Kapitel 1
Allgemeine Einführung in .NET

1.1 Warum .NET?

Einem Leser, der über fundierte Grundlagenkenntnisse verfügt, eine Thematik nahezubringen, die seine Aufmerksamkeit erregt und ihm neue Kenntnisse vermittelt, ist ein nicht ganz einfaches Unterfangen. Dabei gleichzeitig einen Programmieranfänger behutsam in die abstrakte Denkweise der Programmlogik einzuführen, ohne zugleich Frust und Enttäuschung zu verbreiten, dürfte nahezu unmöglich sein. Ich versuche mit diesem Buch dennoch, diesen Weg zu beschreiten, auch wenn es manchmal einer Gratwanderung zwischen zwei verschiedenen Welten gleicht. Dabei baue ich schlicht und ergreifend auf den jahrelangen Erfahrungen auf, die ich als Trainer in vielen Seminaren mit teilweise ausgesprochen heterogenen Gruppen erworben habe.

Vielleicht wissen Sie überhaupt noch nicht, was sich hinter .NET verbirgt? Vielleicht haben Sie sich für dieses Buch entschieden, ohne die Tragweite Ihres Entschlusses für .NET zu kennen. Ich möchte Ihnen das zunächst einmal erläutern.

Blicken wir ein paar Jahre zurück, sagen wir in die 90er-Jahre, und stellen wir uns die Frage, wie damals Anwendungen entwickelt wurden und wie sich die IT-Welt während dieser Zeit entwickelt hat. Am Anfang des von uns betrachteten Jahrzehnts war der Hauptschauplatz der Desktop-PC, Netzwerke steckten noch mehr oder weniger in den Kinderschuhen. Grafische Benutzeroberflächen hielten langsam Einzug auf den Rechnern, das Internet war einem nur mehr oder weniger elitären Benutzerkreis bekannt und zugänglich. Desktop-PCs wurden mit immer besserer Hardware ausgestattet; ein Super-PC von 1990 galt zwei Jahre später als total veraltet und musste wegen der gestiegenen Anforderungen der Software an die Hardware oft zumindest drastisch aufgerüstet, wenn nicht sogar ersetzt werden.

Sie merken vielleicht an diesen wenigen Worten, wie dramatisch sich die IT-Welt seitdem verändert hat. Die Evolution betraf aber nicht nur Software und Hardware. Software muss, ehe sie den Benutzer bei seiner täglichen Arbeit unterstützen kann, entwickelt werden. Hier kochten viele Unternehmen ein eigenes Süppchen und warben bei den Entwicklern und Entscheidungsträgern mit Entwicklungsumgebungen, die zum einem auf den unterschiedlichsten Programmiersprachen aufsetzten und zudem mit eigenen Funktionsbibliotheken aufwarteten: Borlands Delphi, Microsofts Visual Basic, für die Puristen C und C++ – um nur die bekanntesten Vertreter zu nennen.

Die Vielfalt betraf jedoch nicht nur die Entwicklung der Software. Immer neue Plattformen, angepasst an den jeweils aktuellen Trend der Zeit, eroberten den Markt und verschwanden nicht selten auch schnell wieder. Die Unternehmensnetzwerke mussten mit der stürmischen Entwicklung Schritt halten, wurden komplexer und komplizierter und öffneten sich zunehmend auch der Welt nach außen.

In dieser Periode begann auch der Siegeszug des Internets. Obgleich es anfangs nur als weltweiter Verteiler statischer Dateninformationen positioniert war, wurden immer mehr Technologien ausgedacht, die die statischen Webseiten durch dynamische ersetzten, die dem Anwender nicht immer dieselben Informationen bereitstellten, sondern genau die, für die er sich interessierte. Datenbanken wurden hinter die Webserver geschaltet und fütterten die Webseiten mit dem aktuellsten Informationsstand.

Kluge Köpfe erkannten auch sehr schnell, dass die Spezifikationen des Internets sich auch dazu eignen, mehrere Unternehmen zu koppeln. Damit wurde die Grundlage dafür geschaffen, dass Sie heute im Reisebüro oder im Internetbrowser eine Reise buchen können, die nicht nur den Flug, sondern gleichzeitig eine gültige Hotelzimmerbuchung, vielleicht sogar samt Mietwagen, umfasst – obwohl hierzu schon drei Informationsquellen mit unterschiedlicher Software abgezapft werden müssen: ein nicht ganz einfaches Unterfangen, wenn Sie bedenken, dass möglicherweise die Schnittstellen, über die die verschiedenen Komponenten sich zwangsläufig austauschen müssen, nicht einheitlich definiert sind.

Bei dieser rasanten Entwicklung der Möglichkeiten, Daten auszutauschen oder auch nur weiterzuleiten, sollten Sie nicht vergessen, dass auch die Hardware eine ähnliche Entwicklung genommen hat. Ein Phone besitzen heutzutage schon die meisten schulpflichtigen Kinder, und der mobile Sektor entwickelt sich immer noch dramatisch schnell. Ein Ende dieser Spirale ist derzeit nicht zu erkennen.

An der Schnittstelle all dieser Vielfältigkeit steht der Entwickler – was nutzen die beste Hardware und die ausgeklügelten Spezifikationen, wenn die Bits sich nicht den Weg von einem zum anderen Endpunkt bahnen? Für diesen Bitfluss wollen Sie als Entwickler sorgen. Damit fangen aber wegen der oben erwähnten Vielgestaltigkeit der IT-Welt die Probleme an: verschiedene Plattformen, unterschiedliche Programmiersprachen, mehrere Klassenbibliotheken, eine Vielzahl zu beachtender Spezifikationen usw.

Einen ersten Schritt in Richtung Vereinheitlichung beschritt die Firma Sun mit Java. Der Erfolg, den diese plattformunabhängige Sprache hatte und auch immer noch hat, war auch ein Zeichen für Microsoft, um das Entwicklerterrain zu kämpfen. Nach einer eingehenden Analyse der Anforderungen, die gegen Ende der 90er-Jahre an die damalige Software gestellt wurden, sowie einer Trendanalyse der Folgejahre wurde das .NET Framework entwickelt. Dabei konnte Microsoft die »Gunst der späten Stunde« nutzen und die Nachteile und Schwachpunkte, die jedes Produkt, also auch Java, hat, durch neue Ideen ausmerzen.

Nein, .NET ist natürlich auch kein Heilsbringer und wird sicherlich nicht die Menschheit überdauern. Aber nach heutigen Maßstäben ist .NET das wahrscheinlich effizienteste Frame-

work, in dessen Mittelpunkt die .NET Klassenbibliothek steht. Diese bietet Ihnen alles, was Sie zum Entwickeln brauchen – egal, ob es sich um eine einfache Anwendung handelt, die nur ein paar Daten anzeigt, oder um eine Unternehmensanwendung großen Stils. Sie können Desktop-Anwendungen genauso erstellen wie eine hochkomplexe Internetanwendung. Sie können die Microsoft Office-Produkte damit programmieren, fremde Datenquellen anzapfen, Programme für Ihr Phone schreiben und vieles mehr. Dazu müssen Sie sich nicht immer wieder in neue Programmiersprachen und neue Entwicklungsumgebungen einarbeiten, denn alles ist wie aus einem Guss.

Ich möchte jetzt nicht den Eindruck vermitteln, dass alles ganz einfach ist und Sie demnächst ganz tolle Anwendungen mit den tollsten Features präsentieren können. Dafür ist die .NET-Klassenbibliothek einfach zu umfangreich. Aber Sie können sich darauf verlassen, dass Sie sich nun auf das Wesentliche Ihrer Arbeit konzentrieren können: Sie arbeiten unabhängig vom Typ der zu entwickelnden Anwendung immer in derselben Umgebung, zum Beispiel mit Visual Studio 2019. Sie brauchen sich nicht immer wieder aufs Neue in andere Programmiersprachen einzuarbeiten, sondern können auf gewonnene Kenntnisse aufsetzen. Und Ihnen werden alle Mittel an die Hand gegeben, um auf wirklich einfachste Weise mit fremden Anwendungen zu kommunizieren, wenn sich diese an bestimmten, allgemein anerkannten Spezifikationen orientieren.

Eine Funktionssammlung (eigentlich müsste ich an dieser Stelle richtigerweise von einer Klassenbibliothek sprechen) ist nur so gut, wie sie auch zukünftige Anforderungen befriedigen kann. Dass .NET hier architektonisch den richtigen Weg beschritten hat, beweist die derzeit aktuelle Version 4.7.x.

Genau an dieser Stelle darf ich Ihnen natürlich auch den großen Haken nicht verschweigen, den die ansonsten so hervorragende Umgebung hat: Sie werden mit Sicherheit niemals alle Tiefen von .NET ergründen. Als jemand, der von der ersten Beta-Version mit dabei war, muss ich sagen, dass ich mich immer wieder aufs Neue davon überraschen lassen muss, welche Fähigkeiten in der .NET-Klassenbibliothek schlummern. Verabschieden Sie sich von der Idee, jemals alle Klassen mit ihren Fähigkeiten erfassen zu können. Dafür ist die Klassenbibliothek ist einfach zu mächtig.

1.1.1 Die Zukunft von .NET

.NET lebt, aber das .NET Framework ist mit Visual Studio 2019 in seine letzte Runde eingetreten. Warum das, werden Sie sich an dieser Stelle fragen.

Wie viele Dinge im Leben, bedarf .NET einer Grunderneuerung. Viele Bibliotheken sind veraltet und wurden durch neue Bibliotheken ersetzt. Viele APIs (Programmierschnittstellen) sind obsolet, sollten also nicht mehr verwendet werden. Allerdings wurde bisher der alte »Ballast« immer mitgeschleppt. Zudem hat sich sehr viel in der Welt der Programmierung getan: Neben Windows werden nunmehr auch andere Betriebssysteme wie macOS und Linux von Microsoft akzeptiert. Darüber hinaus dürfen wir nicht übersehen, dass der mobile

Sektor in allen Bereichen Einzug gehalten hat: Android hat beispielsweise in Deutschland einen Marktanteil von 71 %, iOS liegt bei ungefähr 25 %. Den Trend zu alternativen Betriebssystemen konnte Microsoft nicht weiter ignorieren. Das äußerte sich einerseits in der Übernahme von Xamarin (ein Tool, mit dem sich Windows-Phone-, iOS- und Android-Apps mit C# entwickeln lassen), aber auch die Einführung von .NET Core. Und damit ist auch bereits das wichtige Stichwort gefallen: .NET Core. .NET Core beschreibt Klassenbibliotheken, die sowohl unter Windows, aber auch unter Linux und macOS lauffähig sind. Gleichzeitig wird der alte Ballast über Bord geworfen, .NET Core ist also schlanker als das .NET Framework.

Welche Konsequenzen wird .NET Core auf die Entwicklung einer Software mit C# ausüben? Die Antwort ist recht einfach, nämlich (fast) keine. Sie werden weiterhin Ihre Programme in Visual Studio entwickeln und die Sprache C# nutzen. Möglicherweise werden Sie sich Core-spezifische Sprachergänzungen ansehen, die nur im Umfeld von .NET Core genutzt werden können. Das betrifft bereits die aktuelle Version C# 8.0, wenn auch in sehr geringem Umfang. Was ich Ihnen in diesem Buch zeigen werde, hat also Zukunft.

.NET Core existiert derzeit in der Version 2.0. Ursprünglich war geplant, .NET Core 3.0 zusammen mit Visual Studio 2019 zu veröffentlichen. Das hat aber nicht geklappt. Momentan (April 2019) ist die Veröffentlichung des Releases auf die zweite Hälfte 2019 datiert. In Kapitel 18, »Die Zukunft: .NET Core und .NET Standard«, werde ich auf .NET Core eingehen und darüber hinaus den neu definierten .NET Standard erklären.

1.1.2 Ein paar Worte zu diesem Buch

Mit der Einführung von .NET im Jahr 2002 änderte sich die Philosophie der Anwendungsentwicklung – zumindest im Hause Microsoft. Die Karten wurden neu gemischt, denn das architektonische Konzept neu. Da .NET von seinem Ansatz her grundsätzlich plattformunabhängig ist, ähnlich wie Java, zeigte Microsoft gleichzeitig zum ersten Mal ernsthaft die Akzeptanz anderer Plattformen. Ehrlicherweise muss man jedoch auch anmerken, dass die Plattformunabhängigkeit von Microsoft anfangs praktisch nicht weiter unterstützt wurde. Eine kleine Gruppe von Enthusiasten verschrieben sich dem Mono-Projekt und versuchten damit, eine .NET-Basis für Linux zu schaffen, konnten aber mit der rasanten Entwicklung von .NET nicht Schritt halten.

.NET ist 100%ig objektorientiert. Das ist Fakt. Obwohl das objektorientierte Programmieren schon seit vielen Jahren in vielen Sprachen eingeführt worden ist, sind nicht alle professionellen Entwickler in der Lage, auf dieser Basis Programme zu entwickeln. Teilweise sträuben sie sich sogar mit Händen und Füßen gegen die Denkweise in Klassen und Objekten, denn ihre Denkweise ist zu sehr in der prozeduralen Programmierung verwurzelt.

Es spielt keine Rolle, ob man einfachste Programme zur Konsolenausgabe entwickelt, lokale Windows-Anwendungen oder Applikationen für das Internet – immer spielen Klassen und Objekte die tragende Rolle. Daher ist es unumgänglich, zunächst die Grundlagen einer .NET-

Entwicklungssprache einschließlich des objektorientierten Ansatzes zu beherrschen, bevor man sich in das Abenteuer visualisierter Oberflächen stürzt.

In diesem Buch möchte ich Ihnen diese notwendigen Grundlagen fundiert und gründlich vermitteln und danach zeigen, wie Sie mit der *Windows Presentation Foundation* (*WPF*) Windows-Anwendungen entwickeln und wie Sie mit dem *Entity Framework* auf SQL-Datenbanken zugreifen können. Das Buch ist in Kapitel aufgeteilt, die thematisch und logisch aufeinander aufbauen. Jedes Kapitel enthält wiederum einzelne Abschnitte, die ein untergeordnetes Thema abgrenzen. Die Gliederung könnte man wie folgt beschreiben:

- Einführung in die Entwicklungsumgebung
- die Sprachsyntax von Visual C# einschließlich des objektorientierten Ansatzes
- die wichtigsten .NET-Klassenbibliotheken
- Datenzugriffe mit dem Entity Framework
- die Entwicklung einer grafischen Benutzerschnittstelle mit der Windows Presentation Foundation (WPF)
- Komponententests mit MSTest

In diesem Kapitel werde ich zuerst die elementaren Grundlagen von .NET erörtern. Zwangsläufig fallen deshalb schon im ersten Kapitel Begriffe, die Ihnen möglicherweise zu diesem Zeitpunkt nicht sehr viel sagen. Ich gebe gern zu, auch ich hasse Bücher, die sich zunächst ausgiebig über eine Technologie auslassen, mit Fachbegriffen jonglieren und sich erst nach einigen frustrierenden Seiten dem eigentlichen Thema widmen. Dennoch ist es unumgänglich, zuerst den Kern von .NET mit seinen Vorteilen für den Programmierer zu erläutern, bevor man sich mit der Sprache auseinandersetzt. Allerdings werde ich mir Mühe geben, Sie dabei nicht allzu sehr zu strapazieren, und mich auf das beschränken, was für den Einstieg als erste Information unumgänglich ist. Lassen Sie sich also nicht entmutigen, wenn ein Begriff fällt, den Sie nicht zuordnen können, und lesen Sie ganz locker weiter – in diesem Buch werde ich nichts als bekannt voraussetzen, Sie werden alles noch intensiv lernen.

Bevor wir uns ab Kapitel 2 der Sprache C# widmen, werde ich die überarbeitete Entwicklungsumgebung Visual Studio 2019 vorstellen (die übrigens jetzt auch mit der WPF gestaltet wurde). Wenn Sie mit einer alten Version von Visual Studio gearbeitet haben, werden Sie sicherlich schnell mit der neuen vertraut, obwohl sich in der neusten Version das Layout deutlich verändert hat. Sollten Sie keine Erfahrungen mitbringen, dürften am Anfang einige Probleme mit dem Handling auftreten. Dazu kann ich Ihnen nur einen Rat geben: Lassen Sie sich nicht aus der Fassung bringen, wenn sich »wie von Geisterhand« klammheimlich plötzlich ein Fenster in die Entwicklungsumgebung scrollt oder Sie die Übersicht verlieren – vor den Erfolg haben die Götter den Schweiß gesetzt.

In Kapitel 2 beginnen wir mit dem eigentlichen Thema dieses Buches. Ich stelle Ihnen die Syntax der Sprache C# 8.0 vor, lasse dabei aber noch sämtliche Grundsätze des objektorientierten Ansatzes weitestgehend außer Acht. Sie sollen zunächst lernen, Variablen zu dekla-

rieren, mit Daten zu operieren, Schleifen zu programmieren usw. In Kapitel 3 bis Kapitel 18 wenden wir uns ausführlich dem objektorientierten Ansatz zu und werden auch ein paar besondere Technologien beleuchten.

Diese Kapitel gehören sicherlich zu den wichtigsten in diesem Buch, denn Sie werden niemals eine .NET-basierte Anwendung entwickeln können, wenn Sie nicht in der Lage sind, klassenorientierten Code zu lesen und zu schreiben.

Datenbanken spielen in nahezu jeder Anwendung eine wichtige Rolle. In Kapitel 19 bis Kapitel 21 werden wir uns daher mit dem Entity Framework beschäftigen. Anschließend stelle ich Ihnen die *Windows Presentation Foundation* (*WPF*) vor. Mit dieser Programmierschnittstelle können Sie Windows-Anwendungen entwickeln, basierend auf der Beschreibungssprache *XAML*.

Vielleicht werden Sie sich fragen, wo denn *ASP.NET-Webanwendungen, ASP.NET-Webdienste, .NET-Remoting*, *Windows Communication Foundation* (*WCF*) usw. ihre Erwähnung finden. Meine Antwort dazu lautet: Nirgendwo in diesem Buch. Denn schauen Sie sich nur den Gesamtumfang des Buches an, das Sie gerade in den Händen halten. Die Themen, die darin beschrieben sind, werden nicht nur oberflächlich behandelt, sondern gehen oft auch ins Detail. Es bleibt also kein Platz mehr für die anderen Technologien.

1.1.3 Die Beispielprogramme

Begleitend zu der jeweiligen Thematik werden wir in jedem Kapitel Beispiele entwickeln, die Sie auf *www.rheinwerk-verlag.de/4699* unter Materialien zum Buch finden. Im Buch sind diese Beispiele am Anfang des Quellcodes häufig wie folgt gekennzeichnet:

```
\\Beispiel: ..\Kapitel 3\GeometricObjectsSolution_1
```

Eine allgemeine Bemerkung noch zu den Beispielen und Codefragmenten: Als Autor eines Programmierbuches steht man vor der Frage, welchen Schwierigkeitsgrad die einzelnen Beispiele haben sollen. Werden komplexe Beispiele gewählt, liefert man häufig eine Schablone, die in der täglichen Praxis mit mehr oder weniger vielen Änderungen oder Ergänzungen übernommen werden kann. Andererseits riskiert man damit aber auch, dass mit der Komplexität der Blick des Lesers für das Wesentliche verloren geht und schlimmstenfalls die Beispiele nicht mit der Intensität studiert werden, die zum Verständnis der Thematik erforderlich wäre.

Ich habe mich für einfachere Beispielprogramme entschieden. Einen erfahrenen Entwickler sollte das weniger stören, weil er sich normalerweise mehr für die Möglichkeiten der Sprache interessiert, während für einen Einsteiger kleine, überschaubare Codesequenzen verständlicher und letztendlich auch motivierender sind.

1.2 .NET unter die Lupe genommen

1.2.1 Das Entwicklerdilemma

Mit .NET hat Microsoft im Jahr 2002 eine Entwicklungsplattform veröffentlicht, die inzwischen von vielen Entwicklungsteams akzeptiert und auch eingesetzt wird. Kommerzielle Gründe spielten für Microsoft sicherlich auch eine Rolle, damals einen Neuanfang in der Philosophie seiner Softwareentwicklung herbeizuführen.

In den Jahren zuvor hatte sich bereits abgezeichnet, dass sich die Ansprüche an moderne Software grundlegend ändern würden. Das Internet spielte dabei wohl die wesentlichste Rolle, aber auch die Anforderung, dem erhöhten Aufkommen clientseitiger Anfragen an einen Zentralserver durch skalierbare Anwendungen zu begegnen. Der Erfolg von Java, das sich in den Jahren zuvor als eine der bedeutendsten Programmiersprachen etablierte, mag der Beweis dafür sein, denn Java spielt seine Stärken in erster Linie bei der Entwicklung webbasierter und verteilter Anwendungen aus.

Die damaligen Probleme waren nicht neu, und Technologien gab es bereits schon länger – auch bei Microsoft. Mit COM/COM+ ließen sich zwar auch vielschichtige und skalierbare Anwendungen entwickeln, aber unzweifelhaft war die Programmierung von COM+ wegen der damit verbundenen Komplexität als nicht einfach zu bezeichnen. Es gibt nicht sehr viele Entwickler, die von sich behaupten können, diese Technologie »im Griff« gehabt zu haben. Damit trat auch ein Folgeproblem auf, denn grundsätzlich gilt: Je komplizierter eine Technologie ist, desto fehleranfälliger wird die Software. Man muss nicht unbedingt ein Microsoft-Gegner sein, um zu sagen, dass selbst der Urheber dieser Technologien diese oft nur unzureichend in den hauseigenen Produkten umsetzt.

Die Aussage, dass die Vorteile der .NET-Systemplattform nur der Entwicklung verteilter Systeme wie dem Internet zugutekommen, beschreibt ihre Möglichkeiten nur völlig unzureichend. Selbstverständlich lassen sich auch einfache Windows- und Konsolenanwendungen auf Basis von .NET entwickeln. Die Vorteile beziehen sich aber nicht nur auf Anwendungen selbst, sondern lösten auch ein Dilemma der Entwickler. Die Entscheidung für eine bestimmte Programmiersprache war in der Vergangenheit fast schon eine Glaubensfrage gewesen – nicht nur, was die Programmiersprache anging, denn die Festlegung auf eine bestimmte Sprache war auch die Entscheidung für eine bestimmte Funktions- bzw. Klassenbibliothek.

Windows-Programme basieren alle auf der Systemschnittstelle einer Funktionssammlung, die als *Win32-API* bezeichnet wird. Da diese Funktionssammlung einige tausend Funktionen enthält, wurden verwandte Funktionalitäten in Klassen zusammengeführt und konnten über Methodenaufrufe angesprochen werden. Dieses Prinzip vereinfachte die Programmierung deutlich, aber bedauerlicherweise gab es nicht eine einzige, sondern gleich mehrere herstellerspezifische Klassenbibliotheken, die zwar ein ähnliches Leistungsspektrum aufwiesen, aber grundlegend anders definiert waren. Die *Microsoft Foundation Classes* (*MFC*) für

Visual C++ ist die Klassenbibliothek von Microsoft, und Borland-Inprise kochte mit der *Object Windows Library* (*OWL*) ein eigenes Süppchen. Der Wechsel von einer Programmiersprache zu einer anderen bedeutete in der Regel auch, sich in eine andere Bibliothek einzuarbeiten. Beides kostet nicht nur sehr viel Zeit, sondern bedeutet auch finanziellen Aufwand.

