatt diese Änderung jetzt Zeile für Zeile durchzuführen, können Sie dies mit diesem neuen Feature auf einmal machen. Markieren Sie dabei den gewünschten Bereich mit Shift + Alt und den entsprechenden Pfeiltasten und führen Sie die gewünschten Änderungen durch. Dieses Vorgehen ist in der folgenden Abbildung 3.35 und Abbildung 3.36 dargestellt.

```
public class Class1
{
    private int value1;
    private int value2;
    private int value3;
    private int value4;
    private int value5;
    private int value6;
```

Abbildung 3.35: Auswahl der Änderung in der Box

```
public class Class1
{
    internal int value1;
    internal int value2;
    internal int value3;
    internal int value4;
    internal int value5;
    internal int value6;
```

Abbildung 3.36: Durchgeführte Änderung in der Box

3.9.5 Dokumentation von Programmcode

Sie haben in C# eine ganze Reihe von Möglichkeiten, Programmcode zu dokumentieren. Zum einen kommen einfache Elemente zum Einsatz, mit denen Sie einzelne Programmzeilen oder Programmabschnitte mit Kommentaren versehen können, zum anderen bietet C# auch eine Fülle von XML-Tags an, die Sie in Ihren Code einfügen können, um eine Dokumentation erstellen zu können. Beide Möglichkeiten wollen wir Ihnen nachfolgend vorstellen.

Basis-Programmdokumentation

Als Basis-Programmdokumentation steht Ihnen die Verwendung von Kommentaren im Programmtext zur Verfügung. Hierauf wollen wir nicht weiter eingehen. Nur so viel: Ein gut dokumentiertes Programm erhöht die Nachvollziehbarkeit von Programmcode auch nach einem halben Jahr oder später.

Programmdokumentation mit XML-Unterstützung

Die Kommentarzeilen, in denen die XML-Dokumentation verborgen wird, müssen durch drei vorangestellte /// kenntlich gemacht werden. Wenn Sie dies gemacht haben, werden alle gültigen XML-Tags erkannt. Microsoft hat eine Liste von XML-Tags zusammengestellt, die als Empfehlung aufgeführt sind. Einige dieser empfohlenen Tags werden vom C#-Compiler sogar auf ihre syntaktische Korrektheit hin überprüft.

XML-Tag	Erklärung
`<c>` `</c>`	Programmcode wird durch diese XML-Tags gekennzeichnet. Der durch die Tags eingeschlossene Text muss sich innerhalb einer Kommentarzeile befinden.
`<code>` `</code>`	Code wird gekennzeichnet. Der eingeschlossene Block kann sich über mehrere Zeilen erstrecken.
`<example>` `</example>`	Mit diesen Tags schließen Sie üblicherweise einen Abschnitt, der ein Codebeispiel und eine Erklärung hierzu enthält, ein.
`<exception cref= "`*Bezeichner*`">` `</exception>` (Syntaxprüfung)	Ausnahmen werden durch dieses Tag markiert. Über `cref` wird in Anführungszeichen die beschriebene Ausnahmemethode angegeben.

XML-Tag	Erklärung
<include file="*Dateiname*" path="*TagPfad*[@name='*id*']" /> (Syntaxprüfung)	Über diese Tags lässt sich ein eigenständiges Dokumentationsfile mit Pfadangabe einbinden. Die Parameter file und path legen den Dateinamen und den Pfad der einschließenden XML-Tags im Dokument fest. Innerhalb des innersten Tags muss ein ID-Parameter mitgegeben werden. Parameterbezeichner ist @name.
<list type="*Typ*"> </list>	Über dieses Tag wird eine Liste spezifiziert. Die Art der Liste wird über type festgelegt. Es stehen bullet (für eine Liste von Bullet-Points), number (für eine nummerierte Liste) oder table (für eine Tabelle) zur Verfügung. Wenn Sie eine Tabelle aufbauen, können Sie eine einzelne Zelle mit dem Tag <term> kennzeichnen.
<para> </para>	Über dieses Tag erzeugen Sie eine Formatierung als Paragraph für einen einzelnen Abschnitt. Üblicherweise sollte dieses Tag verwendet werden, um innerhalb von übergeordneten Tags eine formale Strukturierung zu erreichen.
<param name="*Name*"> </param> (Syntaxprüfung)	Mit diesem Tag werden Methodenparameter für die Dokumentation kenntlich gemacht. Es empfiehlt sich, dieses Tag in der Dokumentation von Methodendeklarationen zu verwenden. IntelliSense wertet dieses Tag ebenfalls aus.
<paramref name="*Name*" />	Markierung eines Tags, der im Namen festgelegt wird, als Parameter. Auf diesen Namen kann dann eine besondere Formatierung bei Bedarf angewendet werden.
<permission cref= "*Element*"> </permission> (Syntaxprüfung)	Für das in *Element* beschriebene Element werden Informationen über die Zugriffsrechte bereitgestellt. Dabei wird vom Compiler das Element auf Existenz geprüft.
<remarks> </remarks>	Die darüber erstellten Kommentare werden auch im Objekt-Browser und bei IntelliSense angezeigt.
<returns> </returns>	Rückgabewerte werden durch dieses Tag gekennzeichnet.
<see cref="*Element*" /> (Syntaxprüfung)	Referenzierung auf ein anderes Element (als Link), das in die Dokumentation eingefügt wird. Der Compiler überprüft bei der Erstellung, dass dieses Element existiert.
<seealso cref="*Element*" /> (Syntaxprüfung)	Referenzierung auf ein anderes Element (als Test), das in die Dokumentation eingefügt wird. Der Compiler überprüft bei der Erstellung, dass dieses Element existiert.
<summary> </summary>	Strukturierungselement zum Kenntlichmachen einer Objektbeschreibung. Diese Beschreibung wird über IntelliSense und den Objekt-Browser erkannt und angezeigt.
<typeparam name="*Name*"> </typeparam> (Syntaxprüfung)	Markierung und Erklärung eines Typparameters, der dokumentiert werden soll. Der Name des Parameters wird in *Name* festgelegt und vom Compiler geprüft.
<value> </value>	Beschreibung einer Eigenschaft (bzw. eines Werts)

Tabelle 3.16: Übersicht Schlüsselwörterfür XML-Dokumentation

Wenn Sie übrigens das Kleiner-Zeichen oder das Größer-Zeichen in Ihrer Dokumentation verwenden wollen, benutzen Sie die entsprechenden HTML-Tags (also `<` bzw. `>`).

Sie erzeugen Ihre Dokumentation, indem Sie Ihr Dokument mit der Compiler-Option `/doc` übersetzen.

3.10 Fazit

In diesem Abschnitt haben Sie einen Einblick in die wesentlichen Elemente von C# erhalten. Es sind unter anderem besprochen worden:

» Datentypen

» Operatoren

» Kontrollstrukturen

» Schleifen

» Prozeduren und Funktionen

» Objektorientierte Elemente

» Fehler und Ausnahmebehandlung

» Die Erstellung einer Dokumentation

Auch einen Einblick in die Bedienung des Visual Web Developers haben Sie erhalten. Die Unterstützung des Visual Web Developers für C# wurde dargestellt.

Nebenher haben wir einige Klassen des .NET Frameworks kurz vorgestellt, die Ihnen die Programmierarbeit erleichtern werden.

Nun verfügen Sie über das Rüstzeug, sich mit den Dingen zu befassen, um die es eigentlich in diesem Buch gehen soll: um die Programmierung mit ASP.NET 4.0.