Es mag fast erstaunen (oder auch nicht) – es gibt neben Windows tatsächlich andere Betriebssysteme, denen man durchaus auch eine Existenzberechtigung zuschreiben muss. Die Entwickler von Java haben das schon vor Jahren erkannt und mit der *Virtual Machine* (*VM*) eine Komponente bereitgestellt, die auf verschiedene Betriebssystemplattformen portiert werden kann. Dies ist einer der größten Vorteile von Java und hat sicherlich viele Entscheidungsträger in den Unternehmen beeinflusst. Code lässt sich auf Windows-Plattformen entwickeln und auf einer Unix-Maschine installieren – ein reizvoller Gedanke, Investitionen von einem bestimmten System zu lösen und sie nicht daran zu binden.

1.2.2 .NET – ein paar allgemeine Eigenschaften

Es ist kein Zufall, dass ich im vorherigen Abschnitt öfter Java erwähnt habe. Wenn Sie das Konzept von Java kennen oder vielleicht in der Vergangenheit sogar mit Java programmiert haben, werden Sie sehr viele Parallelitäten zu .NET sehen. Microsoft ist in der Vergangenheit sicher nicht entgangen, worauf der Erfolg von Java zurückzuführen ist. In Kenntnis der Fakten hat man die Idee, die hinter Java steckt, übernommen und dabei versucht, die bekannten Schwachstellen des Ansatzes bzw. der Sprache auszumerzen. Es darf sich bei Ihnen jetzt allerdings nicht die Meinung festigen, .NET sei nur eine Kopie von Java – .NET hat die Messlatte spürbar höher gelegt.

Wir wollen uns nun ansehen, welche wesentlichen programmiertechnischen Neuerungen .NET mit sich bringt.

- **Objektorientierung**
  .NET ist 100%ig objektbasiert und bildet eine konsistente Schicht zur Anwendungsentwicklung. Es gibt keine Elemente, die sich nicht auf Objekte zurückführen lassen. Sogar so einfache Datentypen wie der Integer werden als Objekte behandelt. Auch Zugriffe auf das darunterliegende Betriebssystem werden durch Klassen gekapselt.
- **WinAPI-32-Ersatz**
  Langfristig beabsichtigt Microsoft, die Win32-API durch die Klassen des .NET Frameworks zu ersetzen. Damit verwischen auch die charakteristischen Merkmale der verschiedenen Sprachen. Ob eine Anwendung mit Visual Basic .NET programmiert wird oder mit C# oder C++ – es spielt keine Rolle mehr. Alle Sprachen greifen auf die gleiche Bibliothek zurück; sprachspezifische, operative Bibliotheken gibt es nicht mehr. Die Konsequenz ist, dass die Wahl einer bestimmten Sprache nicht mehr mit der Entscheidung gleichzusetzen ist, wie effizient eine Anwendung geschrieben werden kann oder was sie zu leisten imstande ist.

- **Plattformunabhängigkeit**
  Anwendungen, die auf .NET basieren, laufen in einer Umgebung, die mit der virtuellen Maschine von Java verglichen werden kann, in der erst zur Laufzeit einer Anwendung der Maschinencode erzeugt wird. Die Spezifikation der Laufzeitumgebung (*Common Language Runtime – CLR*) ist keine geheime Verschlusssache von Microsoft, sondern offen festgelegt. In letzter Konsequenz bedeutet das aber auch, dass sich die Common Language Runtime auch auf Plattformen portieren lässt, die nicht Windows heißen, z. B. auf Unix oder Linux. Ehrlicherweise muss man hier anmerken, dass die Plattformunabhängigkeit in der Vergangenheit eher lustlos von Microsoft verfolgt worden ist. Das hat sich spätestens mit Einführung von .NET Core 1.0 verändert.
- **Sprachunabhängigkeit**
  Es spielt keine Rolle, in welcher Programmiersprache eine Komponente entwickelt wird. Eine in C# geschriebene Klasse kann aus VB.NET, F# oder jeder anderen .NET-konformen Sprache heraus aufgerufen werden, ohne den Umweg über eine spezifizierte Schnittstellentechnologie wie COM/COM+ gehen zu müssen. Darüber hinaus lässt sich beispielsweise eine in Visual C# implementierte Klasse auch aus einer VB.NET-Klasse ableiten – oder umgekehrt.
- **Speicherverwaltung**
  Die Freigabe von nicht mehr benötigtem Speicher war schon immer ein Problem. Unter .NET braucht sich ein Entwickler darum nicht mehr zu kümmern, da der im Hintergrund arbeitende Prozess des *Garbage Collectors* diese Aufgaben übernimmt und nicht mehr benötigte Objekte erkennt und automatisch aus dem Speicher entfernt.
- **Weitergabe**
  Ein .NET-Programm weiterzugeben ist viel einfacher geworden – insbesondere im Vergleich zu einem auf COM basierenden Programm, das Einträge in die Registrierungsdatenbank vornehmen muss. Im einfachsten Fall reicht es vollkommen aus, ein .NET-Programm (d. h. eine EXE- oder DLL-Datei) in das dafür vorgesehene Verzeichnis zu kopieren. Darüber hinaus ist aber auch die Verteilung mit einem Installationsassistenten und – ganz neu unter .NET 2.0 – mit ClickOnce möglich.

1.2.3 Das Sprachenkonzept

Die drei Entwicklungssprachen, die in der Vergangenheit hauptsächlich das Bild in der Anwendungsentwicklung prägten, waren C++, Java und Visual Basic 6.0. Seit dem Jahr 2002 und dem Erscheinen des .NET Frameworks 1.0 gesellten sich noch die .NET-Sprachen dazu, allen voran C#.

Betrachten wir jetzt nur die drei zuerst genannten Sprachen. Nehmen wir an, wir würden mit jeder ein einfaches ausführbares Programm schreiben. Wie sehen die Kompilate dieser drei Sprachen aus, und wie werden die drei Kompilate ausgeführt, wenn wir sie auf einen Rechner kopieren, auf dem nur das Betriebssystem installiert ist?

- Nach der Kompilierung des C/C++-Quellcodes erhalten wir eine *.exe*-Datei, die beispielsweise durch einen einfachen Doppelklick im Explorer des frisch installierten Rechners gestartet werden kann. Das Kompilat wird jedoch auf einer anderen Plattform nicht lauffähig sein, denn dazu wäre zuerst eine Neukompilierung erforderlich.
- Eine mit dem VB6-Compiler erzeugte ausführbare Datei kann auf unserer jungfräulichen Betriebssysteminstallation nicht sofort gestartet werden, obwohl die Dateiendung *.exe* lautet. Wir benötigen zur Ausführung einen Interpreter, d. h. das Laufzeitmodul von Visual Basic, der uns den kompilierten Zwischencode in den ausführbaren nativen CPU-Maschinencode übersetzt. Die Portierung eines VB-Programms auf eine andere Plattform ist nicht möglich.
- Java arbeitet prinzipiell ähnlich wie Visual Basic 6.0. Es wird ein Zwischencode generiert, der sogenannte *Bytecode*. Die kompilierten Dateien haben die Dateiendung *.class*. Zur Laufzeit wird dieser Code zuerst durch einen Interpreter geschickt, der als *virtuelle Maschine* (*VM*) bezeichnet wird. Vorausgesetzt, die VM wurde bei der Installation des Betriebssystems installiert, kann man die Java-Anwendung starten. Das Kompilat ist sogar plattformunabhängig und kann auch auf andere Systeme verteilt werden.

Insbesondere die Plattformunabhängigkeit des Kompilats ist bisher ein deutliches Argument für viele Unternehmen gewesen, nicht nur in heterogenen Umgebungen verstärkt auf Java zu setzen.

Entwickeln wir eine .NET-basierte Anwendung, ähnelt der Ablauf der Kompilierung bis zum Start der Laufzeitumgebung dem Ablauf unter Java. Zuerst wird ein Zwischencode erzeugt, der CPU-unabhängig ist. Die Dateiendung lautet *.exe*, wenn wir eine eigenstartfähige Anwendung entwickelt haben. Allerdings ist diese Datei nicht ohne weiteres lauffähig, sie benötigt zur Laufzeit einen »Endcompiler«, der den Zwischencode in nativen, plattformspezifischen Code übersetzt. Der Zwischencode einer .NET-Anwendung wird als *MSIL*-Code (*Microsoft Intermediate Language*) oder nur kurz als *IL* bezeichnet, und der Endcompiler wird *JIT*-Compiler (*Just-In-Time*) oder auch nur kurz *JITter* genannt.

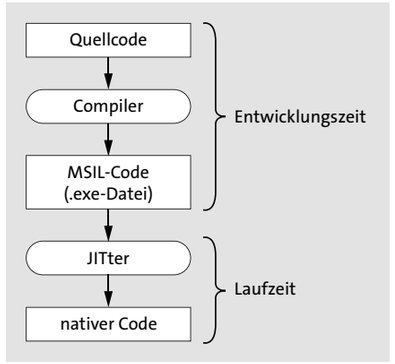

Abbildung 1.1 Der Ablauf der Entwicklung eines .NET-Programms bis zur Laufzeit

1.2.4 Die Common Language Specification (CLS)

Wenn Sie sich in Abbildung 1.1 den Prozessablauf vom Quellcode bis zur Ausführung einer .NET-Anwendung ansehen, müssten Sie sich sofort die Frage stellen, wo der Unterschied im Vergleich zu einer Java-Anwendung zu finden ist – das Diagramm scheint, bis auf die Namensgebung, austauschbar zu sein. Dabei verzichten wir jedoch darauf, andere spezifische Merkmale der beiden Umgebungen zu betrachten, die bei einer genaueren Analyse auch eine Rolle spielen würden.

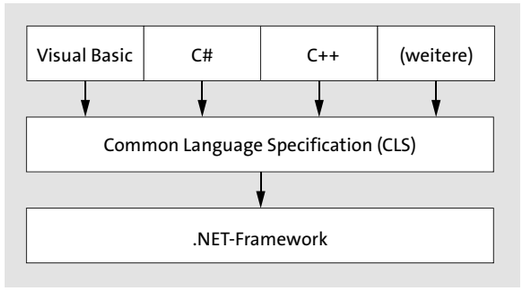

Abbildung 1.2 Die Common Language Specification als Basis der Sprachunabhängigkeit

Vielleicht ist es Ihnen nicht aufgefallen, aber ich habe die Wörter ».NET-Anwendung« und »Java-Anwendung« benutzt – eine kleine Nuance mit weitreichender Konsequenz. Eine Java-Anwendung ist, darauf weist schon der Name hin, mit der Programmiersprache Java entwickelt worden; eine .NET-Anwendung hingegen ist nicht sprachgebunden. Sicher, in diesem Buch werden wir uns mit C# beschäftigen, aber es macht praktisch keinen Unterschied, ob die Anwendung in C#, in Visual Basic .NET oder F# entwickelt worden ist. Ausschlaggebend ist am Ende des Kompiliervorgangs nur ein kompatibler IL-Code, ungeachtet der zugrunde liegenden Sprache.

Um sprachunabhängigen Code erzeugen zu können, muss es Richtlinien geben, an die sich alle .NET-Sprachen halten müssen, um ein Fiasko zu vermeiden. Diese Richtlinien, in denen die fundamentalen Eigenschaften einer .NET-kompatiblen Sprache festgelegt sind, werden durch die *Common Language Specification* (*CLS*) beschrieben. Die Common Language Specification ist ein offener Standard. Das hatte schon frühzeitig zur Folge, dass lange vor der offiziellen Einführung von .NET viele Softwareunternehmen andere Sprachen, beispielsweise Delphi, Eiffel und Cobol, auf .NET portierten.

Wenn alle Sprachen tatsächlich gleichberechtigt sind und dasselbe Ergebnis liefern, stellt sich natürlich die Frage, warum es zukünftig nicht nur eine Sprache gibt. Sogar Microsoft bietet mit C#, F#, C++ und VB .NET in Visual Studio vier verschiedene Sprachen an. Der Grund ist recht einfach: Man möchte den Entwicklern nicht eine vollkommen neue Sprache aufzwingen, sondern ihnen die gewohnte sprachspezifische Syntax lassen.

Wenn Sie nun anmerken sollten, dass es sich bei C# um eine völlig neue Sprache handelt, die mit der Veröffentlichung des .NET Frameworks zur Verfügung gestellt worden ist, haben Sie vollkommen recht. Allerdings assoziiert bereits der Name C# unzweifelhaft, dass die Wurzeln dieser Sprache in C/C++ zu finden sind.

Die Konsequenzen, die sich aus der CLS ergeben, sind weitreichend – nicht für den Endanwender, den es nicht im geringsten interessiert, in welcher Sprache seine Applikation entwickelt wird, sondern vielmehr für ein heterogenes Entwicklerteam in einem Softwareunternehmen. Die Entscheidung, eine Anwendung auf der Grundlage von .NET zu entwickeln, ist keine Entscheidung für oder gegen eine Sprache – es ist eine konzeptionelle Festlegung. Die Bedeutung der einzelnen Sprachen rückt in den Hintergrund, denn die Komponenten, die in einer .NET-konformen Sprache geschrieben sind, können problemlos miteinander interagieren. Eine Klasse, die in C# geschrieben ist, kann von einer Klasse in Visual Basic .NET beerbt werden. Beide Klassen können Daten miteinander austauschen und Ausnahmen weiterreichen. Es gibt unter .NET keine bevorzugte Programmiersprache.

1.2.5 Das Common Type System (CTS)

Jede Entwicklungsumgebung beschreibt als eines ihrer wichtigsten Merkmale ein Typsystem, in dem einerseits Datentypen bereitgestellt werden und andererseits Vorschriften definiert sind, nach denen ein Entwickler die standardmäßigen Typen durch eigene erweitern kann. Darüber hinaus muss eine Regelung getroffen werden, wie auf die Typen zugegriffen wird.

Mit dem *Common Type System* (*CTS*) der .NET-Plattform wird die sprachübergreifende Programmentwicklung spezifiziert und sichergestellt, dass Programmcode unabhängig von der zugrunde liegenden Sprache miteinander interagieren kann. Damit legt das Common Type System die Grundlage für die im vorhergehenden Abschnitt erläuterte Sprachunabhängigkeit.

Alle Typen, die unter .NET zur Verfügung gestellt werden, lassen sich in zwei Kategorien aufteilen:

- Wertetypen
- Referenztypen

Wertetypen werden auf dem Stack abgelegt. Zu ihnen gehören die in der Entwicklungsumgebung eingebauten ganzzahligen Datentypen und die Datentypen, die Fließkommazahlen beschreiben. Referenztypen werden hingegen auf dem Heap abgelegt. Zu ihnen gehören unter anderem die aus den Klassen erzeugten Objekte.

Obwohl Wertetypen im ersten Moment nicht den Anschein erwecken, dass sie von der .NET-Laufzeitumgebung als Objekte behandelt werden, ist dies kein Widerspruch zu der Aussage von vorhin, dass .NET nur Objekte kennt. Tatsächlich erfolgt zur Laufzeit eine automatische

Umwandlung von einen Werte- in einen Referenztyp durch ein Verfahren, das als *Boxing* bezeichnet wird.

Typen können ihrerseits Mitglieder enthalten: Felder, Eigenschaften, Methoden und Ereignisse. Dem Common Type System nur die Festlegung von Typen zuzuschreiben würde die vielfältigen Aufgaben nur vollkommen unzureichend beschreiben. Das CTS gibt zudem die Regeln vor, nach denen die Sichtbarkeit dieser Typmitglieder festgelegt wird. Ein als öffentlich deklariertes Mitglied eines vorgegebenen Typs könnte beispielsweise über die Grenzen der Anwendung hinaus sichtbar sein; andere Sichtbarkeiten beschränken ein Mitglied auf die aktuelle Anwendung oder sogar nur auf den Typ selbst.

Das vom Common Type System festgelegte Regelwerk ist grundsätzlich nichts Neues. Alle anderen Sprachen, auch die, die nicht auf .NET aufsetzen, weisen ein ähnliches Merkmal auf, um ein Typsystem in die Sprache zu integrieren. Aber es gibt einen entscheidenden Unterschied, durch den sich alle Sprachen der .NET-Umgebung vom Rest abheben: Während die Definition des Typsystems bei herkömmlichen Sprachen Bestandteil der Sprache selbst ist, wandert das .NET-Typsystem in die Laufzeitumgebung. Die Folgen sind gravierend: Kommunizieren zwei Komponenten miteinander, die in unterschiedlichen Sprachen entwickelt worden sind, sind keine Typkonvertierungen mehr notwendig, da sie auf demselben Typsystem aufsetzen.

Stellen Sie sich vor, es würde keine Regelung durch das CTS geben und C# würde einen booleschen Typ definieren, der 2 Byte groß ist, während C++ .NET denselben Datentyp definiert, jedoch mit einer Größe von 4 Byte. Der uneingeschränkte Informationsaustausch wäre nicht möglich, sondern würde zu einem Merkmal der Sprache degradiert. Im gleichen Moment würde das ansonsten sehr stabile Framework wie ein Kartenhaus in sich zusammenbrechen – eine fundamentale Stütze wäre ihm entzogen. Dieses Dilemma ist nicht unbekannt und beschert anderen Sprachen große Schwierigkeiten dabei, Funktionen der WinAPI-32 direkt aufzurufen. Ein Beispiel für diese Sprachen ist Visual Basic 6.0.

1.2.6 Das .NET Framework

Ein Framework ist ein Gerüst, mit dem Anwendungen entwickelt, kompiliert und ausgeführt werden. Es setzt sich aus verschiedenen Richtlinien und Komponenten zusammen. Sie haben in Abschnitt 1.2.4 mit der Common Language Specification (CLS) und dem Common Type System (CTS) bereits einen Teil des .NET Frameworks kennengelernt. Wir müssen aber dieses Anwendungsgerüst noch um zwei sehr wichtige Komponenten ergänzen:

- die *Common Language Runtime* (CLR)
- die .NET-Klassenbibliothek

Sie können in manchen Veröffentlichungen noch weitere Komponentenangaben finden, beispielsweise ADO.NET und ASP.NET. Es ist wohl mehr eine Sache der Definition, wo die Grenzen eines Frameworks gesetzt werden, da sich dieser Begriff nicht mit einer klar umris-

senen Definition beschreiben lässt. Die .NET-Klassenbibliothek ihrerseits stellt einen Oberbegriff dar, unter dem sich sowohl ADO.NET als auch ASP.NET eingliedern lassen.

1.2.7 Die Common Language Runtime (CLR)

Die *Common Language Runtime* (*CLR*) ist die Umgebung, in der die .NET-Anwendungen ausgeführt werden – sie ist gewissermaßen die allen gemeinsame Laufzeitschicht. Der Stellenwert dieser Komponente kann nicht hoch genug eingestuft werden, denn mit ihren Fähigkeiten bildet die CLR den Kern von .NET.

Die CLR ist ein Verwalter – auf Englisch *manager*. Tatsächlich wird der Code, der in der Common Language Runtime ausgeführt wird, auch als *verwalteter Code* bezeichnet – oder im Englischen als *managed code*. Umgekehrt kann mit Visual Studio 2019 auch unverwalteter Code geschrieben werden. In *unverwaltetem* oder *unmanaged Code* sind beispielsweise Treiberprogramme geschrieben, die direkt auf die Hardware zugreifen und deshalb plattformabhängig sind.

Sie müssen sich die Common Language Runtime nicht als eine Datei vorstellen, der eine bestimmte Aufgabe im .NET Framework zukommt, wenn verwalteter Code ausgeführt wird. Vielmehr beschreibt die CLR zahlreiche Dienste, die als Bindeglied zwischen dem verwalteten IL-Code und der Hardware den Anforderungen des .NET Frameworks entsprechen und diese sicherstellen. Zu diesen Diensten gehören:

- der *Class Loader*, der Klassen in die Laufzeitumgebung lädt
- der *Type Checker* zur Unterbindung unzulässiger Typkonvertierungen
- der *JITter*, der den MSIL-Code zur Laufzeit in nativen Code übersetzt, der im Prozessor ausgeführt werden kann
- der *Exception Manager*, der die Ausnahmebehandlung unterstützt
- der *Garbage Collector*, der eine automatische Speicherbereinigung anstößt, wenn Objekte nicht mehr benötigt werden
- der *Code Manager*, der die Ausführung des Codes verwaltet
- die *Security Engine*, die sicherstellt, dass der User über die Berechtigung verfügt, den angeforderten Code auszuführen
- die *Debug Machine* zum Debuggen der Anwendung
- der *Thread Service* zur Unterstützung multithreadingfähiger Anwendungen
- der *COM Marshaller* zur Sicherstellung der Kommunikation mit COM-Komponenten (COM = *Component Object Model*)

Die Liste ist zwar lang, vermittelt aber einen Einblick in die verschiedenen unterschiedlichen Aufgabenbereiche der Common Language Runtime.

1.2.8 Die .NET-Klassenbibliothek

Das .NET Framework, das inzwischen in der Version 4.8 vorliegt, ist ausnahmslos objektorientiert ausgerichtet. Für Entwickler, die sich bisher erfolgreich dem objektorientierten Konzept widersetzt und beharrlich auf prozeduralen Code gesetzt haben (solche gibt es häufiger, als Sie vielleicht vermuten), fängt die Zeit des Umdenkens an, denn an der Objektorientierung führt unter .NET kein Weg mehr vorbei.

Alles in .NET Framework wird als Objekt betrachtet. Dazu zählen sogar die nativen Datentypen der *Common Language Specification* wie der Integer. Die Folgen sind weitreichend, denn schon mit einer einfachen Deklaration wie

```
int iVar;
```

erzeugen wir ein Objekt mit allen sich daraus ergebenden Konsequenzen. Wir werden darauf in einem der folgenden Kapitel noch zu sprechen kommen.

Die .NET-Klassen stehen nicht zusammenhangslos im Raum, wie beispielsweise die Funktionen der WinAPI-32, sondern stehen ausnahmslos in einer engen Beziehung zueinander, der .NET-Klassenhierarchie. Eine Klassenhierarchie können Sie sich wie einen Familienstammbaum vorstellen, im dem sich, ausgehend von einer Person, alle Nachkommen abbilden lassen. Auch die .NET-Klassenhierarchie hat einen Ausgangspunkt, gewissermaßen die Wurzel der Hierarchie: Es ist die Klasse `Object`. Jede andere Klasse des .NET Frameworks kann auf sie zurückgeführt werden und erbt daher ihre Methoden. Außerdem kann es weitere Nachfolger geben, die sowohl die Charakteristika der Klasse `Object` erben als auch die ihrer direkten Vorgängerklasse. Auf diese Weise bildet sich eine mehr oder weniger ausgeprägte Baumstruktur.

Für Visual-C++-Programmierer ist eine Klassenhierarchie nichts Neues, sie arbeiten bereits seit vielen Jahren mit den *MFC* (*Microsoft Foundation Classes*). Auch Java-Programmierer haben sich an eine ähnliche Hierarchie gewöhnen müssen.

Eine Klassenhierarchie basiert auf einer Bibliothek, die strukturiert ihre Dienste zum Wohle des Programmierers bereitstellt und letztendlich die Programmierung vereinfacht. Um allerdings in den Genuss der Klassenbibliothek zu kommen, ist ein erhöhter Lernaufwand erforderlich. Wenn man aber aus dieser Phase heraus ist, kann man sehr schnell und zielorientiert Programme entwickeln, die anfänglichen Investitionen zahlen sich also schnell aus.

Einen kurzen Überblick über den Inhalt der .NET-Klassenbibliothek zu geben ist schwer, wenn nicht sogar vollkommen unmöglich, denn es handelt sich dabei um einige Tausend vordefinierte Typen. Wenn man sich jetzt vorstellt, dass in jeder Klasse mehr oder weniger viele Methoden definiert sind, also Funktionen im prozeduralen Sinne, kommt man sehr schnell in Größenordnungen von einigen Zehntausend Methoden, die insgesamt von den Klassen veröffentlicht werden. Alle zu kennen dürfte nicht nur an die Grenze der Unwahrscheinlichkeit stoßen, sondern diese sogar deutlich überschreiten. Außerdem kann man davon ausgehen, dass im Laufe der Zeit immer weitere Klassen mit immer mehr zusätz-

lichen und verfeinerten Features in die Klassenhierarchie integriert werden – sowohl durch Microsoft selbst als auch durch Drittanbieter.

1.2.9 Das Konzept der Namespaces

Da jede Anwendung von Funktionalitäten lebt und der Zugriff auf die Klassenbibliothek zum täglichen Brot eines .NET-Entwicklers gehört, ist ein guter Überblick über die Klassen und insbesondere deren Handling im Programmcode sehr wichtig. Hier kommt uns ein Feature entgegen, das die Arbeit deutlich erleichtert: Es sind die *Namespaces*. Ein Namespace ist eine logische Organisationsstruktur, die völlig unabhängig von der Klassenhierarchie eine Klasse einem bestimmten thematischen Gebiet zuordnet. Damit wird das Auffinden einer Klasse, die bestimmte Leistungsmerkmale aufweist, deutlich einfacher. Das Konzept ist natürlich nicht ganz neu. Ob Java wieder Pate gestanden hat, wissen wir nicht. Aber in Java gibt es eine ähnliche Struktur, die als *Package* bezeichnet wird.

Dass das Auffinden einer bestimmten Klasse erleichtert wird, ist nur ein Argument, das für die Namespaces spricht. Einem zweiten kommt eine ebenfalls nicht zu vernachlässigende Bedeutung zu: Jede Klasse ist durch einen Namen gekennzeichnet, der im Programmcode benutzt wird, um daraus möglicherweise ein Objekt zu erzeugen und auf dessen Funktionalitäten zuzugreifen. Der Name muss natürlich eindeutig sein, schließlich können Sie auch nicht erwarten, dass ein Brief, der nur an »Hans Fischer« adressiert ist, tatsächlich den richtigen Empfänger erreicht. Namespaces verhindern Kollisionen zwischen identischen Klassenbezeichnern, sind also mit der vollständigen Adressierung eines Briefes vergleichbar. Nur innerhalb eines vorgegebenen Namespace muss ein Klassenname eindeutig sein.

Die Namespaces sind auch wieder in einer hierarchischen Struktur organisiert. Machen Sie aber nicht den Fehler, die Klassenhierarchie mit der Hierarchie der Namespaces zu verwechseln. Eine Klassenhierarchie wird durch die Definition der Klasse im Programmcode festgelegt und hat Auswirkungen auf die Fähigkeiten einer Klasse, bestimmte Operationen auszuführen, während die Zuordnung zu einem Namespace keine Konsequenzen für die Fähigkeiten eines Objekts einer Klasse hat. Dass Klassen, die einem bestimmten Namespace zugeordnet sind, auch innerhalb der Klassenhierarchie eng zusammenstehen, ist eine Tatsache, die aus den Zusammenhängen resultiert, ist aber kein Muss.

Wenn die Aussage zutrifft, dass Namespaces in einer baumartigen Struktur organisiert werden, muss es auch eine Wurzel geben. Diese heißt im .NET Framework `System`. Dieser Namespace organisiert die fundamentalsten Klassen in einen Verbund. Weiter oben habe ich erwähnt, dass sogar die nativen Datentypen wie der Integer auf Klassendefinitionen basieren – im Namespace `System` ist diese Klasse neben vielen weiteren zu finden. (Anmerkung: Falls Sie die Klasse jetzt aus Neugier suchen sollten – sie heißt nicht Integer, sondern `Int32`).

Unterhalb von `System` sind die anderen Namespaces angeordnet. Sie sind namentlich so gegliedert, dass man schon erahnen kann, über welche Fähigkeiten die einem Namespace zugeordneten Klassen verfügen. Damit Sie ein Gefühl hierfür bekommen, sind in Tabelle 1.1 auszugsweise ein paar Namespaces angeführt.

Namespace	Beschreibung
`System.Collections`	Klassen, die Auflistungen beschreiben
`System.Data`	Enthält die Klassen für den Zugriff über ADO.NET auf Datenbanken.
`System.Drawing`	Klassen, die grafische Funktionalitäten bereitstellen
`System.IO`	Klassen für Ein- und Ausgabeoperationen
`System.Web`	Enthält Klassen, die im Zusammenhang mit dem Protokoll HTTP stehen.
`System.Windows.Forms`	Enthält Klassen für die Entwicklung Windows-basierter Anwendungen.

Tabelle 1.1 Auszug aus den Namespaces des .NET Frameworks

Die Tabelle gibt kaum mehr als einen Bruchteil aller .NET-Namespaces wieder. Sie sollten allerdings erkennen, wie hilfreich diese Organisationsstruktur bei der Entwicklung einer Anwendung sein kann. Wenn Sie die Lösung zu einem Problem suchen, kanalisieren die Namespaces Ihre Suche und tragen so zu einer effektiveren Entwicklung bei.

Ich kann in diesem Buch natürlich nicht alle Namespaces, geschweige denn alle Klassen des .NET Frameworks behandeln. Ob das überhaupt jemals ein Buch zu leisten vermag, darf mehr als nur angezweifelt werden – zu umfangreich ist die Klassenbibliothek.

Sie sollten die wichtigsten Klassen und Namespaces kennen. Was zu den wichtigsten Komponenten gezählt werden kann, ist naturgemäß subjektiv. Ich werde mich daher auf diejenigen konzentrieren, die praktisch in jeder Anwendung von Belang sind bzw. bei jeder eigenen Klassendefinition in die Überlegung einbezogen werden müssen. In diesem Sinne werde ich mich auf die fundamentalen Bibliotheken beschränken, einschließlich der Bibliotheken, die zur Entwicklung einer Windows-Anwendung notwendig sind.

1.3 Assemblies

Das Ergebnis der Kompilierung von .NET-Quellcode ist eine Assembly. Bei der Kompilierung wird, abhängig davon, welchen Projekttyp Sie gewählt haben, entweder eine EXE- oder eine DLL-Datei erzeugt. Wenn Sie nun in diesen Dateien ein Äquivalent zu den EXE- oder DLL-Dateien sehen, die Sie mit Visual Basic 6.0 oder C/C++ erzeugt haben, liegen Sie falsch – beide sind nicht miteinander vergleichbar.

Assemblies liegen im IL-Code vor. Zur Erinnerung: IL bzw. MSIL ist ein Format, das erst zur Laufzeit einer Anwendung vom JITter in nativen Code kompiliert wird. Eine Assembly kann nicht nur eine, sondern auch mehrere Dateien enthalten – eine Assembly ist daher eher als die Baugruppe einer Anwendung zu verstehen.

Assemblies liegen, wie auch die herkömmlichen ausführbaren Dateien, im *PE*-Format (*Portable Executable*) vor, einem Standardformat für Programmdateien unter Windows. Das Öffnen einer PE-Datei hat zur Folge, dass die Datei der Laufzeitumgebung übergeben und als Folge dessen ausgeführt wird. Daher wird Ihnen beim Starten auch kein Unterschied zwischen einer Assembly und einer herkömmlichen Datei auffallen.

1.3.1 Die Metadaten

Assemblies weisen eine grundsätzlich neue, andersartige Struktur auf. Assemblies enthalten nämlich nicht nur IL-Code, sondern auch sogenannte *Metadaten*. Die Struktur einer kompilierten .NET-Komponente gliedert sich demnach in

- IL-Code und
- Metadaten.

Metadaten sind Daten, die eine Komponente beschreiben. Das hört sich im ersten Moment kompliziert an, ist aber ein ganz triviales Prinzip. Nehmen wir an, Sie hätten die Klasse *Auto* mit den Methoden *Fahren*, *Bremsen* und *Hupen* entwickelt. Wird diese Klasse kompiliert und der IL-Code erzeugt, lässt sich nicht mehr sagen, was der Binärcode enthält, und vor allem, wie er genutzt werden kann. Wenn eine andere Komponente auf die Idee kommt, den kompilierten Code eines *Auto*-Objekts zu nutzen, steht sie vor verschlossenen Türen.

Den Zusammenhang zwischen Metadaten und IL-Code können Sie sich wie das Verhältnis zwischen einem Inhaltverzeichnis und dem Buchtext vorstellen: Man sucht unter einem Stichwort im Inhaltverzeichnis nach einem bestimmten Begriff, findet eine Seitenzahl und kann zielgerichtet im Buch das gewünschte Thema nachlesen. Viel mehr machen die Metadaten eines .NET-Kompilats auch nicht, wenn auch die Funktionsweise naturgemäß etwas abstrakter ist: Sie liefern Objektinformationen, beispielsweise die Eigenschaften eines Objekts und die Methoden. Das geht sogar so weit, dass wir über die Metadaten in Erfahrung bringen, wie die Methoden aufgerufen werden müssen.

Das grundsätzliche Prinzip der Aufteilung in Code und Metadaten ist nicht neu und wurde auch schon unter COM angewandt – allerdings mit einem kleinen, aber doch sehr wesentlichen Unterschied: COM trennt Code und Metadaten. Die Metadaten einer COM-Komponente, die man auch als *Typbibliothek* bezeichnet, werden in die Registry eingetragen und dort ausgewertet. Das ist nicht gut, denn schließlich sollten Sie Ihren Personalausweis immer bei sich tragen und ihn nicht irgendwo hinterlegen. Ebenso sollte auch der Code nicht von seinen Metadaten getrennt werden. COM ist dazu nicht in der Lage; erst innerhalb

des .NET Frameworks wird dieser fundamentalen Forderung nach einer untrennbaren Selbstbeschreibung Rechnung getragen.

Die Metadaten versorgen die .NET-Laufzeitumgebung mit ausreichenden Informationen zum Erstellen von Objekten sowie zum Aufruf von Methoden und Eigenschaften. Sie bilden eine klar definierte Schnittstelle und vereinheitlichen den Objektzugriff, was allen .NET-Entwicklern zugutekommt: Unabhängig von der Sprache – vorausgesetzt, sie ist .NET-konform – können problemlos Objekte verwendet werden, die von anderen Entwicklern bereitgestellt werden. Dass die Objekte in einer beliebigen .NET-Sprache entwickelt sein können, brauche ich fast nicht zu erwähnen.

1.3.2 Das Manifest

Die Folgen der Trennung von Code und Selbstbeschreibung einer COM-Komponente sind uns wahrscheinlich allen bewusst: Durch die Installation einer neuen Anwendung werden alte COM-Komponenten überschrieben, die für andere Anwendungen von existenzieller Bedeutung sind. Die Auswirkungen können fatal sein: Eine Anwendung, die auf die Methoden der überschriebenen Komponente zugreifen will, kann sich im schlimmsten Fall mit einem Laufzeitfehler sang- und klanglos verabschieden.

Mit Assemblierungen gehören diese Fehler definitiv der Vergangenheit an. Verantwortlich dafür sind Metadaten, die nicht die einzelnen Objekte, sondern die Assemblierung als Ganzes beschreiben. Diese Daten werden als *Manifest* bezeichnet. Ein Manifest enthält die folgenden Informationen:

- Name und Versionsnummer der Assembly
- Angaben über andere Assemblierungen, von denen die aktuelle Assembly abhängt
- die von der Assembly veröffentlichten Typen
- Sicherheitsrichtlinien, nach denen der Zugriff auf die Assembly festgelegt wird

Das Manifest befreit eine Assembly von der Notwendigkeit, sich in die Registrierung eintragen zu müssen, und die logischen Konsequenzen gehen sogar noch weiter: Während sich COM-Komponenten erst durch eine Setup-Routine oder zusätzliche Tools in die Registrierungsdatenbank eintragen, können Sie mit den primitivsten Copy-Befehlen eine Assemblierung in ein beliebiges Verzeichnis kopieren – Altbewährtes ist manchmal doch nicht so schlecht.

1.4 Die Entwicklungsumgebung

.NET-Anwendungen lassen sich »notfalls« auch mit MS Editor entwickeln. Natürlich macht das keinen Spaß und ist mühevoll. Auf die Unterstützung, die eine moderne Entwicklungsumgebung bietet, werden Sie vermutlich nicht verzichten wollen. Microsoft bietet mit

Visual Studio 2019 ein Entwicklungstool an, mit dem sich nahezu jede beliebige Anwendung entwickeln lässt.

1.4.1 Editionen von Visual Studio 2019

Es gibt mehrere verschiedene Editionen, die spezifisch auf die unterschiedlichen Anforderungen bei der Anwendungsentwicklung zugeschnitten sind:

- **Visual Studio 2019 Enterprise:** Diese Edition gibt auch großen Entwicklerteams Tools einer effizienten Lebenszyklusverwaltung in die Hand.
- **Visual Studio 2019 Professional:** Diese Edition ist für den professionellen Einsatz kleinerer Entwicklerteams durchaus gut geeignet. Im Vergleich zur Enterprise Edition weist sie einige Einschränkungen auf, wenn Sie die Architektur einer Software beispielsweise validieren oder einige spezielle Testtools einsetzen möchten.
- **Visual Studio 2019 Community Edition:** Die kostenlose Version von Visual Studio hat natürlich nochmals weitere Einschränkungen gegenüber der Professional Edition, ist aber sehr gut zum Lernen und auch für kleinere Projekte geeignet.

Wenn Sie sich für die genauen Details der drei genannten Versionen interessieren, sollten Sie diese im Internet nachlesen oder die drei Editionen austesten. Während die Community unbegrenzt einsetzbar ist, können Sie die Professional und Enterprise Edition ebenfalls downloaden und 30 Tage lang kostenlos testen.

1.4.2 Hard- und Softwareanforderungen

Es verwundert nicht, dass die Systemanforderungen relativ hoch sind:

- Betriebssysteme: Windows 10, Version 1703, Windows Server 2016 oder 2019, Windows Server 2012 R2, Windows 8.1, Windows 7 SP1
- Architekturen: 32 Bit (x86) und 64 Bit (x64)
- Prozessor: 1,8 GHz
- RAM: mind. 2 GB, empfohlen werden 8 GB
- Festplatte: mindestens 800 MB, aber je nach Installation könnte der Bedarf auch maximal 210 GB erreichen. Typischerweise werden um die 50 GB benötigt. Eine SSD ist empfehlenswert.
- Grafikkarte: Auflösung mindestens 1.280 × 720. Je höher die Auflösung, desto besser ist das Arbeiten mit Visual Studio 2019.

Das sind die wichtigsten Eckdaten. Je nach Installation (.NET Core, Xamarin etc.) sind darüber hinaus weitere Anforderungen zu beachten. Sollten Sie sich dafür interessieren, sollten Sie diese auf der Website von Microsoft nachlesen.

1.4.3 Die Installation

Die Installation von Visual Studio 2019 verläuft in der Regel problemlos, unterscheidet sich jedoch von den Installationen bis Visual Studio 2015. Nachdem Sie die EXE-Datei aus dem Internet heruntergeladen haben, wird – so noch nicht auf Ihrem Rechner installiert – der Visual Studio Installer auf Ihrem Rechner installiert. Dieses Tool ist sehr sinnvoll, denn es gestattet, jederzeit Änderungen an einer bestehenden Installation vorzunehmen. Auch Aktualisierungen, die in einem kurzen Rhythmus von Microsoft bereitgestellt werden, werden vom Visual Studio Installer aus gestartet.

In Abbildung 1.3 sehen Sie die Oberfläche des Visual Studio Installers. Sie erkennen, dass sowohl Visual Studio 2017 als auch Visual Studio 2019 bereits installiert sind. Über die Schaltfläche ÄNDERN passen Sie bei Bedarf die aktuelle Installation an: Sollte eine Aktualisierung anstehen, ändert sich die Beschriftung der Schaltfläche in AKTUALISIEREN.

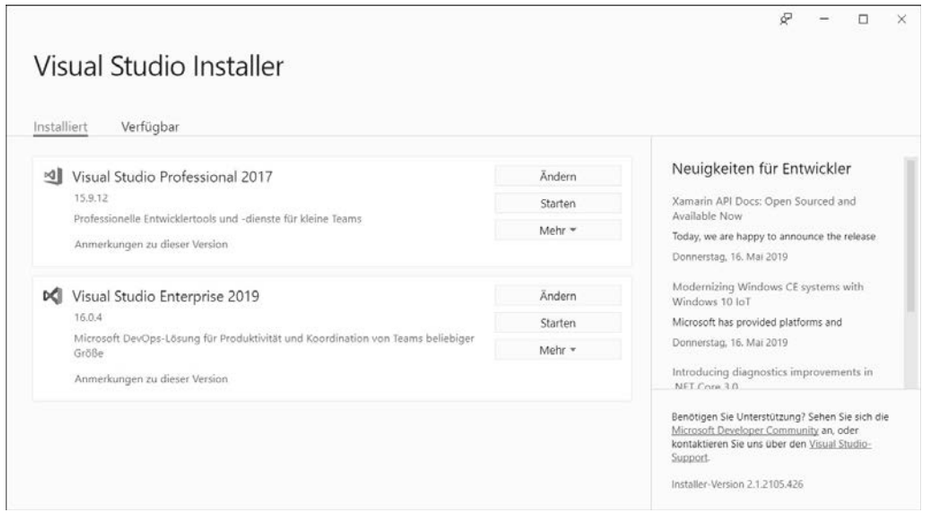

Abbildung 1.3 Die Oberfläche des Visual Studio Installers

Kommen wir zur Installation von Visual Studio selbst. Die Oberfläche, in dem Sie die Auswahl darüber treffen, was Sie installieren möchten, sehen Sie in Abbildung 1.4. Um den Beispielen in diesem Buch zu folgen, sollten Sie im Minimum die Option .NET-DESKTOPENTWICKLUNG auswählen.

Nachdem Sie Ihre Auswahl getroffen haben, klicken Sie bitte noch nicht sofort auf die Schaltfläche rechts unten (in Abbildung 1.4. ist sie mit SCHLIESSEN beschriftet). Achten Sie einmal darauf, dass das Dialogfenster etwas unscheinbare Registerkarten anbietet. In Abbildung 1.4 ist WORKLOADS selektiert. Daneben befinden sich aber noch EINZELNE KOMPONENTEN, SPRACHPAKETE und INSTALLATIONSPFADE. Insbesondere die erstgenannte Registerkarte, EINZELNE KOMPONENTEN, verbirgt noch zahlreiche Zusatzoptionen für die Installation von

Visual Studio 2019 (siehe Abbildung 1.5). Ich empfehle Ihnen, hier in jedem Fall HELP VIEWER auszuwählen. Erst mit dieser Wahl ist es möglich, zumindest Teile der erforderlichen .NET-Dokumentation lokal auf Ihrem Rechner zu installieren und anzusehen. Ansonsten sind Sie auf das Internet angewiesen.

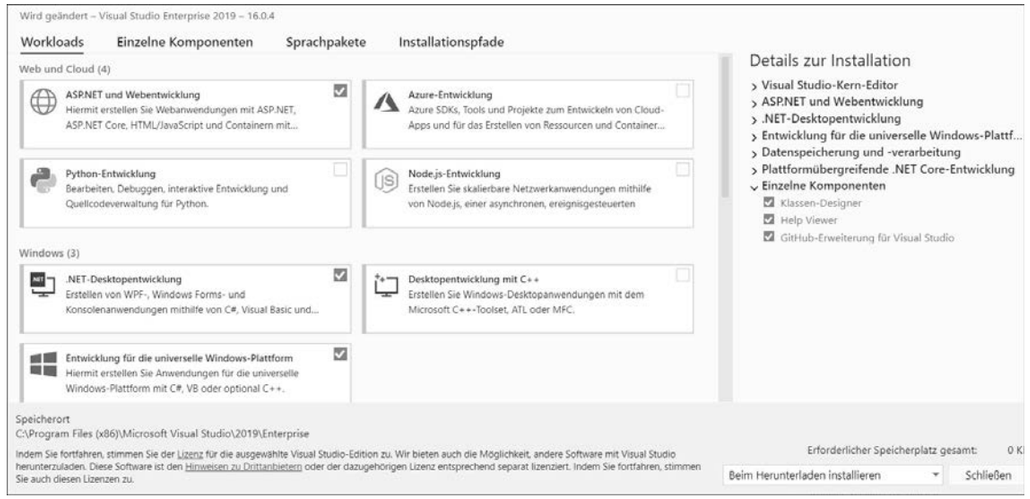

Abbildung 1.4 Auswahldialog der Installationsroutine

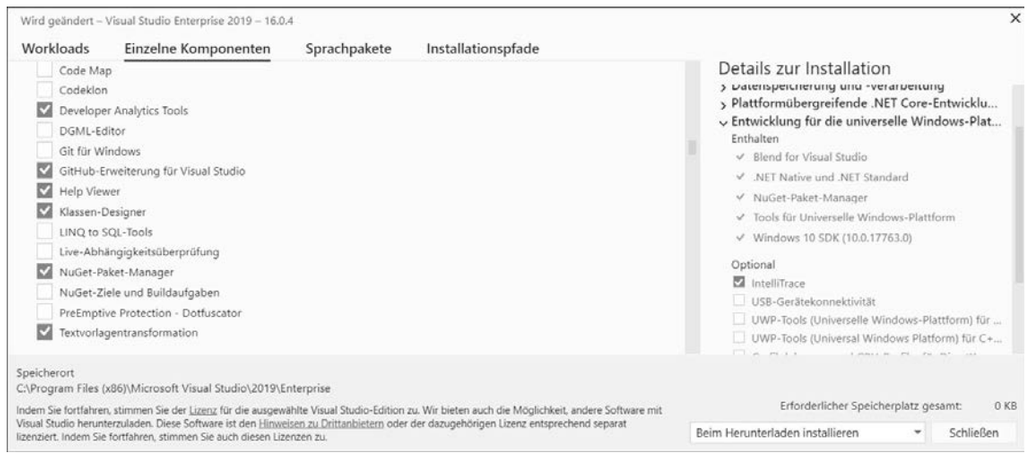

Abbildung 1.5 Weitere spezifische Auswahloptionen

Möchten Sie Visual Studio 2019 mit mehreren Sprachen betreiben, können Sie die gewünschten Sprachen in der Registerkarte SPRACHPAKETE auswählen.

1.4.4 Die Entwicklungsumgebung von Visual Studio 2019

Wie Sie ein neues Projekt im Visual Studio anlegen, unterscheidet sich gravierend von allen älteren Visual Studios. Ich werde Ihnen das in Kapitel 2, »Grundlagen der Sprache C#«, zeigen. An dieser Stelle möchte ich Ihnen nur die wichtigsten Fenster erklären, mit denen Sie es zu tun haben werden:

- der Code-Editor
- der visuelle Editor
- der Projektmappen-Explorer
- das Eigenschaftsfenster
- die Toolbox
- die Fehlerliste

Hier alle Fenster aufzulisten, mit denen Sie während der Entwicklung einer .NET-Anwendung konfrontiert werden, ist nahezu unmöglich. Ich belasse es deshalb bei den genannten, die Sie, mit Ausnahme des Code-Editors, in Abbildung 1.6 wiederfinden. Dabei entspricht die Anordnung ungefähr der, die Sie nach der Installation vorfinden, wenn Sie eine WPF-Anwendung entwickeln wollen.

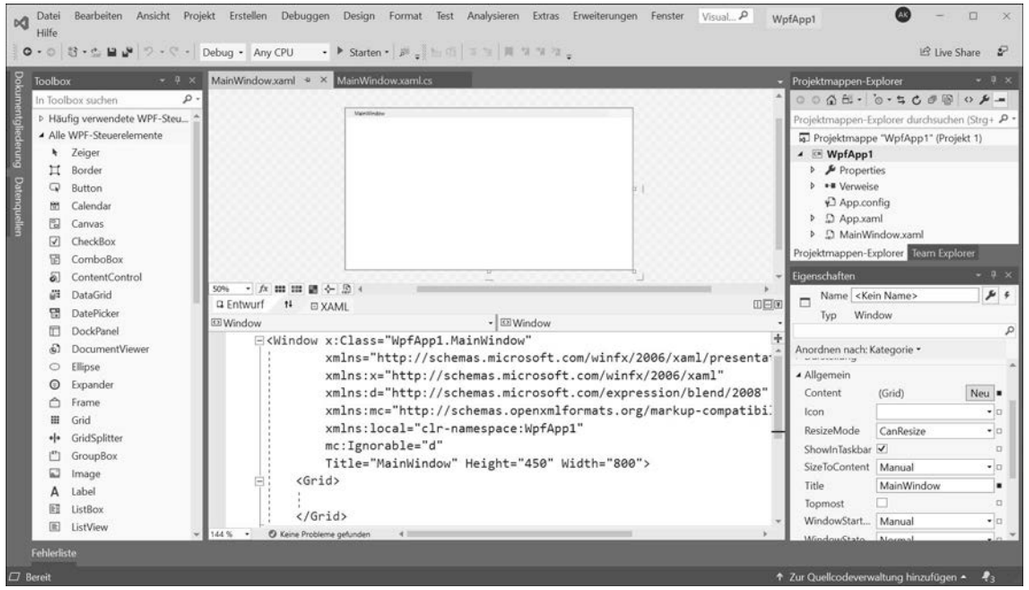

Abbildung 1.6 Die Entwicklungsumgebung eines WPF-Projekts

Nachfolgend möchte ich Ihnen kurz die wichtigsten Fenster von Visual Studio 2019 vorstellen.

Der Code-Editor

Die wichtigste Komponente der Entwicklungsumgebung ist natürlich das Fenster, in dem wir unseren Programmcode schreiben. Abhängig von der gewählten Programmiersprache und der Projektvorlage wird automatisch Code generiert – gewissermaßen als Unterstützung zum Einstieg in das Projekt. Sie können in den meisten Fällen diesen Code nach Belieben ändern – solange Sie wissen, welche mögliche Konsequenz das nach sich zieht.

Insgesamt gesehen ist die Handhabung des *Code-Editors* nicht nur sehr einfach, sondern sie unterstützt den Programmierer durch standardmäßig bereitgestellte Features. Zu diesen zählen:

- automatischer Codeeinzug (Tabulatoreinzug). Die Breite des Einzugs lässt sich auch manuell anders festlegen.
- automatische Generierung von Code, beispielsweise zur Kennzeichnung des Abschlusses eines Anweisungsblocks
- Ein- und Ausblendung der Anweisungsblöcke (Namespaces, Klassen, Prozeduren)
- IntelliSense-Unterstützung
- Darstellung jeder geöffneten Quellcodedatei auf einer eigenen Registerkarte
- eigene Vorder- und Hintergrundfarbe der verschiedenen Elemente

Darüber hinaus lassen sich viele Einstellungen benutzerdefiniert ändern und den eigenen Wünschen anpassen. Dazu öffnen Sie das Menü Extras und wählen hier Optionen...

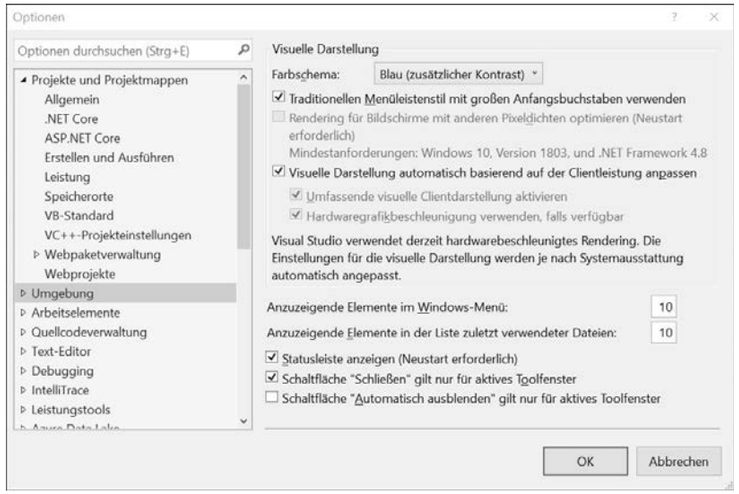

Abbildung 1.7 Der Dialog »Optionen«

Eine Anwendung kann sich aus mehreren Quellcodedateien zusammensetzen. Für jede geöffnete Quellcodedatei wird im Code-Editor eine eigene Registerkarte bereitgestellt. Wird

die Anzahl der angezeigten Registerkarten zu groß, lässt sich jede einzelne über das spezifische Kreuz rechts oben auf der Karte wieder schließen.

Quellcode kann sehr lang und damit insgesamt auch unübersichtlich werden. Mit Hilfe der Zeichen »+« und »–« können Sie Codeblöcke aufklappen und wieder schließen. Ist ein Block geschlossen, wird nur die erste Zeile angezeigt, die mit drei Punkten endet. Insgesamt trägt diese Möglichkeit maßgeblich zu einer erhöhten Übersichtlichkeit des Programmcodes bei.

Per Vorgabe zeigt Visual Studio 2019 nur einen Code-Editor im Zentralbereich an. Oft werden Sie aber das Bedürfnis haben, gleichzeitig den Code von zwei Quellcodedateien einzusehen, und werden nicht mehr zwischen den Registerkarten umschalten wollen. Um das zu erreichen, klicken Sie im Editorbereich mit der rechten Maustaste auf eine beliebige Registerkarte und öffnen damit das Kontextmenü. Sie erhalten dann die Auswahl zwischen Neue horizontale Registerkartengruppe und Neue vertikale Registerkartengruppe.

Der Projektmappen-Explorer

Jede .NET-Anwendung setzt sich aus mehreren Codekomponenten zusammen, und jede .NET-Anwendung kann ihrerseits ein Element einer Gruppe von Einzelprojekten sein, die als *Projektmappe* bezeichnet wird. Der *Projektmappen-Explorer* zeigt die Struktur aller geladenen Projekte an, indem er einerseits die einzelnen Quellcodedateien, die unter Visual C# die Dateiendung *.CS* haben, angibt und andererseits alle Abhängigkeiten eines Projekts (Verweise) mitteilt.

Für uns ist der Projektmappen-Explorer neben der Klassenansicht, die ich im folgenden Abschnitt beschreiben werde, diejenige Komponente der Entwicklungsumgebung, die uns bei der Navigation in unserem Anwendungscode maßgeblich unterstützt: Ein Doppelklick auf eine der aufgelisteten Dateien öffnet im Code-Editor eine Registerkarte, die den Quellcode der Datei enthält. Der Projektmappen-Explorer in Abbildung 1.8 enthält zwei Projekte: *ConsoleApp1* und *ConsoleAppl2*.

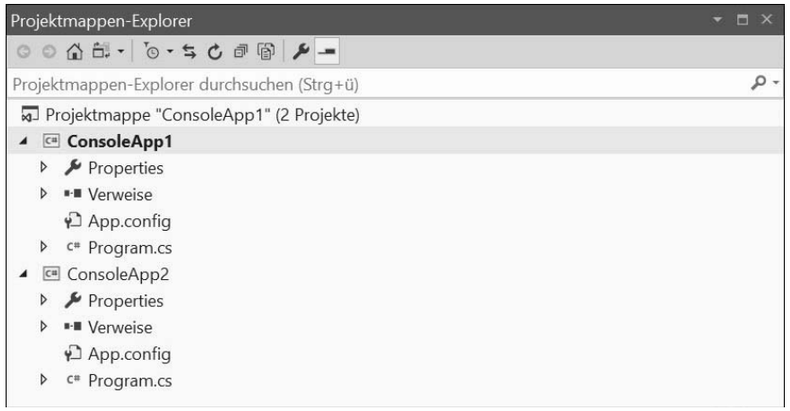

Abbildung 1.8 Der Projektmappen-Explorer

Das Eigenschaftsfenster

Ein Fenster, das sich von Anfang an in der Entwicklungsumgebung einnistet, ist das Fenster *Eigenschaften*. Seine ganze Stärke bei der Anwendungsentwicklung spielt dieses Fenster hauptsächlich dann aus, wenn grafische Oberflächen wie die einer Windows-Anwendung eine Rolle spielen. Sie können hier auf sehr einfache und übersichtliche Art und Weise die Eigenschaften von Schaltflächen, Forms etc. einstellen.

Abbildung 1.9 zeigt den Eigenschaften-Dialog, wenn im Projektmappen-Explorer ein WPF-Window markiert ist. Sie könnten nun beispielsweise die Eigenschaft `Background` ändern, um eine vom Standard abweichende Hintergrundfarbe des Fensters festzulegen. Ändern lassen sich natürlich nur die aktivierten Eigenschaften, die in schwarzer Schriftfarbe erscheinen. Eigenschaften in grauer Schriftfarbe sind schreibgeschützt.

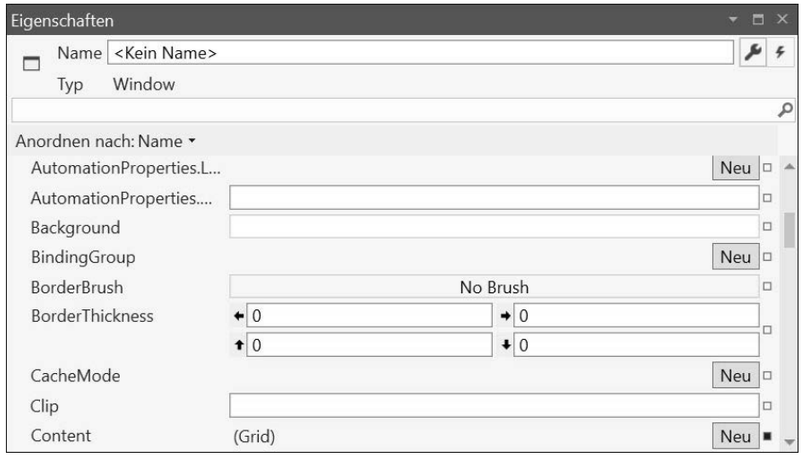

Abbildung 1.9 Das Eigenschaftsfenster

Der Werkzeugkasten (Toolbox)

Die *Toolbox* dient einzig und allein zur Entwicklung grafischer Oberflächen. Sie enthält die Steuerelemente, die mit Visual Studio 2019 ausgeliefert werden, und ist registerkartenähnlich in mehrere Bereiche aufgeteilt. Welche Bereiche angeboten werden, hängt vom Projekttyp ab und variiert daher. Abbildung 1.10 zeigt die Werkzeugsammlung eines WPF-Anwendungsprojekts. Wenn Sie beispielsweise beabsichtigen, das Layout einer Webform zu gestalten, werden in einer Registerkarte nur die Steuerelemente angeboten, die in einer HTML-Seite platziert werden können.

Abbildung 1.10 Die Toolbox

Im zweiten Teil dieses Buches, in dem wir uns der WPF-Programmierung widmen, werden Sie lernen, die meisten der in der Toolbox angebotenen Steuerelemente einzusetzen.

Der Server-Explorer

Die meisten der bisher erwähnten Dialoge der Entwicklungsumgebung dienen der direkten Entwicklungsarbeit. Ich möchte Ihnen aber an dieser Stelle noch einen weiteren Dialog vorstellen, der Sie bei der Anwendungserstellung zumindest indirekt unterstützt: Es ist der *Server-Explorer*. Sie können ihn zur Entwicklungsumgebung von Visual Studio 2019 hinzufügen, indem Sie ihn im Menü ANSICHT auswählen.

Abbildung 1.11 Der Server-Explorer

Die Leistungsfähigkeit des Server-Explorers ist wirklich beeindruckend, denn er integriert den Zugriff auf Dienste und Datenbanken in die Entwicklungsumgebung – und das nicht nur bezogen auf die lokale Maschine, sondern auch auf Systemressourcen, auf die über das Netzwerk zugegriffen werden kann (entsprechende Berechtigungen natürlich vorausgesetzt). Ihnen bleibt es damit erspart, aus Visual Studio heraus immer wieder andere Programme aufzurufen, um an benötigte Informationen zu gelangen.

Kapitel 2
Grundlagen der Sprache C#

2.1 Konsolenanwendungen

2.1.1 Allgemeine Anmerkungen

Nach der Einführung im ersten Kapitel wenden wir uns nun der Programmierung mit C# zu, die sich grundsätzlich in zwei Bereiche unterteilen lässt:

- die fundamentale Sprachsyntax
- die Objektorientierung

Ein tiefgehendes Verständnis beider Ansätze ist Voraussetzung, um eine auf .NET basierende Anwendung entwickeln zu können. Wenn Sie keine Programmierkenntnisse haben, auf die aufgebaut werden kann, ist das gleichzeitige Erlernen beider Teilbereiche schwierig und hindernisreich – ganz abgesehen von den Problemen, die der Umgang mit der komplexen Entwicklungsumgebung aufwirft. Wir werden uns daher in diesem Kapitel zunächst der elementaren Syntax von C# ohne Berücksichtigung der Objektorientierung zuwenden – zumindest weitestgehend, denn ohne den einen oder anderen flüchtigen Blick in die .NET-Klassenbibliothek werden wir nicht auskommen.

Um den Einstieg möglichst einfach zu halten, insbesondere für diejenigen Leser, die sich zum ersten Mal mit der Programmierung beschäftigen, werden wir unsere Programmbeispiele zunächst nur als Konsolenanwendungen entwickeln. Diese sind einerseits überschaubarer als Anwendungen mit visualisiertem UI, andererseits können Sie sich mit der Entwicklungsumgebung schrittweise vertraut machen, ohne durch die vielen Dialogfenster und den automatisch generierten Code sofort den Überblick zu verlieren.

Das Ziel dieses Kapitels ist es, Ihnen die fundamentale Sprachsyntax von C# näherzubringen. Erst danach soll der objektorientierte Ansatz ab Kapitel 3, »Das Klassendesign«, eingehend erläutert werden.

2.1.2 Ein erstes Konsolenprogramm

Sollten Sie bereits mit einer alten Version von Visual Studio gearbeitet haben, werden Sie feststellen, dass sich der Startvorgang nun deutlich geändert hat. Es wird nämlich nicht sofort Visual Studio geöffnet, sondern zuerst ein Dialog wie in Abbildung 2.1 angezeigt.

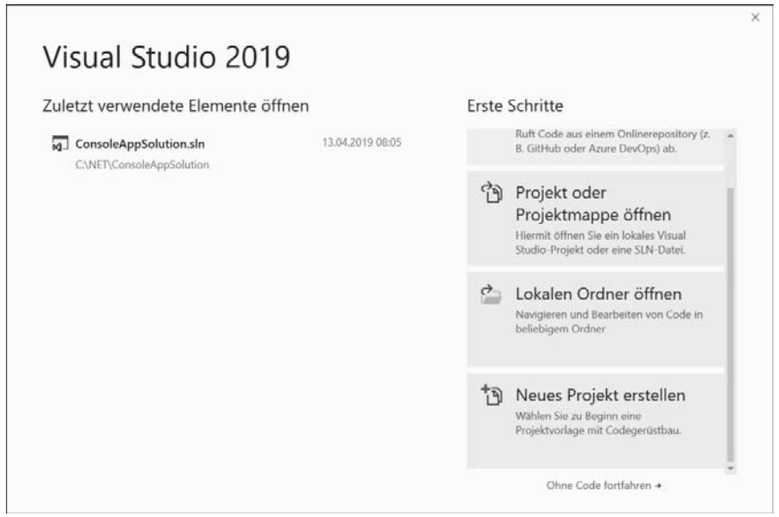

Abbildung 2.1 Der erste Dialog nach dem Start von Visual Studio 2019

In der linken Hälfte des Dialogs werden alle Projekte aufgelistet, die Sie mit Visual Studio erstellt haben. Hier haben Sie per Link die Möglichkeit, sehr schnell ein Projekt zu öffnen, an dem Sie weiterarbeiten möchten. Im rechten Teilbereich werden Ihnen andere Optionen angeboten. Mit dem untersten Eintrag erstellen Sie ein komplett neues Projekt. Haben Sie diese Option gewählt, wird ein weiterer Dialog geöffnet (siehe Abbildung 2.2).

In diesem Dialog müssen Sie die gewünschte Projektvorlage einstellen. Davon gibt es je nachdem, was Sie installiert haben, sehr viele. Um die angebotenen Projektvorlagen einzugrenzen, können Sie die SPRACHE (hier C#), die PLATTFORM (hier Windows) und den PROJEKTTYP einstellen. Da wir uns in den ersten Kapiteln dieses Buches mit den Grundlagen von C# und der objektorientierten Programmierung beschäftigen werden, sollten Sie unter PROJEKTTYP das Angebot KONSOLE auswählen. Möglicherweise werden Ihnen nun mehrere Auswahlmöglichkeiten angezeigt. Achten Sie bitte darauf, dass Sie sich für KONSOLEN-APP (.NET FRAMEWORK) entscheiden und nicht beispielsweise FÜR KONSOLEN-APP (.NET CORE).

Im linken Bereich werden Ihnen alle Projektvorlagen gezeigt, mit denen Sie in der Vergangenheit gearbeitet haben. Oft ist es besser und schneller, hier die Wahl zu treffen.

Der dritte Dialog (siehe Abbildung 2.3) dient dazu, dem neu anzulegenden Projekt einen passenden Bezeichner zu geben und den Speicherort des Projekts festzulegen. Darüber hinaus sollten Sie einen passenden Projektmappenbezeichner wählen. Einer Projektmappe können mehrere Projekte hinzugefügt werden, unabhängig davon, ob sie miteinander in Beziehung stehen oder nicht. Damit ist eine Projektmappe nur ein Verwaltungsorgan von Visual Studio und befreit Sie davon, für jedes Projekt sofort eine neue Instanz von Visual Studio starten zu müssen.

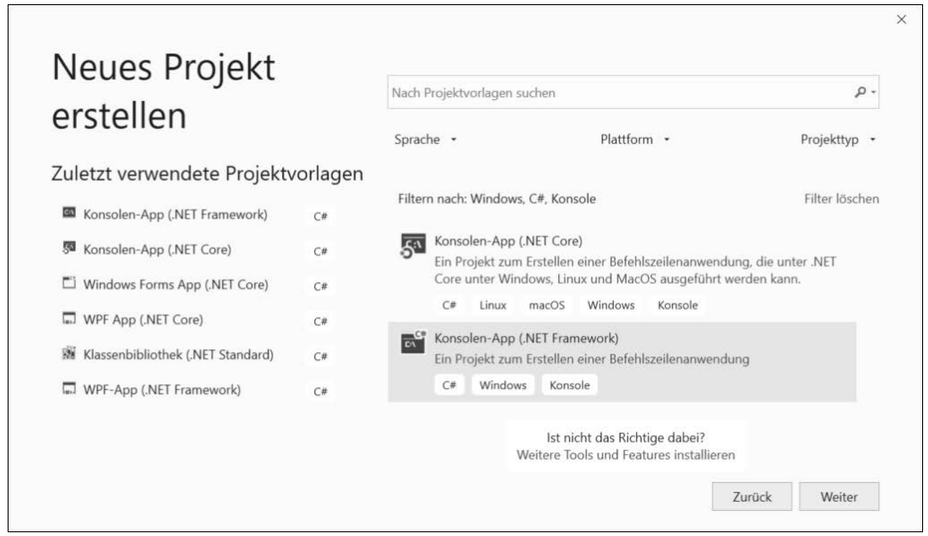

Abbildung 2.2 Auswahl der Projektvorlage

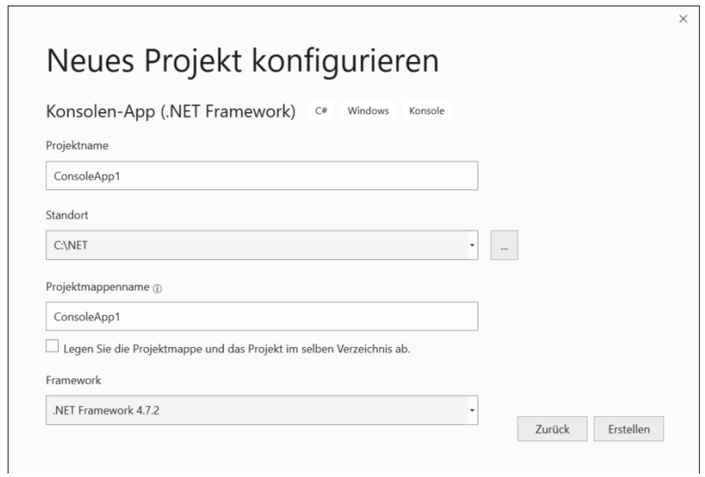

Abbildung 2.3 Anlegen eines neuen Projekts

Nach Betätigen der Taste ERSTELLEN wird im Code-Editor eine Codestruktur angezeigt, die der gewählten Projektvorlage entspricht. Bei einer Konsolenanwendung sieht das so aus:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApp1
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Listing 2.1 Der automatisch generierte Code einer Konsolenanwendung

Im Moment ist es nur wichtig, zu wissen, dass `static void Main` die Methode ist, die beim Starten einer Anwendung als Erstes aufgerufen wird.

Wenn wir das Projekt starten (z. B. über die Schaltfläche STARTEN in der Symbolleiste) öffnet sich kurz das Kommandofenster, das im Moment noch leer ist, in dem wir aber Informationen ausgeben können. Das wollen wir nun in unserer ersten kleinen Anwendung realisieren und uns die Zeichenfolge »C# macht Spaß.« anzeigen lassen. Dazu ergänzen Sie den Programmcode folgendermaßen:

```
static void Main(string[] args)
{
  Console.WriteLine("C# macht Spaß.");
  Console.ReadLine();
}
```

Listing 2.2 Eine erste Ausgabe in der Konsole

Wir haben zwei Zeilen Programmcode eingefügt. Die erste Anweisung dient dazu, eine Ausgabe in die Konsole zu schreiben. Die genaue Syntax werde ich später noch erklären. Würden wir auf die zweite Anweisung verzichten, würde sich das Konsolenfenster zwar öffnen, aber auch sofort wieder schließen, weil das Ende von `Main`, nämlich die schließende geschweifte Klammer, erreicht wird. Wir könnten kaum die Ausgabe der Zeichenfolge lesen. Mit

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

stellen wir also sicher, dass die Konsole so lange geöffnet bleibt, bis der Anwender sie mit der ↵-Taste schließt. Nahezu gleichwertig können Sie auch

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

schreiben. Der Unterschied ist der, dass `ReadKey` auf jede Taste reagiert, `ReadLine` aber nur auf ↵.

Nun wollen wir uns vom Erfolg unserer Aktion natürlich auch noch überzeugen und das Laufzeitverhalten testen. Dazu gibt es mehrere Möglichkeiten:

- Sie klicken in der Symbolleiste auf die Schaltfläche STARTEN.
- Sie wählen im Menü DEBUGGEN das Element DEBUGGEN STARTEN.
- Sie drücken die F5-Taste auf der Tastatur.

Hinweis

Sie können das Projekt aus der Entwicklungsumgebung auch starten, wenn Sie im Menü DEBUGGEN das Untermenü STARTEN OHNE DEBUGGEN wählen. Das hat den Vorteil, dass Sie auf die Anweisung

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

verzichten können. Dafür wird an der Konsole automatisch die Aufforderung `Drücken Sie eine beliebige Taste ...` angezeigt.

Wenn die Ausführung gestartet wird, sollte sich das Konsolenfenster öffnen und wunschgemäß die Zeichenfolge `C# macht Spaß.` anzeigen. Geschlossen wird die Konsole durch Drücken der ↵-Taste. Die Laufzeit wird dann beendet.

Nehmen wir an, Sie hätten einen kleinen Fehler gemacht und vergessen, hinter der Anweisung

```
Console.ReadLine()
```

ein Semikolon anzugeben. Wie Sie später noch erfahren werden, muss jede C#-Anweisung mit einem Semikolon abgeschlossen werden. Nun würde ein syntaktischer Fehler vorliegen, den unser C#-Compiler nicht akzeptiert. Sie bekommen eine Meldung zu dem aufgetretenen Fehler in einem separaten Fenster angezeigt, in der sogenannten *Fehlerliste* (siehe Abbildung 2.4).

Abbildung 2.4 Die Liste mit den Fehlermeldungen

In unserem Beispiel ist nur ein Fehler aufgetreten. Wären es mehrere, würden sie der Reihe nach untereinander angezeigt. Die Beschreibung des Fehlers ist in den meisten Fällen recht informativ – zumindest wenn man etwas Erfahrung im Umgang mit .NET hat. Da sich .NET-Anwendungen in der Regel aus mehreren Dateien zusammensetzen, wird zudem die betroffene Datei genannt und die Zeile, in der der Fehler aufgetreten ist.

Anmerkung

Sollten Sie im Codefenster keine Zeilennummern sehen, können Sie die Anzeige über das Menü Extras • Optionen einstellen. Markieren Sie dazu in der Liste den Eintrag Text-Editor • Alle Sprachen, und setzen Sie das Häkchen vor Zeilennummern.

Doppelklicken Sie auf einen Listeneintrag im Fenster Fehlerliste, springt der Eingabecursor in die Codezeile, die den Fehler verursacht hat. An dieser Stelle sei angemerkt, dass mehrere Fehlereinträge nicht zwangsläufig verschiedene Fehlerursachen haben müssen. Häufig kommt es vor, dass ein einzelner Fehler zu Folgefehlern bei der Kompilierung führt, die alle in der Liste erscheinen. Daher sollten Sie sich in der Fehlerliste immer zuerst dem ersten Eintrag widmen, da seine Beseitigung oft zu einer Reduzierung oder gar kompletten Auflösung der Fehlerliste führt.

2.2 Grundlagen der C#-Syntax

2.2.1 Kennzeichnen, dass eine Anweisung abgeschlossen ist

C#-Programme setzen sich aus vielen Anweisungen zusammen, die der Reihe nach ausgeführt werden. Anweisungen legen fest, was das Programm zu tun hat und auf welche Art und Weise es das tut. Sie haben im vorherigen Abschnitt bereits Ihr erstes, wenn auch sehr kleines Programm mit zwei Anweisungen geschrieben:

```
Console.WriteLine("C# macht Spaß.");
Console.ReadLine();
```

Jede Anweisung verlangt nach einer Kennzeichnung, die das Ende der Anweisung angibt. Dazu wird in C# das Semikolon eingesetzt. Wenn Sie das Semikolon vergessen, erhalten Sie einen Kompilierfehler. Im vorhergehenden Abschnitt hatten wir das sogar provoziert. Auch wenn das sinnlos ist, so dürfen Sie durchaus mehrere Semikolons hintereinanderschreiben, ohne dass explizit eine Anweisung dazwischen stehen muss.

Weil durch ein Semikolon eine Anweisung eindeutig abgeschlossen wird, dürfen auch mehrere Anweisungen in eine Zeile geschrieben werden. Im Umkehrschluss kann eine Anweisung auch problemlos auf mehrere Zeilen verteilt werden, ohne dass es den Compiler stört.

Bei der Gestaltung des Programmcodes lässt C# Ihnen sehr viele Freiheiten. Leerzeichen, Tabulatoren und Zeilenumbrüche können nach Belieben eingestreut werden, ohne dass sich das auf die Kompilierung des Quellcodes oder die Ausführung des Programms auswirkt. Daher dürfte der Code unseres ersten Beispiels oben auch wie folgt aussehen:

```
Console.
      WriteLine("C# macht Spaß.")    ;
```



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

Listing 2.3 »Streuung« des Programmcodes

Dass eine Streuung wie die gezeigte die gute Lesbarkeit des Codes beeinträchtigt, steht außer Frage. Aber der Compiler führt diesen Code dennoch korrekt aus.

Mit Einrückungen tragen Sie zu einer guten Lesbarkeit des Programmcodes bei. Sehen Sie sich dazu das Listing 2.2 an. Anweisungen, die innerhalb von geschweiften Klammern stehen, werden üblicherweise nach rechts eingerückt. Wenn Sie sich an den Beispielen in diesem Buch orientieren, werden Sie sehr schnell ein Gefühl dafür bekommen, wie Sie mit Einrückungen leichter lesbaren Code schreiben. Feste Regeln gibt es dazu allerdings nicht, es sind stillschweigende Konventionen.

2.2.2 Anweisungs- und Gliederungsblöcke

C#-Programmcode ist blockorientiert, das heißt, dass C#-Anweisungen grundsätzlich immer innerhalb eines Paars geschweifter Klammern geschrieben werden. Jeder Block kann eine beliebige Anzahl von Anweisungen enthalten – oder auch keine. Somit hat ein Anweisungsblock allgemein die folgende Form:

```
{
   Anweisung 1;
   Anweisung 2;
   [...]
}
```

Listing 2.4 Einfacher Anweisungsblock

Anweisungsblöcke lassen sich beliebig ineinander verschachteln. Dabei beschreibt jeder Anweisungsblock eine ihm eigene Ebene, z. B.:

```
{
   Anweisung 1;
   {
      Anweisung 2;
      Anweisung 3;
   }
   Anweisung 4;
}
```

Listing 2.5 Verschachtelte Anweisungsblöcke

Beachten Sie, wie Einzüge hier dazu benutzt werden, optisch die Zugehörigkeit einer oder mehrerer Anweisungen zu einem bestimmten Block deutlich zu machen. Die Anweisungen 2 und 3 sind zu einem Block zusammengefasst, der sich innerhalb eines äußeren Blocks befindet. Zum äußeren Anweisungsblock gehören Anweisung 1 und Anweisung 4 sowie natürlich auch der komplette innere Anweisungsblock.

2.2.3 Kommentare

Sie sollten nicht mit Kommentaren geizen. Kommentare helfen, den Programmcode der Anwendung besser zu verstehen. C# bietet zwei Möglichkeiten, Kommentare, die vom Compiler während des Kompiliervorgangs ignoriert werden, in den Quellcode einzustreuen. Die am häufigsten benutzte Variante ist die Einleitung eines Kommentars mit zwei Schrägstrichen:

```
// dies ist ein Kommentar
```

Ein //-Kommentar gilt für den Rest der gesamten Codezeile, kann jedes beliebige Zeichen enthalten und darf auch nach einer abgeschlossenen Anweisung stehen.

```
Console.WriteLine("..."); //Konsolenausgabe
```

Sollen viele zusammenhängende Zeilen zu einem längeren Kommentar zusammengefasst werden, bietet sich die zweite Alternative an, bei der ein Kommentar mit /* eingeleitet und mit */ abgeschlossen wird. Alle Zeichen, die sich dazwischen befinden, sind Bestandteil des Kommentars.

```
/* Console.WriteLine("...");
Console.ReadLine();*/
```

Tatsächlich kann man diesen Kommentar sogar mitten in einer Anweisung schreiben, ohne dass der C#-Compiler das als Fehler ansieht:

```
Console.WriteLine /* Kommentar */("...");
```

Die Entwicklungsumgebung von Visual Studio bietet eine recht interessante und einfache Alternative, insbesondere größere Blöcke auf einen Schlag auszukommentieren. Sie müssen dazu nur sicherstellen, dass in der Entwicklungsumgebung die Symbolleiste TEXT-EDITOR angezeigt wird. Dazu brauchen Sie nur mit der rechten Maustaste das Kontextmenü einer der aktuellen Symbolleisten zu öffnen. Im Kontextmenü finden Sie alle Symbolleisten der Entwicklungsumgebung aufgelistet. Da die Anzahl als nicht gering bezeichnet werden kann, lassen sich die einzelnen Symbolleisten nach Bedarf ein- oder ausblenden.

Die Symbolleiste *Text-Editor* enthält zwei Schaltflächen, um markierte Codeblöcke auszukommentieren ❶ oder eine Kommentierung wieder aufzuheben ❷ (siehe Abbildung 2.5).

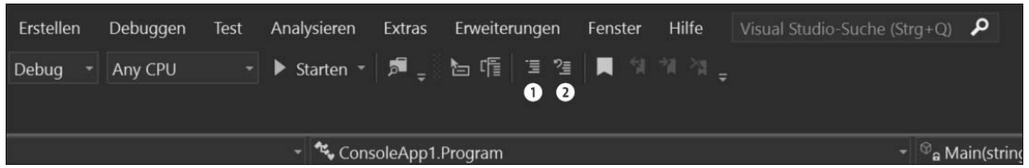

Abbildung 2.5 Kommentare mit Hilfe der Symbolleiste

> **Anmerkung**
>
> In diesem Buch werden alle Kommentare grau hinterlegt, um auf einen Blick deutlich zwischen Code und Kommentar unterscheiden zu können. Zudem bezieht sich in diesem Buch jeder Kommentar auf die darunterliegende Anweisung.

2.2.4 Die Groß- und Kleinschreibung

C# gehört zu der Gruppe von Programmiersprachen, die zwischen Groß- und Kleinschreibung unterscheiden. Falls Sie die Anweisung zur Konsolenausgabe mit

```
Console.Writeline("...");
```

codieren, werden Sie bei der Kompilierung mit einer Fehlermeldung konfrontiert. Korrekt müsste es lauten:

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

Die Folge ist, dass zwei gleich lautende Bezeichner, die sich nur durch Groß- und Kleinschreibung unterscheiden, in C# auch für zwei unterschiedliche Programmelemente stehen.

2.2.5 Die Struktur einer Konsolenanwendung

Sehen Sie sich noch einmal Listing 2.1 an. Dabei handelt es sich um den Code, den uns die Entwicklungsumgebung nach dem Öffnen eines neuen Projekts erzeugt.

Wir erkennen nun mehrere verschachtelte Blockstrukturen. Der äußere Block definiert einen Namespace mit dem Namen `ConsoleApp1`. Namespaces dienen zur Sicherstellung der Eindeutigkeit eines Bezeichners (hier der Klasse `Program`). Wir kommen auf die Details der Namespaces am Ende von Kapitel 3, »Das Klassendesign«, noch einmal zu sprechen. Im Namespace eingebettet ist eine Klassendefinition, die einen eigenen Anweisungsblock beschreibt:

```
namespace ConsoleApp1
{
  class Program
  {
  }
}
```

Listing 2.6 Im Namespace eingebettete Klassendefinition

C# ist eine 100%ig objektorientierte Sprache. Das bedeutet, dass grundsätzlich immer eine Klassendefinition vorliegen muss, um mit einem Objekt arbeiten zu können.

Eine Klasse beschreibt einen Typ, und in unserem Fall heißt dieser Typ `Program`. Der Bezeichner `Program` ist nur als Vorschlag der Entwicklungsumgebung anzusehen und darf innerhalb des aktuellen Namespaces (hier also `ConsoleApp1`) frei vergeben werden, solange die Eindeutigkeit innerhalb des Namespace gewährleistet ist.

Wie Sie sehen, kommen wir schon an dieser Stelle zum ersten Mal mit Klassen in Berührung. Was eine Klasse darstellt und wie man sie einsetzt, beschreibe ich in Kapitel 3, »Das Klassendesign«. Interessanter ist für uns momentan der Anweisungsblock, der innerhalb der Klasse `Program` vordefiniert ist. In ihm wird die Methode `Main` beschrieben:

```
static void Main(string[] args)
{
}
```

Bei `Main` handelt es sich um eine Methode, die für uns von herausragender Bedeutung ist: Wenn wir die Laufzeitumgebung einer eigenstartfähigen Anwendung (EXE-Datei) starten, wird zuerst `Main` ausgeführt. Sie dürfen den Namen von `Main` auch nicht ändern und müssen natürlich auch die Großschreibweise berücksichtigen, denn beim Start der Laufzeitumgebung wird immer nach `Main` gesucht und nicht nach `main` oder nach `start`.

Weiter oben habe ich gesagt, dass Anweisungen immer innerhalb eines Blocks aus geschweiften Klammern codiert werden. Wir können diese Aussage nun präzisieren: Anweisungen werden grundsätzlich immer innerhalb des Anweisungsblocks einer Methode implementiert (siehe Listing 2.2).

Sehen wir uns noch kurz die Definition der `Main`-Methode an. Die beiden dem Bezeichner vorausgehenden Schlüsselwörter `static` und `void` sind zwingend notwendig. Sollten Sie bereits mit C++ oder Java Anwendungen entwickelt haben, werden Sie die Bedeutung dieser beiden Modifizierer kennen: Mit `static` werden Methoden bezeichnet, die beim Aufruf kein konkretes Objekt voraussetzen, und `void` beschreibt eine Methode ohne Rückgabewert. Im Moment soll diese Information genügen, denn eine genauere Kenntnis hätte derzeit keine Auswirkungen auf die ersten Schritte in die Welt der C#-Programme.

Ein Konsolenprogramm wird gestartet, indem Sie den Namen der Anwendungsdatei an der Konsole eingeben. Manchmal ist es notwendig, dem Programm beim Start Anfangswerte mitzuteilen, die vom laufenden Programm zur Ausführung und Weiterverarbeitung benötigt werden. Angenommen, Sie wollen einer Konsolenanwendung mit dem Dateinamen *MyApplication.exe* die drei Zahlen 10, 14 und 17 übergeben, sähe der Aufruf an der Eingabeaufforderung wie folgt aus:

```
MyApplication.exe 10 14 17
```

Diese drei Zahlen werden vom Parameter `args`, der hinter dem Bezeichner `Main` in runden Klammern angegeben ist, in Empfang genommen.

Wie die übergebenen Daten im Programmcode verarbeitet werden können, folgt später. Die Angabe der Parameterliste ist optional. Benötigt ein Programm bei seinem Aufruf keine Daten, kann die Parameterliste leer bleiben. Die Angabe der runden Klammern ist aber unbedingt erforderlich.

2.3 Variablen und Datentypen

Dateninformationen bilden die Grundlage der Datenverarbeitung und hauchen einem Programm Leben ein: Daten können anwendungsspezifisch sein, den Zustand von Objekten beschreiben, Informationen aus Datenbanken repräsentieren oder auch nur eine Netzwerkadresse. Daten bilden also gemeinhin die Basis der Gesamtfunktionalität einer Anwendung.

2.3.1 Variablendeklaration

Praktisch jedes Programm benötigt Daten, um bestimmte Aufgaben zu erfüllen. Daten werden in Variablen vorgehalten. Dabei steht eine Variable für eine Adresse im Hauptspeicher des Rechners. Ausgehend von dieser Adresse wird eine bestimmte Anzahl von Bytes reserviert – entsprechend dem Typ der Variablen. Das, was eine Variable repräsentiert, kann vielfältiger Art sein: eine einfache Ganzzahl, eine Fließkommazahl, ein einzelnes Zeichen, eine Zeichenkette, eine Datums- oder Zeitangabe, aber auch die Referenz auf die Startadresse eines Objekts.

Der Bezeichner einer Variablen dient dazu, die Speicheradresse des Werts im Programmcode mit einem Namen anzusprechen, der sich einfach merken lässt. Er ist also vom Wesen her nichts anderes als ein Synonym oder Platzhalter eines bestimmten Speicherorts.

Variablen müssen deklariert werden. Unter einer *Deklaration* wird die Bekanntgabe des Namens der Variablen sowie des von ihr repräsentierten Datentyps verstanden. Die Deklaration muss vor der ersten Wertzuweisung an die Variable erfolgen. Dabei wird zuerst der

Datentyp angegeben, dahinter der Variablenname. Abgeschlossen wird die Deklaration mit einem Semikolon. Beispielsweise könnte eine zulässige Deklaration wie folgt aussehen:

```
int value;
```

Damit wird dem Compiler mitgeteilt, dass der Bezeichner `value` für einen Wert steht, der vom Typ einer Ganzzahl, genauer gesagt vom Typ `int` (Integer) ist. Mit

```
value = 1000;
```

wird dieser Variablen ein gültiger Wert zugewiesen. Man spricht dann auch von der *Initialisierung* der Variablen.

Wenn Sie versuchen, auf eine nicht deklarierte Variable zuzugreifen, wird der C#-Compiler einen Fehler melden. Ebenso falsch ist es, den Inhalt einer nicht initialisierten Variablen auswerten zu wollen. Deklaration und Initialisierung können auch in einer einzigen Anweisung erfolgen:

```
int value = 0;
```

Auf diese Weise vermeiden Sie eine nicht initialisierte Variable. Müssen Sie mehrere Variablen vom gleichen Typ deklarieren, können Sie die Bezeichner, getrennt durch ein Komma, hintereinander angeben:

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

Sie können dann auch eine oder mehrere Variablen sofort initialisieren:

```
int a, b = 9, c = 12;
```

2.3.2 Der Variablenbezeichner

Ein Variablenname unterliegt besonderen Reglementierungen:

- Ein Bezeichner darf sich nur aus alphanumerischen Zeichen und dem Unterstrich zusammensetzen. Leerzeichen und andere Sonderzeichen wie beispielsweise #, § und $ sind nicht zugelassen.
- Ein Bezeichner muss mit einem Buchstaben oder dem Unterstrich anfangen.
- Ein einzelner Unterstrich als Variablenname ist nicht zulässig.
- Der Bezeichner muss eindeutig sein. Er darf nicht gleichlautend mit einem Schlüsselwort, einer Methode, einer Klasse oder einem Objektnamen sein.

Noch ein Hinweis zur Namensvergabe: Wählen Sie grundsätzlich beschreibende Namen, damit Ihr Code später besser lesbar wird. Einfache Bezeichner wie `x` oder `y` usw. sind wenig aussagekräftig. Besser wäre eine Wahl wie `color`, `salary` oder `firstName`. Nur den Zählervariablen von Schleifen werden meistens Kurznamen gegeben.

Hinweis

Die hier exemplarisch angegebenen Variablenbezeichner fangen alle mit einem Kleinbuchstaben an. Folgen Sie der allgemeinen .NET-Namenskonvention, sollten alle Variablen, die innerhalb einer Methode definiert sind, mit einem Kleinbuchstaben beginnen. Man spricht bei diesen Variablen auch von *lokalen Variablen*. Alle weiteren Fälle jetzt aufzuführen, würde den Rahmen momentan sprengen. Es sei aber angemerkt, dass in diesem Buch die Variablenbezeichner fast durchweg der .NET-Namenskonvention folgen.

2.3.3 Der Zugriff auf eine Variable

Wir wollen uns jetzt noch ansehen, wie wir uns den Inhalt einer Variablen an der Konsole ausgeben lassen können. Wir deklarieren dazu eine Variable vom Typ `long` und weisen ihr einen Wert zu, den wir danach an der Konsole ausgeben lassen.

```
static void Main(string[] args)
{
  long value = 4711;
  Console.WriteLine("value = {0}", value);
  Console.ReadLine();
}
```

Listing 2.7 Variablen innerhalb der Methode »Main«

Die Ausgabe im Befehlsfenster wird wie folgt lauten:

```
value = 4711
```

In diesem Listing wird eine andere Variante der Konsolenausgabe erstmalig benutzt:

```
Console.WriteLine("value = {0}",value);
```

In diesem Listing wird eine andere Variante der Konsolenausgabe erstmalig benutzt:

```
Console.WriteLine("value = {0}",value);
```

Sie haben bereits gesehen, dass mit `Console.WriteLine` eine einfache Konsolenausgabe codiert wird. `WriteLine` ist eine Methode, die in der Klasse `Console` definiert ist. Jetzt fehlt noch die genaue Erklärung der verwendeten Syntax zur Ausgabe der Daten.

2.3.4 Ein- und Ausgabemethoden der Klasse »Console«

Es bleibt uns nichts anderes übrig, als an dieser Stelle einen kleinen Ausflug in die Welt der Klassen und Objekte zu unternehmen, weil wir immer wieder mit den Methoden verschiedener Klassen arbeiten werden. Es handelt sich dabei meist um Methoden, die an der Eingabe-

konsole Ein- und Ausgabeoperationen durchführen: `Write` und `WriteLine` sowie `Read` und `ReadLine`.

Die Methoden »WriteLine«, »ReadLine«, »Write« und »Read«

Die Klasse `Console` ermöglicht es, über die beiden Methoden `Write` und `WriteLine` auf die Standardausgabeschnittstelle zuzugreifen. Der Begriff »Ausgabeschnittstelle« mag im ersten Moment ein wenig verwirren, aber tatsächlich wird darunter die Anzeige an der Konsole verstanden. `WriteLine` und `Write` unterscheiden sich dahingehend, dass die erstgenannte Methode dem Ausgabestring automatisch einen Zeilenumbruch anhängt und den Cursor in die folgende Ausgabezeile setzt. Nach dem Aufruf der Methode `Write` verbleibt der Eingabecursor in der aktuellen Ausgabezeile. Beide Methoden sind aber ansonsten identisch.

Grundsätzlich gilt: Wollen wir die Methode eines Objekts oder einer Klasse aufrufen, geben wir den Objekt- bzw. Klassennamen an und von diesem durch einen Punkt getrennt den Namen der Methode. Man spricht hierbei auch von der sogenannten *Punktnotation*. An den Methodennamen schließt sich ein Klammerpaar an. Allgemein lautet die Syntax also:

```
Objektname.Methodenname();
```

Sie können sich mit dieser Syntax durchaus schon vertraut machen, denn sie wird Ihnen ab sofort überall begegnen, da sie in objektorientiertem Programmcode elementar ist.

Das runde Klammerpaar hinter der `Read`- bzw. `ReadLine`-Methode bleibt immer leer. Bei den Methoden `Write` und `WriteLine` werden innerhalb der Klammern die auszugebenden Daten einschließlich ihres Ausgabeformats beschrieben. Allerdings dürfen auch bei den beiden letztgenannten Methoden die Klammern leer bleiben. Im einfachsten Fall wird einer der beiden Ausgabemethoden eine Zeichenfolge in Anführungsstrichen übergeben:

```
Console.WriteLine("C# macht Spaß.");
```

Formatausdrücke in den Methoden »Write« und »WriteLine«

Damit sind die Möglichkeiten der `Write`/`WriteLine`-Methoden noch lange nicht erschöpft. Die flexiblen Formatierungsmöglichkeiten erlauben die Ausgabe von Daten an beliebigen Positionen innerhalb der Ausgabezeichenfolge. Dazu dient ein Platzhalter, der auch als *Formatausdruck* bezeichnet wird. Er ist an den geschweiften Klammern zu erkennen und enthält zumindest eine Zahl. Hinter der auszugebenden Zeichenfolge werden, durch ein Komma getrennt, die Informationen übergeben, was anstelle des Formatausdrucks auszugeben ist. Sehen wir uns dazu ein Beispiel an:

```
string text1 = "C#";
string text2 = "Spass";
Console.Write("{0} macht {1}.", text1, text2);
```

Listing 2.8 Formatausdruck in der Methode »Console.WriteLine«

Hier sind die beiden Variablen `text1` und `text2` vom Typ `string` deklariert, die mit einer in Anführungsstrichen gesetzten Zeichenfolge initialisiert werden.

Die auszugebende Zeichenfolge wird in Anführungsstriche gesetzt. Getrennt durch Kommata werden dahinter die beiden Variablen `text1` und `text2` bekannt gegeben. Der Inhalt der zuerst genannten Variablen `text1` ersetzt den Formatausdruck `{0}` innerhalb der Ausgabezeichenfolge, die zweite Variable `text2` ersetzt den Formatausdruck `{1}`. Entscheidend ist, dass dem ersten Parameter (`text1`) die Zahl 0 zugeordnet wird, dem zweiten (`text2`) die Zahl 1 usw. Die Konsolenausgabe lautet:

```
C# macht Spaß.
```

Innerhalb des Ausgabestrings müssen die anzuzeigenden Listenelemente nicht der Reihenfolge nach durchlaufen werden. Man kann sie beliebig ansprechen oder sogar einfach ungenutzt lassen. Die Anweisung

```
Console.Write("{1} macht {0}.", text1, text2);
```

würde demnach zu der folgenden Ausgabe führen:

```
Spaß macht C#.
```

Der Formatausdruck dient nicht nur der eindeutigen Bestimmung des Elements, er ermöglicht auch eine weitergehende Einflussnahme auf die Ausgabe. Soll der einzusetzende Wert eine bestimmte Breite einnehmen, gilt die syntaktische Variante:

```
{N, M}
```

Dabei gilt Folgendes:

- `N` ist ein nullbasierter Zähler.
- `M` gibt die Breite der Ausgabe an.

Unbesetzte Plätze werden durch eine entsprechende Anzahl von Leerzeichen aufgefüllt. Sehen wir uns dazu ein Codefragment an:

```
int value = 10;
Console.WriteLine("Ich kaufe {0,3} Eier", value);
Console.WriteLine("Ich kaufe {0,10} Eier", value);
```

Listing 2.9 Erweiterte Formatierungsmöglichkeiten

Die Ausgabe von Listing 2.9 lautet:

```
Ich kaufe  10 Eier
Ich kaufe         10 Eier
```

Die erste Ausgabe hat eine Gesamtbreite von drei Zeichen, die Zahl selbst ist allerdings nur zwei Ziffern breit. Daher wird vor der Zahl ein Leerzeichen gesetzt. Da für die Breite der zwei-

ten Ausgabe zehn Zeichen vorgeschrieben sind, werden links von der Zahl acht Leerstellen eingefügt.

Die Breitenangabe darf auch negativ sein. Die Ausgabe erfolgt dann linksbündig, daran schließen sich die Leerstellen an.

Sie können den Formatausdruck so spezifizieren, dass numerische Ausgabedaten eine bestimmte Formatierung annehmen. Das führt uns zu der vollständigen Syntax des Formatausdrucks:

```
// Syntax des Formatausdrucks
{N [,M][:Format]}
```

`Format` spezifiziert, wie die Daten angezeigt werden. In Tabelle 2.1 werden die möglichen Optionen aufgelistet.

Formatangabe	Beschreibung
C	Zeigt die Zahl im lokalen Währungsformat an.
D	Zeigt die Zahl als dezimalen Integer an.
E	Zeigt die Zahl im wissenschaftlichen Format an (Exponentialschreibweise).
F	Zeigt die Zahl im Festpunktformat an.
G	Eine numerische Zahl wird entweder im Festpunkt- oder im wissenschaftlichen Format angezeigt. Zur Anzeige kommt das »kompakteste« Format.
N	Zeigt eine numerische Zahl einschließlich Kommaseparatoren an.
P	Zeigt die numerische Zahl als Prozentzahl an.
X	Die Anzeige erfolgt in Hexadezimalnotation.

Tabelle 2.1 Formatangaben der Formatausgabe

An alle Formatangaben kann eine Zahl angehängt werden, aus der die Anzahl der signifikanten Stellen hervorgeht. Nachfolgend sollen einige Beispiele den Einsatz der Formatangaben demonstrieren:

```
int value = 4711;
// Ausgabe: value=4,711000E+03
Console.WriteLine("value={0:E}", value);
// Ausgabe: value=4,71E+003
Console.WriteLine("value={0:E2}", value);
float value1 = 0.2512F;
// Ausgabe: value1=     0,2512
```



```
Console.WriteLine("value1={0,10:G}", value1);
// Ausgabe: value1=25,1200%
Console.WriteLine("value1={0:P4}", value1);
```

Listing 2.10 Verschieden formatierte Ausgaben

Stringinterpolation

Formatausdrücke haben das Ziel, durch Vermeidung von Stringverkettungen eine Zeichenfolge besser lesbar zu machen. Andererseits können viele Formatausdrücke dazu führen, dass die Zuordnung eines Formatausdrucks zu seinem Wert mühsam wird. Hier hilft ein Feature weiter, das in C# 6.0 eingeführt wurde: die Stringinterpolation. Die String-Interpolation erlaubt die Angabe der Variablen direkt innerhalb der geschweiften Klammern des Formatausdrucks. Dazu ein kleines Beispiel:

```
int a = 67;
int b = 771;
Console.WriteLine($"a = {a}, b = {b}");
```

Listing 2.11 String-Interpolation

Beachten Sie, dass vor der Zeichenfolge das Zeichen `$` gesetzt werden muss.

Escape-Zeichen

Ähnlich wie andere Hochsprachen stellt C# eine Reihe von Escape-Sequenzen zur Verfügung, die dann verwendet werden, wenn Sonderzeichen innerhalb einer Zeichenfolge ausgegeben werden sollen. Beispielsweise erzwingen Sie mit dem Zeichen `\n` einen Zeilenumbruch:

```
Console.Write("C#\nmacht\nSpaß.");
```

An der Konsole wird dann

```
C#
macht
Spaß.
```

angezeigt.

Escape-Zeichen	Beschreibung
\'	Fügt ein Hochkomma in die Zeichenfolge ein.
\''	Fügt Anführungsstriche ein.
\\	Fügt einen Backslash in die Zeichenfolge ein.

Tabelle 2.2 Die Escape-Zeichen

Escape-Zeichen	Beschreibung
\a	Löst einen Alarmton aus.
\b	Führt zum Löschen des vorhergehenden Zeichens.
\f	Löst einen Formularvorschub bei Druckern aus.
\n	Löst einen Zeilenvorschub aus (entspricht der Funktionalität der ↵-Taste).
\r	Führt zu einem Wagenrücklauf.
\t	Führt auf dem Bildschirm zu einem Tabulatorsprung.
\u	Fügt ein Unicode-Zeichen in die Zeichenfolge ein.
\v	Fügt einen vertikalen Tabulator in eine Zeichenfolge ein.

Tabelle 2.2 Die Escape-Zeichen (Forts.)

Mit Escape-Sequenzen lässt sich die Ausgabe von Sonderzeichen sicherstellen. Es ist aber auch vorstellbar, dass Zeichen, die vom Compiler als Escape-Sequenz interpretiert werden, selbst Bestandteil der Zeichenfolge sind. Fügen Sie dann noch einen zweiten Backslash ein. Dazu ein kleines Beispiel. Angenommen, Sie möchten die Ausgabe

```
Hallo\nWelt
```

erzwingen. Sie müssen dann die folgende Anweisung codieren:

```
Console.WriteLine("Hallo\\nWelt");
```

Um die Interpretation als Escape-Sequenz für eine gegebene Zeichenfolge vollständig abzuschalten, wird vor der Zeichenfolge das Zeichen @ gesetzt.

```
Console.Write(@"C#\nmacht\nSpaß.");
```

Jetzt lautet die Konsolenausgabe:

```
C#\nmacht\nSpaß.
```

Die Methoden »ReadLine« und »Read«

Die Methode `ReadLine` liest ein oder mehrere Zeichen aus dem Eingabestrom – in unserem Fall ist das die Tastatur. Die Bereitschaft der Methode, auf Zeichen zu warten, endet mit dem Zeilenumbruch, der jedoch selbst nicht zu den eingelesenen Daten gehört. Die eingelesene Zeichenfolge wird von der Methode als Zeichenfolge vom Typ `string` zurückgeliefert und kann einer `string`-Variablen zugewiesen werden.

```
string input = Console.ReadLine();
Console.WriteLine(input);
```


Wir haben bisher die `ReadLine`-Methode dazu benutzt, um die Konsole bis zum Drücken der ↵-Taste geöffnet zu halten. In diesem Fall war der Eingabestrom immer leer, der Rückgabewert wurde ignoriert und landete im Nirwana.

Werfen wir nun einen Blick auf die `Read`-Methode. Diese nimmt nur ein Zeichen aus dem Eingabestrom und gibt dessen ASCII-Wert zurück. Der Rückgabewert von `Read` ist daher keine Zeichenfolge, sondern eine Zahl vom Typ `int`.

Es gibt aber noch einen weiteren, nicht weniger wichtigen Unterschied zwischen `Read` und `ReadLine`: Die `ReadLine`-Methode liest eine ganze Zeile und benutzt den Zeilenumbruch dazu, das Ende der Eingabe zu erkennen. Danach wird der Zeilenumbruch dem Eingabestrom entnommen und gelöscht. Die `Read`-Methode arbeitet anders, denn der Zeilenumbruch wird nicht aus dem Eingabestrom geholt, sondern verbleibt dort und wird so lange gepuffert, bis er von einer anderen Anweisung gelöscht wird. Das kann wiederum nur die Methode `ReadLine` sein. Schauen Sie sich dazu Listing 2.12 an:

```
static void Main(string[] args)
{
  int input = Console.Read();
  Console.WriteLine(input);
  Console.ReadLine();
}
```

Listing 2.12 Ein Zeichen mit »Console.Read« einlesen

Nach dem Start des Programms wartet `Read` auf die Eingabe des Anwenders und erkennt am Zeilenumbruch das Eingabeende. Der Zeilenumbruch befindet sich weiterhin im Eingabestrom und harrt geduldig der kommenden Anweisungen. Die Anweisung in der letzten Zeile, die `ReadLine`-Methode, reagiert als Erstes wieder auf den Eingabestrom, erkennt darin den Zeilenumbruch und verarbeitet ihn. Das ist gleichzeitig auch das Signal, mit der nächsten Anweisung fortzufahren. Da aber das Ende der `Main`-Methode erreicht ist, schließt sich das Konsolenfenster. Erst ein zweiter Aufruf von `ReadLine` würde den eigentlich angedachten Zweck erfüllen, nämlich das Fenster geöffnet zu halten und die Ausgabe der `WriteLine`-Methode auf unbestimmte Zeit anzuzeigen.

2.3.5 Die elementaren Datentypen von .NET

Die .NET-Laufzeitumgebung verfolgt das Konzept der Objektorientierung nach strengen Maßstäben. Selbst einfache Datentypen werden als Objekte angesehen, die Methoden bereitstellen, um mit einer Variablen bestimmte Aktionen auszuführen. In Tabelle 2.3 sind alle Datentypen von C# zusammenfassend aufgeführt, die allgemein als *elementare Datentypen* bezeichnet werden.

.NET-Laufzeittyp	C#-Alias	CLS-konform	Wertebereich
`Byte`	`byte`	ja	0 ... 255
`SByte`	`sbyte`	nein	–128 ... 127
`Int16`	`short`	ja	-2^{15} ... $2^{15}-1$
`UInt16`	`ushort`	nein	0 ... 65.535
`Int32`	`int`	ja	-2^{31} ... $2^{31}-1$
`UInt32`	`uint`	nein	0 ... $2^{32}-1$
`Int64`	`long`	ja	-2^{63} ... $2^{63}-1$
`UInt64`	`ulong`	nein	0 ... $2^{64}-1$
`Single`	`float`	ja	$1{,}4 \times 10^{-45}$ bis $3{,}4 \times 10^{38}$
`Double`	`double`	ja	$5{,}0 \times 10^{-324}$ bis $1{,}7 \times 10^{308}$
`Decimal`	`decimal`	ja	+/–79E27 ohne Dezimalpunktangabe; +/–7.9E–29, falls 28 Stellen hinter dem Dezimalpunkt angegeben werden. Die kleinste darstellbare Zahl beträgt +/–1.0E–29.
`Char`	`char`	ja	Unicode-Zeichen zwischen 0 und 65.535
`String`	`string`	ja	ca. 2^{31} Unicode-Zeichen
`Boolean`	`bool`	ja	`true` oder `false`
`Object`	`object`	ja	Eine Variable vom Typ `Object` kann jeden anderen Datentyp enthalten, ist also universell.

Tabelle 2.3 Die elementaren Datentypen

In der ersten Spalte ist der Typbezeichner in der .NET-Klassenbibliothek angeführt. In der zweiten Spalte steht der C#-Alias, der bei der Deklaration einer Variablen dieses Typs angegeben werden kann.

In der dritten Spalten ist angegeben, ob der Typ den Vorgaben der *Common Language Specification* (*CLS*) entspricht. Das ist, wie Sie sehen können, nicht bei allen Datentypen der Fall. Doch welche Konsequenzen hat das für Sie und Ihr Programm? Wie ich bereits in Kapitel 1, »Allgemeine Einführung in .NET«, erwähnt habe, steht C# nur an der Spitze vieler .NET-spezifischer Programmiersprachen. Alle müssen der CLS entsprechen, das ist die Spielregel. Für

die in Tabelle 2.3 aufgeführten nicht-CLS-konformen Datentypen bedeutet das, dass eine .NET-Sprache diese Typen nicht unterstützen muss. Infolgedessen sind auch unter Umständen keine Operatoren für diese Datentypen definiert und es können keine mathematischen Operationen durchgeführt werden.

Wie Tabelle 2.3 auch zu entnehmen ist, basieren alle Typen auf einer entsprechenden Definition im .NET Framework. Das hat zur Folge, dass anstelle der Angabe des C#-Alias zur Typbeschreibung auch der .NET-Laufzeittyp genannt werden kann. Damit sind die beiden folgenden Deklarationen der Variablen `value` absolut gleichwertig:

```
int value;
Int32 value;
```

Ganzzahlige Datentypen

C# stellt acht ganzzahlige Datentypen zur Verfügung, von denen vier vorzeichenbehaftet sind, der Rest nicht. Die uns interessierenden CLS-konformen Datentypen sind:

- `Byte`
- `Int16`
- `Int32`
- `Int64`

`Int16`, `Int32` und `Int64` haben einen Wertebereich, der nahezu gleichmäßig über die negative und positive Skala verteilt ist. Die vorzeichenlosen Datentypen, zu denen auch `Byte` gehört, decken hingegen nur den positiven Wertebereich, beginnend bei 0, ab. Der vorzeichenlose Typ `Byte`, der im Gegensatz zu `SByte` CLS-konform ist, ist insbesondere dann von Interesse, wenn auf binäre Daten zugegriffen wird.

Dezimalzahlen

Versuchen Sie einmal, die beiden folgenden Codezeilen zu kompilieren:

```
float value = 0.123456789;
Console.WriteLine(value);
```

Normalerweise würde man erwarten, dass der C#-Compiler daran nichts zu beanstanden hat. Dennoch zeigt er erstaunlicherweise einen Kompilierfehler an. Wie ist das zu erklären?

Auch ein Literal wie unsere Zahl 0,123456789 muss zunächst temporär in den Speicher geschrieben werden, bevor es endgültig der Variablen zugewiesen werden kann. Um eine Zahl im Speicher abzulegen, muss die Laufzeitumgebung aber eine Entscheidung treffen: Es ist die Entscheidung darüber, wie viel Speicherplatz dem Literal zugestanden wird. Das kommt aber auch der Festlegung auf einen bestimmten Datentyp gleich.

Merkregel

Literale, die eine Dezimalzahl beschreiben, werden von der .NET-Laufzeitumgebung als `Double`-Typ angesehen. Literale hingegen, die eine Ganzzahl beschreiben, werden von der Laufzeitumgebung als `int` (`Int32`) betrachtet.

Nun kommt es bei der Zuweisung unseres Literals an `value` jedoch zu einem Problem: Das Literal ist vom Typ `double`, und die Variable, die den Inhalt aufnehmen soll, ist vom Typ `float`. Per Definition weist `double` aber einen größeren Wertebereich als `float` auf – mit der Folge, dass unter Umständen vom Literal ein Wert beschrieben sein könnte, der größer ist als der, den ein `float` zu speichern vermag. Der Compiler verweigert deshalb diese Zuweisung.

Es gibt einen sehr einfachen Ausweg aus diesem Dilemma: Hängen Sie dazu an das Literal ein passendes Suffix an, hier `F` (oder gleichwertig `f`), mit dem Si den Typ `float` für das Literal erzwingen:

```
float value = 0.123456789F;
Console.WriteLine(value);
```

Nun ist der C#-Compiler in der Lage, den Inhalt an der Konsole anzuzeigen – vorausgesetzt, die Zahl entspricht dem Wertebereich eines `float`-Typs.

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

Tabelle 2.4 Typsuffix der Fließkommazahlen

Die Genauigkeit von Dezimalzahlen

Die drei Typen `float`, `double` und `decimal`, mit denen unter C# Fließkommazahlen dargestellt werden, beschreiben nicht nur unterschiedliche Wertebereiche, sondern auch – was im Grunde genommen noch viel wichtiger ist – unterschiedliche Genauigkeiten. Auf herkömmlichen Systemen beträgt die Genauigkeit eines `float`-Typs etwa zehn Stellen, die eines `double`-Typs etwa 16 Stellen, die eines `decimal`-Typs ca. 25–26. Abhängig ist die Genauigkeit dabei immer von der Anzahl der Ziffern des ganzzahligen Anteils der Dezimalzahl.

Die zeichenbasierten Datentypen »string« und »char«

Variablen vom Typ `char` können ein Zeichen des Unicode-Zeichensatzes aufnehmen. Unicode ist die Erweiterung des ein Byte großen ASCII- bzw. ANSI-Zeichensatzes mit seinen insgesamt 256 verschiedenen Zeichen. Unicode berücksichtigt die Bedürfnisse außereuropäischer Zeichensätze, für die eine Ein-Byte-Codierung nicht ausreichend ist. Jedes Unicode-Zeichen beansprucht zwei Byte, folglich ist der Unicode-Zeichensatz auf 65.536 Zeichen beschränkt. Die ersten 128 Zeichen (0–127) entsprechen denen des ASCII-Zeichensatzes, die folgenden 128 Zeichen beinhalten unter anderem Sonderzeichen und Währungssymbole.

Literale, die dem Typ `char` zugewiesen werden, werden in einfache Anführungsstriche gesetzt, z. B.:

```
char letter = 'A';
```

Um den ASCII-Wert eines einzelnen Zeichens zu erhalten, weisen Sie einfach den Typ `char` einem Zahlentyp wie beispielsweise einem `int` zu:

```
char letter = 'A';
int letterASCII = letter;
// Ausgabe: 65
Console.WriteLine(letterASCII);
```

Listing 2.13 Ermitteln des ASCII-Werts eines Characters

Die implizite Umwandlung eines `char` in einen Zahlenwert bereitet anscheinend keine Probleme, der umgekehrte Weg – die Umwandlung eines Zahlenwerts in einen `char` – ist allerdings nicht ohne weiteres möglich.

`char` beschränkt sich auf ein Zeichen. Um eine Zeichenkette, die sich aus keinem oder bis zu maximal ca. 2^{31} Einzelzeichen zusammensetzt, zu speichern oder zu bearbeiten, deklarieren Sie eine Variable vom Datentyp `string`. Die Einzelzeichen werden dabei wie bei `char` als Unicode-Zeichen der Größe 16 Bit behandelt. Zeichenketten werden grundsätzlich in doppelte Anführungsstriche gesetzt:

```
string str = "C# ist spitze."
```

Die Datentypen »Boolean«

Variablen vom Typ `bool` (`Boolean`) können nur zwei Zustände beschreiben, nämlich `true` oder `false`, z. B.:

```
bool flag = true;
```

`false` ist der Standardwert.

Hinweis

In vielen Programmiersprachen wird `false` numerisch mit 0 beschrieben und `true` durch alle Werte, die von 0 abweichen. .NET ist hier viel strenger, denn `true` ist nicht –1 und auch nicht 67, sondern ganz schlicht `true`.

Der Datentyp »Object«

Der allgemeinste aller Datentypen ist `Object`. Er beschreibt in seinen vier Byte einen Zeiger auf die Speicheradresse eines Objekts. Eine Variable dieses Typs kann jeden beliebigen anderen Datentyp beschreiben: Ob es sich um eine Zahl, eine Zeichenfolge, eine Datenbankverbindung oder um ein anderes Objekt wie zum Beispiel um die Schaltfläche in einem Window handelt, spielt dabei keine Rolle. Zur Laufzeit wird eine auf `Object` basierende Variable passend aufgelöst und die gewünschte Operation darauf ausgeführt.

Um das zu demonstrieren, ist im folgenden Codefragment eine Variable vom Typ `object` deklariert, der zuerst ein Zahlenliteral und anschließend eine Zeichenfolge zugewiesen wird:

```
object universal;
universal = 5;
Console.WriteLine(universal);
universal = "Hallo Aachen";
Console.WriteLine(universal);
```

Listing 2.14 Zuweisungen an eine Variable vom Typ »Object«

Die Variable `universal` verarbeitet beide Zuweisungen anstandslos – an der Konsole wird zuerst die Zahl 5 und danach die Zeichenfolge angezeigt.

Damit ist bei weitem noch nicht alles zum Typ `Object` gesagt. Es gibt noch zahlreiche andere Gesichtspunkte, die einer Erwähnung oder Diskussion würdig wären. Aber dazu müssen wir erst in die Tiefen der Objektorientierung gehen. Für den Moment ist die oberflächliche Erwähnung des Typs `Object` völlig ausreichend.

Die einfachen Datentypen als Objekte

Eine Variable zu deklarieren, sieht harmlos und unscheinbar aus. Und dennoch, hinter dem Variablennamen verbergen sich Möglichkeiten, die Sie bisher vermutlich noch nicht erahnen. In der .NET-Laufzeitumgebung wird alles durch die objektorientierte Brille betrachtet – sogar die einfachen Datentypen.

Ein simpler `Short` soll ein Objekt sein? Wenn Sie dieser Aussage keinen Glauben schenken wollen, schreiben Sie folgende Codezeile:

```
Int16.
```

Beachten Sie bitte hierbei den Punkt, der auf `Int16` folgt. Sie werden feststellen, dass hinter der Punktangabe eine Liste aufgeklappt wird, die *IntelliSense-Unterstützung* (siehe Abbildung 2.6).

In dieser Liste sind alle Eigenschaften und Methoden aufgeführt, die den Typ `Int16` auszeichnen. Sie können aus dem Angebot auswählen, indem Sie mit den Pfeiltasten zu der gewünschten Funktionalität navigieren und dann die ⇆-Taste drücken. Der ausgewählte Eintrag

aus IntelliSense wird vom Code übernommen, was den Vorteil hat, dass Schreibfehler ausgeschlossen sind.

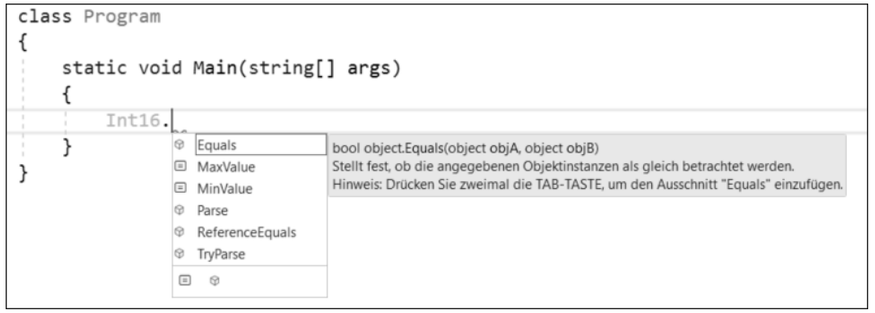

Abbildung 2.6 IntelliSense-Unterstützung in der Entwicklungsumgebung

Wenn Sie beispielsweise wissen wollen, wo die wertmäßige Ober- und Untergrenze des `Int16`-Typs liegen, könnten Sie dies mit Listing 2.15 abfragen:

```
Console.WriteLine("Int16(min) = {0}", Int16.MinValue);
Console.WriteLine("Int16(max) = {0}", Int16.MaxValue);
```

Listing 2.15 Bereichsgrenzen des »Int16«-Datentypen abfragen

An der Konsole erfolgt danach die Anzeige:

```
Int16(min) = -32768
Int16(max) = 32767
```

Wahrscheinlich werden Sie schon festgestellt haben, dass IntelliSense nicht nur im Zusammenhang mit der Punktnotation funktioniert. Sobald Sie in einer Codezeile den ersten Buchstaben eintippen, wird IntelliSense geöffnet und bietet Ihnen alle programmierbaren Optionen an, auf die mit dem eingegebenen Buchstaben zugegriffen werden kann. Die Auswahl erfolgt analog wie oben beschrieben.

2.3.6 Ausgabe ganzzahliger Datentypen

Ganzzahlige Literale lassen sich in unterschiedlichen Formaten angeben:

- im dezimalen Zahlenformat
- im hexadezimalen Zahlenformat
- im binären oder dualen Zahlenformat

Am gebräuchlichsten ist mit Sicherheit das dezimale Format, z. B.:

```
int value = 650;
```

Hexadezimale Zahlen (Basis = 16) erhalten zusätzlich das Präfix `0x` oder `0X`. Die folgende Variable `value` beschreibt die Dezimalzahl 225:

```
int value = 0xE1;
```

In der binären Schreibweise (Basis = 2) wird den Zahlen das Präfix `0b` oder `0B` vorangestellt. Somit ließe sich die Zahl 15 wie folgt abbilden:

```
int value = 0b1111;
```

Digit Separator

Große Zahlen sind unleserlich. Könnten Sie beispielsweise sofort sagen, welche Zahl durch

```
long value = 8761438740847;
```

beschrieben wird? Zur Verbesserung der Lesbarkeit wurde mit C# 7.0 der Unterstrich als Trenner (*Digit Separator*) eingeführt. Damit lässt sich die Variable `value` nun durch

```
long value = 8_761_438_740_847;
```

darstellen. Dabei spielt es keine Rolle, wie Sie die Gruppierung vornehmen: Es ist beliebig.

```
long value = 87_61_438_7_40847;
```

Digit Separators sind nicht nur auf das dezimale Zahlenformat beschränkt. Sie dürfen diese Schreibweise auch im Zusammenhang mit hexadezimalen und binären Zahlen benutzen.

2.3.7 Elementare Typkonvertierung

Sehen wir uns die folgenden beiden Anweisungen in Listing 2.16 an:

```
int value1 = 12000;
long value2 = value1;
```

Listing 2.16 Zuweisung einer »int«-Variablen an eine »long«-Variable

Hier wird die Variable `value1` vom Typ `int` deklariert und ihr ein Wert zugewiesen. Im zweiten Schritt erfolgt wiederum eine Variablendeklaration, diesmal vom Typ `long`. Der Inhalt der zuvor deklarierten Variablen `value1` wird `value2` zugewiesen. Der C#-Compiler wird beide Anweisungen anstandslos kompilieren.

Nun ändern wir die Reihenfolge von Listing 2.16 ab und deklarieren zuerst die `long`-Variable, weisen ihr den Wert von 12.000 zu und versuchen dann, *diese* der `int`-Variablen zuzuweisen:

```
long value1 = 12000;
int value2 = value1;
```

Listing 2.17 Zuweisung einer »long«-Variablen an eine »int«-Variable

Diesmal ist das Ergebnis nicht wie vielleicht erwartet – der C#-Compiler quittiert die Zuweisung mit einer Fehlermeldung, obwohl der Wertebereich eines `int`-Typs die Zuweisung von 12.000 eindeutig verkraftet.

Das auftretende Problem beruht darauf, dass der Wertebereich eines `int`- kleiner als der eines `long`-Typs ist. Im Gegensatz dazu ist die Zuweisung eines `int`-Typs an einen `long` eine zulässige *Operation*, weil der `long` einen größeren Wertebereich abdeckt als ein `int`.

Immer dann, wenn bei einer Operation zwei unterschiedliche Datentypen im Spiel sind, muss der Typ, der rechts vom Zuweisungsoperator steht, in den Typ umgewandelt werden, der sich auf der linken Seite befindet. Man spricht hierbei auch von der *Konvertierung*. Prinzipiell werden zwei Arten der Konvertierung unterschieden:

- die implizite Konvertierung
- die explizite Konvertierung

Die implizite Konvertierung

Eine *implizite Konvertierung* nimmt der C#-Compiler selbst vor und bedarf keines zusätzlichen Programmcodes. Implizit wird immer dann konvertiert, wenn der zuzuweisende Wert grundsätzlich immer kleiner oder gleich dem Datentyp ist, der den Wert empfängt. Schauen wir uns dazu Abbildung 2.7 an.

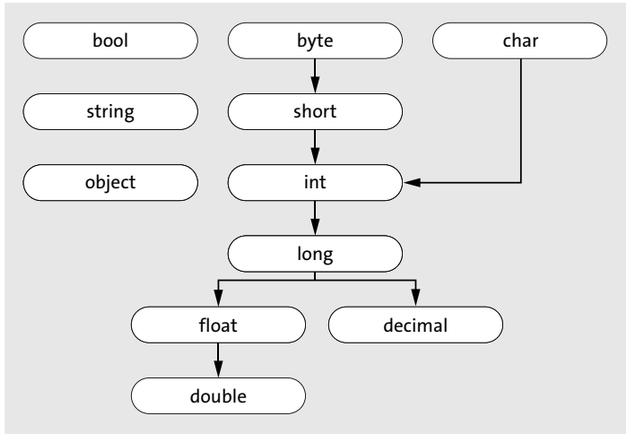

Abbildung 2.7 Die implizite Konvertierung einfacher Datentypen

Die Pfeilrichtung gibt eine implizite Konvertierung vor, entgegengesetzt der Pfeilrichtung wird explizit konvertiert. Demzufolge wird ein `byte` anstandslos implizit in einen `short`, `int`, `long` usw. konvertiert, aber nicht umgekehrt beispielsweise ein `int` in `byte`. Beachten Sie insbesondere, dass es keine impliziten Konvertierungen zwischen den Gleitkommatypen `float`/`double` und `decimal` gibt.

Eine besondere Stellung nehmen `bool`, `string`, `char` und `object` ein. Mit einem `bool` oder einem `string` sind keine impliziten Konvertierungen möglich, ein `char` kann mit Ausnahme von `byte` und `short` jedem anderen Typ zugewiesen werden. Variablen vom Typ `object` wiederum unterliegen Gesichtspunkten, die ich erst ab Kapitel 3, »Das Klassendesign«, erörtern werde.

Die explizite Konvertierung

Unter *expliziter Konvertierung* versteht man die ausdrückliche Anweisung an den Compiler, den Wert eines bestimmten Datentyps in einen anderen umzuwandeln. Explizite Konvertierung folgt einer sehr einfachen Syntax: Vor dem zu konvertierenden Ausdruck wird in runden Klammern der Typ angegeben, in den die Konvertierung erfolgen soll, also:

(*Zieldatentyp*)*Ausdruck*

Man spricht bei den so eingesetzten runden Klammern auch vom *Typkonvertierungsoperator*.

Mit der expliziten Konvertierung wären die folgenden beiden Zuweisungen möglich:

```
float value1 = 3.12F;
decimal value2 = (decimal)value1;
long value3 = 20;
int c = (int)value3;
```

Explizite Konvertierung mit den Methoden der Klasse »Convert«

Der expliziten Konvertierung mit dem Konvertierungsoperator sind Grenzen gesetzt. Beispielsweise bleibt ein boolescher Wert immer ein boolescher Wert. Damit ist die folgende Konvertierung unter C# falsch, obwohl sie in anderen Programmiersprachen durchaus zulässig ist:

```
int value = 1;
// fehlerhafte explizite Konvertierung
bool bolVar = (bool)value;
```

Um auch solche expliziten Konvertierungen zu ermöglichen, bietet die .NET-Klassenbibliothek die Klasse `Convert` an, die eine Reihe von Konvertierungsmethoden bereitstellt.

Methode	Beschreibung
`ToBoolean`(*Ausdruck*)	Konvertiert den Ausdruck in einen `bool`-Typ.
`ToByte`(*Ausdruck*)	Konvertiert den Ausdruck in einen `byte`-Typ.
`ToChar`(*Ausdruck*)	Konvertiert den Ausdruck in einen `char`-Typ.

Tabelle 2.5 Die Konvertierungsmethoden der Klasse »Convert« (Auszug)

Methode	Beschreibung
`ToDecimal`(*Ausdruck*)	Konvertiert den Ausdruck in einen `decimal`-Typ.
`ToDouble`(*Ausdruck*)	Konvertiert den Ausdruck in einen `double`-Typ.
`ToInt16`(*Ausdruck*)	Konvertiert den Ausdruck in einen `short`-Typ.

Tabelle 2.5 Die Konvertierungsmethoden der Klasse »Convert« (Auszug) (Forts.)

Damit ist das Codefragment

```
long value1 = 4711;
int value2 = (int)value1;
```

gleichwertig mit:

```
long value1 = 4711;
int value2 = Convert.ToInt32(value1);
```

In zwei ganz wesentlichen Punkten unterscheidet sich die Konvertierung mit den Methoden der `Convert`-Klasse von der mit dem Konvertierungsoperator:

- Es können Konvertierungen durchgeführt werden, die mit dem Typkonvertierungsoperator `()` unzulässig sind. Allerdings sind die Methoden der Klasse `Convert` nur elementare Datentypen sowie auf einige wenige Klassen beschränkt (z. B. `DateTime`).
- Grundsätzlich werden alle Konvertierungen mit den Methoden der `Convert`-Klasse auf einen eventuellen Überlauf hin untersucht.

Wenden wir uns an dieser Stelle zunächst dem erstgenannten Punkt zu. Angenommen, wir wollen an der Eingabeaufforderung die Eingabe in einer Integer-Variablen speichern, muss die Anweisung dazu wie folgt lauten:

```
int value = Convert.ToInt32(Console.ReadLine());
```

Bekanntlich liefert `ReadLine` die Benutzereingabe als Zeichenfolge vom Typ `string` zurück. Wäre die Methode `Convert.ToInt32` gleichwertig mit dem Typkonvertierungsoperator, würde der C#-Compiler auch die folgende Anweisung anstandslos kompilieren:

```
// fehlerhafte Anweisung
int intDigit = (int)Console.ReadLine();
```

Allerdings wird uns der Compiler diese Anweisung mit einer Fehlermeldung quittieren, denn eine explizite Konvertierung des Typs `string` in einen numerischen Typ mit dem Typkonvertierungsoperator ist auch dann unzulässig, wenn die Zeichenfolge eine Zahl beschreibt. Nur die Methoden der Klasse `Convert` sind so geprägt, dass dennoch eine Konvertierung erfolgt. Natürlich muss die Konvertierung aus logischer Sicht sinnvoll sein. Solange

aber eine Zeichenfolge eine Zahl beschreibt, darf auch eine Zeichenfolge in einen numerischen Typ überführt werden.

Bereichsüberschreitung infolge expliziter Konvertierung

Eine explizite Konvertierung lässt eine einengende Umwandlung zu, beispielsweise wenn ein `long`-Wert einer `int`-Variablen zugewiesen wird. Damit drängt sich sofort eine Frage auf: Was passiert, wenn der Wert der Übergabe größer ist als der Maximalwert des Typs, in den konvertiert wird? Nehmen wir dazu beispielsweise an, wir hätten eine Variable vom Typ `short` deklariert und ihr den Wert 436 zugewiesen. Nun soll diese Variable in den Typ `byte` überführt werden, der den Wertebereich zwischen 0–255 beschreibt.

```
short value1 = 436;
byte value2 = (byte)value1;
Console.WriteLine(value2);
```

Dieser Code resultiert in der folgenden Ausgabe:

```
180
```

Um zu verstehen, wie es zu dieser zunächst unverständlichen Ausgabe kommt, müssen wir uns die bitweise Darstellung der Zahlen ansehen. Für den Inhalt der Variablen `value1` ist das:

```
436 = 0000 0001 1011 0100
```

Nach der Konvertierung liegt das Ergebnis 180 vor, beschrieben durch:

```
180 = 1011 0100
```

Vergleichen wir jetzt die bitweise Darstellung der beiden Zahlen, kommen wir sehr schnell zu der Erkenntnis, dass bei einer expliziten Konvertierung mit dem Typkonvertierungsoperator beim Überschreiten der Bereichsgrenze des Zieldatentyps die überschüssigen Bits einfach ignoriert werden. Aus dem verbleibenden Rest wird die neue Zahl gebildet.

Dieses Verhalten kann zu sehr schwer zu lokalisierenden, ernsthaften Fehlern in einer Anwendung führen. Wenn Sie Programmcode schreiben und explizit konvertieren müssen, sollten Sie daher die Kontrolle über einen eventuell eintretenden Überlauf haben. Unter C# gibt es dazu drei Alternativen:

- die Operatoren `checked` und `unchecked`
- eine entsprechende Einstellung im *Projekteigenschaftsfenster*
- der Verzicht auf den Typkonvertierungsoperator und stattdessen die Verwendung der Klasse `Convert`

Die Operatoren »checked« und »unchecked«

Wenden wir uns zunächst den Schlüsselwörtern `checked` und `unchecked` zu, und schauen wir uns an einem Beispiel den Einsatz und die Wirkungsweise an:


```
// Beispiel: ..\Kapitel 2\CheckedSample
static void Main(string[] args)
{
  // Zahleneingabe anfordern
  Console.Write("Geben Sie eine Zahl im Bereich von ");
  Console.Write($"0...{Int16.MaxValue} ein: ");
  // Eingabe einem short-Typ zuweisen
  short value1 = Convert.ToInt16(Console.ReadLine());
  // Überlaufprüfung einschalten
  byte value2 = checked((byte)value1);
  Console.WriteLine(value2);
  Console.ReadLine();
}
```

Listing 2.18 Arithmetischen Überlauf mit »checked« prüfen

Nach dem Starten der Anwendung wird der Benutzer dazu aufgefordert, eine Zahl im Bereich von 0 bis zum Maximalwert eines `short`-Typs einzugeben. Entgegengenommen wird die Eingabe durch die Methode `Console.ReadLine`, die ihrerseits die Eingabe als Zeichenfolge, also vom Typ `string`, zurückliefert. Um die gewünschte Zahl einer `short`-Variablen zuweisen zu können, muss explizit konvertiert werden. Beachten Sie, dass wir dazu die Methode `ToInt16` der Klasse `Convert` einsetzen müssen, da eine Konvertierung eines `string`- in einen `short`-Typ mit dem Typkonvertierungsoperator nicht zulässig ist:

```
short value1 = Convert.ToInt16(Console.ReadLine());
```

Gibt der Anwender eine Zahl ein, die den Wertebereich des `short`-Typs überschreitet, wird ein Laufzeitfehler ausgelöst und die Laufzeit der Anwendung beendet. Falls der Wertebereich nicht überschritten wird, wird die dann folgende Anweisung ausgeführt:

```
byte value2 = checked((byte)value1);
```

In dieser Anweisung steckt allerdings eine Gemeinheit, denn nun soll der Inhalt der `short`-Variablen einer `byte`-Variablen zugewiesen werden. Je nachdem, welche Zahl der Anwender eingegeben hat, wird die Zuweisung fehlerfrei erfolgen oder – bedingt durch die Überprüfung mit `checked` – zu einem Fehler führen. Löschen Sie `checked` aus dem Programmcode, wird die Zuweisung einer Zahl, die den Wertebereich eines `byte`-Typs überschreitet, keinen Fehler verursachen.

`checked` ist ein Operator und wird verwendet, um einen eventuell auftretenden arithmetischen Überlauf zu steuern. Tritt zur Laufzeit ein Überlauf ein, weil der Anwender eine Zahl eingegeben hat, die den Wertebereich des Typs überschreitet, in den konvertiert werden soll, wird ein Laufzeitfehler ausgelöst, der unter .NET auch als *Ausnahme* bzw. *Exception* bezeichnet wird. Geben wir beispielsweise an der Konsole die Zahl 436 ein, werden wir die in Abbildung 2.8 gezeigte Mitteilung erhalten.

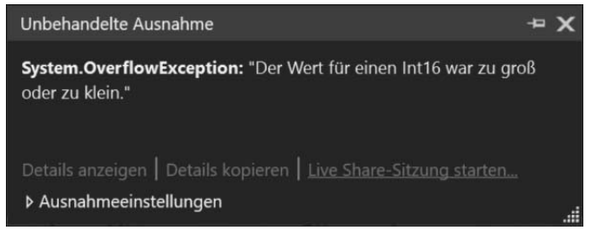

Abbildung 2.8 Fehlermeldung durch Überlauf

Nach dem Schließen der Fehlermeldung wird die Anwendung nicht ordentlich beendet. Nun könnten Sie argumentieren, dass das Beenden der Laufzeitumgebung auch nicht das sein kann, was unbedingt erstrebenswert ist. Dieses Argument ist vollkommen richtig, aber Laufzeitfehler lassen sich mittels Programmcode abfangen, und die Anwendung bleibt danach in einem ordnungsgemäßen Laufzeitzustand. Diesem Thema werden wir uns in Kapitel 7, »Fehlerbehandlung und Debugging«, noch ausgiebig widmen.

Falls nicht nur ein einzelner Ausdruck, sondern mehrere Ausdrücke innerhalb eines Anweisungsblocks auf einen möglichen Überlauf hin kontrolliert werden sollen, können Sie hinter `checked` einen Anweisungsblock angeben, innerhalb dessen der unkontrollierte Überlauf durch die Auslösung eines Laufzeitfehlers unterbunden wird. Wie diese Variante von `checked` eingesetzt wird, können Sie dem folgenden Beispiel entnehmen.

```
static void Main(string[] args)
{
  checked
  {
    short shortValue = 436;
    int integerValue = 1236555;
    byte byteValue = (byte)shtVar;
    shortValue = (short)integerValue;
    Console.WriteLine(byteValue);
    Console.ReadLine();
  }
}
```

Listing 2.19 Mehrere Ausdrücke gleichzeitig auf Überlauf hin prüfen

Wir können festhalten, dass wir mit `checked` eine gewisse Kontrolle ausüben können, falls zur Laufzeit bedingt durch die explizite Konvertierung ein Überlauf eintreten kann. Der Operator `unchecked` ist die Umkehrung der Arbeitsweise von `checked`, er schaltet die Überprüfung des Überlaufs aus und ist der Standard.

Während `checked` sich nur lokal auf den in runden Klammern stehenden Ausdruck bzw. einen eingeschlossenen Anweisungsblock bezieht, kann durch eine Änderung im Projekteigenschaftsfenster die Kontrolle über sämtliche auftretenden Überläufe in einer Anwendung ausgeübt werden. Öffnen Sie dieses Fenster, indem Sie im Projektmappen-Explorer das Projekt markieren, dessen Kontextmenü mit der rechten Maustaste öffnen und dann EIGENSCHAFTEN wählen.

Das Projekteigenschaftsfenster wird als zusätzliche Registerkarte im Code-Editor angezeigt. Am linken Rand werden mehrere Auswahloptionen angeboten. Um unser Problem zu lösen, müssen Sie sich für BUILD entscheiden. Daraufhin öffnet sich eine neue Registerkarte. In dieser sehen Sie rechts unten die Schaltfläche ERWEITERT ... Klicken sie darauf, wird ein Dialog geöffnet, der die von uns gesuchte Option anbietet: AUF ARITHMETISCHEN ÜBER-/UNTERLAUF ÜBERPRÜFEN (siehe Abbildung 2.9). Markieren Sie das Kontrollkästchen, um sicherzustellen, dass eine generelle Überprüfung auf eine Über- oder Unterschreitung des Wertebereichs erfolgt. Damit vermeiden Sie möglichen Datenverlust.

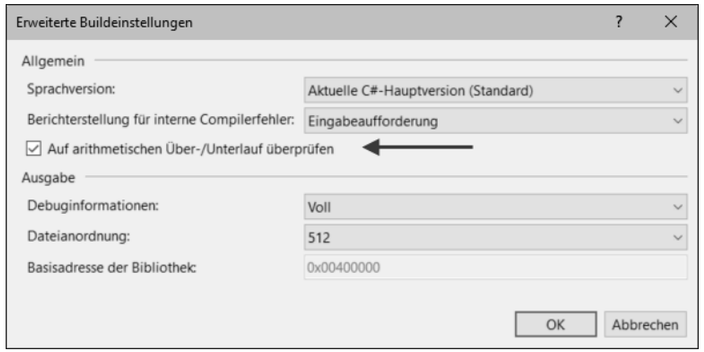

Abbildung 2.9 Einstellen der standardmäßigen Überprüfung des Überlaufs im Projekteigenschaftsfenster

Mit dieser Einstellung können Sie auf alle expliziten Angaben von `checked` verzichten, denn die Überprüfung des Unter- bzw. Überlaufs wird in der Anwendung zum Standard erklärt. Möchten Sie aus bestimmten Gründen auf die Überprüfung verzichten, kommt der Operator `unchecked` ins Spiel und hebt für den entsprechenden Ausdruck die Überprüfung wieder auf.

2.4 Operatoren

Im vorhergehenden Abschnitt haben wir uns eingehend mit den Daten auseinandergesetzt. Nun werden Sie lernen, wie Daten mit C# verarbeitet werden können. Bevor wir uns an die Details begeben, müssen wir uns zunächst mit der Terminologie befassen.

An oberster Stelle steht der Begriff *Ausdruck*. Ein Ausdruck ist die kleinste ausführbare Einheit eines Programms und setzt mindestens einen Operator voraus. Im einfachsten Fall gilt schon die Anweisung

```
value = 22;
```

als Ausdruck. Ein Ausdruck wird immer aus mindestens einem *Operanden* und einem *Operator* gebildet. Der Operator im Beispiel oben ist der Zuweisungsoperator, als Operand gilt sowohl die Konstante 22 als auch die Variable `value`. Operatoren verknüpfen Operanden miteinander und führen Berechnungen durch. Nahezu alle Operatoren von C# benötigen zwei Operanden. Das Kernkonstrukt von Ausdrücken sind die Operatoren, die sich entsprechend ihrer Arbeitsweise in verschiedene Gruppen aufteilen lassen:

- arithmetische Operatoren
- Vergleichsoperatoren
- logische Operatoren
- bitweise Operatoren
- Zuweisungsoperatoren
- sonstige Operatoren

2.4.1 Arithmetische Operatoren

C# kennt die üblichen Operatoren der vier Grundrechenarten Addition, Subtraktion, Division und Multiplikation. Darüber hinaus werden von dieser Gruppe weitere Operatoren beschrieben, die in ihrem Kontext eine besondere Bedeutung haben. Tabelle 2.6 gibt zunächst einen allgemeinen Überblick.

Operator	Beschreibung
+	Hat zwei Funktionalitäten: ▶ Als Additionsoperator bildet er die Summe zweier Operanden (x + y). ▶ Als Vorzeichenoperator beschreibt er eine positive Zahl (+x), ist also ein einstelliger (unärer) Operator.
-	Hat ebenfalls zwei Funktionalitäten: ▶ Als Subtraktionsoperator eingesetzt, bildet er die Differenz zweier Operanden (x - y). ▶ Als unärer Vorzeichenoperator beschreibt er eine negative Zahl (-x).
*	Multiplikationsoperator; multipliziert zwei Operanden (x * y).

Tabelle 2.6 Arithmetische Operatoren

Operator	Beschreibung
/	Divisionsoperator; dividiert zwei Operanden (x / y), behält den Nachkommateil der Division.
%	Restwertoperator; dividiert zwei Operanden und liefert als Ergebnis den Restwert der Operation (x % y).
++	Erhöht den Inhalt des Operanden um 1. Das Ergebnis der Operation ++x ist der Wert des Operanden nach der Erhöhung. Das Ergebnis der Operation x++ ist der Wert des Operanden vor der Erhöhung.
--	Verringert den Inhalt des Operanden um 1. Das Ergebnis der Operation --x ist der Wert des Operanden nach der Verringerung. Das Ergebnis der Operation x-- ist der Wert des Operanden vor der Verringerung.

Tabelle 2.6 Arithmetische Operatoren (Forts.)

Der Einsatz der Operatoren zur Formulierung mathematischer Ausdrücke ist trivial. Zwei Operanden werden miteinander verknüpft, und das Ergebnis der Operation wird der links vom Zuweisungsoperator stehenden Variablen zugewiesen.

```
int value1 = 30;
int value2 = 55;
int result = value1 + value2;
```

Eine besondere Stellung nimmt der %-Operator ein, dessen Ergebnis einer Division der ganzzahlige Divisionsrest ist. Dazu ein Beispiel:

```
int x = 100;
int y = 17;
Console.WriteLine("Division mit % - Ergebnis = {0}", x % y);
```

Die Zahl 17 ist fünfmal in der Zahl 100 enthalten. Damit lautet die Konsolenausgabe `15`.

Dezimalzahlen als Operanden des %-Operators sind ebenfalls zugelassen, die Rückgabe ist dabei selbst eine Dezimalzahl:

```
float x = 100.35F;
float y = 17.45F;
Console.WriteLine("Division mit % - Ergebnis = {0}", x % y);
```

Wenn Sie diesen Code ausführen, wird im Konsolenfenster die Ausgabe `13,09999` erscheinen.

Widmen wir uns jetzt noch einem Beispiel, an dem die Arbeitsweise der Inkrement- bzw. Dekrement-Operationen ersichtlich wird.

Zunächst betrachten wir das Codefragment einer Postfixinkrement-Operation:

```
int x = 5;
int y = x++; // x hat den Inhalt 6 und y den Inhalt 5
```

Zuerst wird der Variablen x der Wert 5 zugewiesen. Im zweiten Schritt wird der aktuelle Inhalt von x an y übergeben und danach x um eins erhöht. Nach Beendigung der zweiten Anweisung weist x den Inhalt 6 auf und y den Inhalt 5.

Ein abweichendes Ergebnis erhalten wir, wenn wir den ++-Operator als Präfixinkrementoperator einsetzen.

```
int x = 5;
int y = ++x; // x und y haben den Inhalt 6
```

In diesem Fall wird zuerst der Inhalt der Variablen x um eins erhöht, und erst danach erfolgt die Zuweisung an die Variable y. Die Folge ist, dass sowohl x als auch y den Wert 6 beschreiben.

Vielleicht wird Ihnen aufgefallen sein, dass Tabelle 2.6 keinen Potenzoperator beschreibt. Das ist keine Unterlassungssünde des Autors, da C# tatsächlich keinen bereitstellt. Stattdessen gibt es in der .NET-Klassenbibliothek eine Klasse namens `Math`, die diverse Methoden für mathematische Operationen bereitstellt, unter anderem die Methode `Pow` zum Potenzieren. Wollen Sie beispielsweise das Ergebnis von 2^5 berechnen, müssen Sie Folgendes codieren:

```
double value = Math.Pow(2, 5);
```

Hinweis

In C# gibt es tatsächlich den ^-Operator. Allerdings wird er als bitweiser Xor-Operator verwendet (siehe Abschnitt 2.4.4).

Besonderheiten einer Division

Bei einer Division zweier ganzer Zahlen gibt es einen Haken, der im ersten Moment nicht offensichtlich ist. Betrachten Sie dazu die beiden folgenden Anweisungen:

```
double value = 3/4;
Console.WriteLine(value);
```

An der Konsole wird nicht, wie zu erwarten wäre, das Ergebnis `0.75` angezeigt, sondern `0`. Die Begründung dieses Phänomens ist recht einfach. Zur Laufzeit muss für die beiden Literale 3 und 4 Speicher reserviert werden. Die Laufzeitumgebung erkennt, dass es sich um ganze Zahlen handelt, und interpretiert den Typ der beiden Literale jeweils als `int`. Das Ergebnis der Division wird vor der endgültigen Zuweisung an `value` zwischengespeichert. Dazu wird Speicherplatz reserviert, der dem Typ des größten der beiden beteiligten Operanden entspricht,

mit der Folge, dass der Dezimalteil des Ergebnisses abgeschnitten wird. Bei der anschließenden Zuweisung an `value` ist das Kind bereits in den Brunnen gefallen – das Ergebnis ist falsch.

Zur Lösung dieser Problematik muss sichergestellt werden, dass einer der beiden Operanden als Dezimalzahl erkannt wird. Sie können das erreichen, indem Sie beispielsweise

```
double value = 3.0/4;
```

codieren. Die Zahl 3 wird jetzt nicht mehr als Integer, sondern als `double` verarbeitet. Dieser Typ ist erfreulicherweise in der Lage, auch Nachkommastellen aufzunehmen, und das Ergebnis wird korrekt angezeigt.

Eine andere Möglichkeit wäre es, einen der Operanden explizit in eine Dezimalzahl zu konvertieren:

```
double value = (double)3 / 4;
```

2.4.2 Vergleichsoperatoren

Vergleichsoperatoren vergleichen zwei Ausdrücke miteinander. Der Rückgabewert ist immer ein boolescher Wert, also entweder `true` oder `false`. Vergleiche können auf Gleichheit bzw. Ungleichheit sowie auf »größer« und »kleiner« durchgeführt werden.

Operator	Beschreibung
`a == b`	(Vergleichsoperator) prüft, ob Ausdruck `a` Ausdruck `b` entspricht, und gibt in diesem Fall `true` zurück.
`a != b`	Ergebnis der Operation ist `true`, wenn `a` ungleich `b` ist.
`a > b`	Ergebnis der Operation ist `true`, wenn `a` größer `b` ist.
`a < b`	Ergebnis der Operation ist `true`, wenn `a` kleiner `b` ist.
`a <= b`	Ergebnis der Operation ist `true`, wenn `a` kleiner oder gleich `b` ist.
`a >= b`	Ergebnis der Operation ist `true`, wenn `a` größer oder gleich `b` ist.

Tabelle 2.7 Vergleichsoperatoren

Sehen wir uns einige boolesche Ausdrücke an:

```
bool compare;
compare = value <= 100;
```

Vergleichsoperatoren genießen eine höhere Priorität als der Zuweisungsoperator, daher wird zuerst der Teilausdruck `value <= 100` ausgewertet. Das Ergebnis des Vergleichs, je nachdem, ob der Vergleich wahr oder falsch ist, wird der Variablen `compare` zugewiesen. Sie können die boolesche Operation auch direkt zur Initialisierung bei der Deklaration verwenden:

```
bool compare = intVar <= 100;
string text1 = "Hallo";
string text2 = "hallo";
bool compare = text1 == text2;
```

2.4.3 Logische Operatoren

Operator	Beschreibung
!	Unärer Negationsoperator. Der Ausdruck !a ist true, wenn a einen unwahren Wert beschreibt, und false, wenn a wahr ist.
&	(And-Operator, 1. Variante) Der Ausdruck a & b ist dann true, wenn sowohl a als auch b true sind. Dabei werden in jedem Fall beide Ausdrücke ausgewertet.
|	(Or-Operator, 1. Variante) Der Ausdruck a | b ist true, wenn entweder a oder b wahr ist. Dabei werden in jedem Fall beide Ausdrücke ausgewertet.
^	(Xor-Operator) Der Ausdruck a ^ b ist true, wenn die beiden beteiligten Operanden unterschiedliche Wahrheitswerte haben.
&&	(And-Operator, 2. Variante) Der Ausdruck a && b ist true, wenn sowohl a als auch b true sind. Zuerst wird a ausgewertet. Sollte a false sein, ist in jedem Fall der Gesamtausdruck unabhängig von b auch falsch. b wird dann nicht mehr ausgewertet.
||	(Or-Operator, 2. Variante) Der Ausdruck a || b ist true, wenn entweder a oder b true ist. Zuerst wird a ausgewertet. Sollte a bereits true sein, ist in jedem Fall der Gesamtausdruck unabhängig von b auch wahr. b wird dann nicht mehr ausgewertet.

Tabelle 2.8 Logische Operatoren

Das Ergebnis einer Operation, an der logische Operatoren beteiligt sind, lässt sich am besten anhand einer Wahrheitstabelle darstellen.

Bedingung 1	Bedingung 2	And-Operator	Or-Operator	Xor-Operator
false	false	false	false	false
true	false	false	true	true
false	true	false	true	true
true	true	true	true	false

Tabelle 2.9 Wahrheitstabelle

Sehr häufig werden logische Operatoren benutzt, wenn eine Entscheidung darüber getroffen werden muss, welcher Programmcode in Abhängigkeit vom Ergebnis einer Bedingungsprüfung ausgeführt werden soll:

```
if(x != y)
  Console.WriteLine("x ist ungleich y");
```

In diesem einfachen Beispiel, das auch ohne größere Erklärung verständlich sein dürfte, wird die WriteLine-Methode dann ausgeführt, wenn die Bedingung

```
x != y
```

erfüllt ist, also true liefert.

Bedingungen können durchaus komplexer werden und neben logischen Operatoren auch mehrere Vergleichsoperatoren enthalten. Betrachten wir das folgende Codefragment:

```
if(x < 5 || y > 20)
  Console.WriteLine("Bedingung ist erfüllt");
```

In diesem Codefragment haben wir es mit drei verschiedenen Operatoren zu tun. Da stellt sich sofort die Frage, in welcher Reihenfolge sie zur Bildung des Gesamtergebnisses herangezogen werden. Von den drei Operatoren hat der ||-Operator die geringste Priorität, < und > sind in dieser Hinsicht gleichwertig (siehe Abschnitt 2.4.8). Folglich wird zuerst das Ergebnis aus

```
x < 5
```

gebildet und danach das aus

```
y > 20
```

Beide Teilergebnisse sind entweder true oder false und werden am Schluss mit || verglichen, woraus das endgültige Resultat gebildet wird. Manchmal ist es allerdings wegen der besseren Lesbarkeit einer komplexen Bedingung durchaus sinnvoll, auch im Grunde genommen überflüssige Klammerpaare zu setzen:

```
if((x < 5) || (y > 20))
```

Interessant sind insbesondere die ähnlichen Paare & und && bzw. | und ||. Um die Unterschiede in der Verhaltensweise genau zu verstehen, wollen wir ein kleines Beispielprogramm entwickeln, das auch syntaktische Elemente enthält, die bisher noch nicht unser Thema waren.

```
// Beispiel: ..\Kapitel 2\LogischeOperatoren
class Program
{
  static void Main(string[] args)
  {
```

```
    int x = 8;
    int y = 9;
    // wenn die Bedingung wahr ist, dann dies durch eine
    // Ausgabe an der Konsole bestätigen
    if ((x != y) | DoSomething())
      Console.WriteLine("Bedingung ist erfüllt");
    Console.ReadLine();
  }
  // benutzerdefinierte Methode
  static bool DoSomething()
  {
    Console.WriteLine("in DoSomething");
    return true;
  }
}
```

Listing 2.20 Testen einer komplexeren Bedingungsprüfung

Neu ist in diesem Beispiel die Definition einer Methode, die hier `DoSomething` heißt. `DoSomething` macht nicht sehr viel: Sie schreibt nur eine Meldung in das Konsolenfenster und gibt immer den booleschen Wert `true` als Ergebnis des Aufrufs zurück. In `Main` werden den beiden Variablen `x` und `y` feste Werte zugewiesen. Daraus folgt, dass die Bedingung

```
x != y
```

immer wahr ist. Verknüpft wird diese Bedingung über den Oder-Operator | mit dem Aufruf der benutzerdefinierten Funktion. Da diese einen booleschen Wert zurückliefert, ist der Code syntaktisch korrekt. Führen wir das Programm aus, wird an der Konsole

```
in DoSomething
Bedingung ist erfüllt
```

angezeigt. Halten wir an dieser Stelle die folgende Tatsache fest: Zwei Ausdrücke sind mit dem Oder-Operator | verknüpft. Beide Bedingungen werden vollständig geprüft, bevor das Gesamtergebnis der Operation feststeht. Der Wahrheitstabelle (Tabelle 2.9) können wir aber entnehmen, dass die Gesamtbedingung in jedem Fall `true` ist, wenn einer der beiden Ausdrücke wahr ist. Folglich wäre es auch vollkommen ausreichend, nach dem Prüfen der Bedingung `x != y` die zweite Bedingung keiner eigenen Überprüfung zu unterziehen, da das Endergebnis bereits feststeht. Hier betritt nun der zweite Oder-Operator (||) die Bühne. Wenn wir die Bedingung nun mit

```
if((x != y) || DoSomething())
```

formulieren, lautet die Ausgabe an der Konsole nur noch:

```
Bedingung ist erfüllt
```


Der Wahrheitsgehalt der zweiten Bedingung wird erst gar nicht mehr überprüft, da er das Endergebnis nicht mehr beeinflussen kann. Genauso arbeiten auch die beiden Operatoren & und &&.

In der Praxis kann dieser Unterschied bedeutend hinsichtlich der Performance einer Anwendung sein. Wenn die zweite Bedingung nämlich eine länger andauernde Ausführungszeit für sich beansprucht und das Ergebnis der ersten Operation die Prüfung der zweiten Bedingung unnötig macht, leisten || bzw. && durchaus ein kleinen Beitrag zur Verbesserung der Gesamtleistung.

2.4.4 Bitweise Operatoren

Bitweise Operatoren dienen dazu, auf die Bitdarstellung numerischer Operanden zuzugreifen. Dabei kann die Bitdarstellung eines numerischen Operanden sowohl abgefragt als auch manipuliert werden.

Operator	Beschreibung
~	Invertiert jedes Bit des Ausdrucks (Einerkomplement).
|	Aus x|y resultiert ein Wert, bei dem die korrespondierenden Bits von x und y Or-verknüpft werden.
&	Aus x&y resultiert ein Wert, bei dem die korrespondierenden Bits von x und y And-verknüpft werden.
^	Aus x^y resultiert ein Wert, bei dem die korrespondierenden Bits von x und y Xor-verknüpft werden.
<<	Aus x<<y resultiert ein Wert, der durch die Verschiebung der Bits des ersten Operanden x um die durch im zweiten Operanden y angegebene Zahl nach links entsteht.
>>	Aus x>>y resultiert ein Wert, der durch die Verschiebung der Bits des ersten Operanden x um die durch im zweiten Operanden y angegebene Zahl nach rechts entsteht.

Tabelle 2.10 Bitweise Operatoren

Beachten Sie, dass die Operatoren & und | sowohl als Vergleichsoperatoren (vergleiche auch mit Tabelle 2.7) auch als bitweise Operatoren eingesetzt werden können. Als Vergleichsoperatoren werden zwei boolesche Operanden miteinander verglichen und ein Wahrheitswert als Ergebnis der Operation zurückgeliefert. Bitweise Operatoren vergleichen hingegen die einzelnen Bits einer bestimmten Speicheradresse und bilden daraus das Ergebnis. Wir sehen uns jetzt an einigen Beispielen an, wie Sie diese Operatoren einsetzen können.

Beispiel 1: Im Folgenden werden die beiden Literale 13 und 5 mit dem bitweisen &-Operator verknüpft:

```
a = 13 & 5;
Console.WriteLine(a);
```

Die Bitdarstellung dieser beiden Literale sieht wie folgt aus:

```
13 = 0000 0000 0000 1101
 5 = 0000 0000 0000 0101
```

An der Konsole wird als Ergebnis die Zahl 5 angezeigt, was der Bitdarstellung

```
0000 0000 0000 0101
```

entspricht. Wir können unser Ergebnis auch wie folgt interpretieren:

Eine vorgegebene Bitsequenz kann mit dem bitweisen &-Operator daraufhin untersucht werden, ob die vom rechten Operanden beschriebenen Bits in der vorgegebenen Bitfolge gesetzt sind. Das ist genau dann der Fall, wenn das Ergebnis der &-Verknüpfung dasselbe Ergebnis liefert wie im rechtsseitigen Operanden angegeben.

Beispiel 2: Verknüpfen wir nun zwei Literale mit dem bitweisen Oder-Operator |, also beispielsweise:

```
int a = 71 | 49;
Console.WriteLine(a);
```

Die Bitdarstellung dieser beiden Literale sieht wie folgt aus:

```
71 = 0000 0000 0100 0111
49 = 0000 0000 0011 0001
```

Das Ergebnis wird 119 lauten oder in Bitdarstellung:

```
0000 0000 0111 0111
```

Beispiel 3: Dem Xor-Operator ^ kommt ebenfalls eine ganz besondere Bedeutung zu, wie das folgende Beispiel zeigt:

```
int a = 53;
a = a ^ 22;
Console.WriteLine(a);
```

Sehen wir uns zunächst wieder die durch die beiden Literale beschriebenen Bitsequenzen an:

```
53 = 0000 0000 0011 0101
22 = 0000 0000 0001 0110
```


Lassen wir uns das Ergebnis an der Konsole anzeigen, wird 35 ausgegeben. Das entspricht folgender Bitfolge:

```
0000 0000 0010 0011
```

Hier werden also das zweite, dritte und das fünfte Bit des linken Operanden invertiert – so wie es der rechte Operand vorgibt. Analysieren wir das Ergebnis, kommen wir zu der folgenden Merkregel:

In einer vorgegebenen Bitsequenz können ganz bestimmte Bits mit dem bitweisen ^-Operator invertiert werden. Die Ausgangsbitfolge steht links vom Operator, und die Zahl, die die Bits repräsentiert, die invertiert werden sollen, steht rechts vom Operator.

Wenden wir auf das Ergebnis ein zweites Mal den ^-Operator an, also

```
int a = 53;
a = a ^ 22;
a = a ^ 22;
```

wird die Variable a wieder den ursprünglichen Wert 53 enthalten.

Beispiel 4: Zum Abschluss nun noch ein Beispiel mit dem Verschiebeoperator <<. Die Bits der Zahl 37 sollen um zwei Positionen nach links verschoben werden, und die Anzeige soll sowohl im Dezimal- als auch im Hexadezimalformat erfolgen.

```
c = 37 << 2;
Console.WriteLine("dezimal    : {0}",c);
Console.WriteLine("hexadezimal: 0x{0:x}",c);
```

Die Zahl 37 entspricht der Bitdarstellung:

```
0000 0000 0010 0101
```

Nach der Verschiebung um die geforderten zwei Positionen nach links ergibt sich:

```
0000 0000 1001 0100
```

Das entspricht wiederum der Zahl 148 oder in hexadezimaler Schreibweise 0x94, was uns auch die Laufzeitumgebung bestätigt.

Mit

```
c = 37 >> 2;
```

lautet das Ergebnis 9, was zu der folgenden Aussage führt:

Bei der Bitverschiebung eines positiven Operanden mit dem <<- oder >>-Operator werden die frei werdenden Leerstellen mit 0-Bits aufgefüllt.

2.4.5 Zuweisungsoperatoren

Bis auf die Ausnahme des einfachen Gleichheitszeichens dienen alle anderen Zuweisungsoperatoren zur verkürzten Schreibweise einer Anweisung, bei der der linke Operand einer Operation gleichzeitig der Empfänger des Operationsergebnisses ist.

Operator	Beschreibung
=	x = y weist x den Wert von y zu.
+=	x += y weist x den Wert von x + y zu.
-=	x -= y weist x den Wert von x - y zu.
*=	x *= y weist x den Wert von x * y zu.
/=	x /= y weist x den Wert von x / y zu.
%=	x %= y weist x den Wert von x % y zu.
&=	x &= y weist x den Wert von x & y zu.
|=	x |= y weist x den Wert von x | y zu.
^=	x ^= y weist x den Wert von x ^ y zu.
<<=	x <<= y weist x den Wert von x << y zu.
>>=	x >>= y weist x den Wert von x >> y zu.

Tabelle 2.11 Zuweisungsoperatoren

2.4.6 Stringverkettung

Den +-Operator haben Sie bereits in Verbindung mit arithmetischen Operationen kennengelernt. Ihm kommt allerdings eine zweite Aufgabe zu, nämlich die Verkettung von Zeichenfolgen. Ist wenigstens einer der beiden an der Operation beteiligten Operanden vom Typ `string`, bewirkt der +-Operator eine Stringverkettung. Bei Bedarf wird der Operand, der nicht vom Typ `string` ist, implizit in einen solchen konvertiert. Das Ergebnis der Stringverkettung ist wieder eine Zeichenfolge. Nachfolgend finden Sie einige Anweisungen, die Beispiele für Stringverkettungen zeigen.

```
string text1 = "Leckere";
string text2 = "Suppe";
// text3 hat den Inhalt "leckere Suppe"
string text3 = text1 + " " + text2;
int value = 4711;
string text = "Hallo";
```



```
// text hat den Inhalt "Hallo4711"
text += value;
string text1 = "4";
string text2 = "3";
// an der Konsole wird "43" ausgegeben
Console.WriteLine(text1 + text2);
```

2.4.7 Sonstige Operatoren

Wir sind noch nicht am Ende der Aufzählung der Operatoren von C# angelangt. Es stehen dem Entwickler noch einige besondere Operatoren zur Verfügung, mit denen Sie in den vorhergehenden Abschnitten teilweise schon gearbeitet haben oder die Sie im weiteren Verlauf dieses Buches noch kennenlernen werden. Der Vollständigkeit halber sind die Operatoren dieser Gruppe in Tabelle 2.12 aufgeführt.

Operator	Beschreibung
.	Der Punktoperator wird für den Zugriff auf die Eigenschaften oder Methoden einer Klasse verwendet, z. B. `Console.ReadLine();`
[]	Der []-Operator wird für Arrays, Indexer und Attribute verwendet, z. B. `arr[10]`.
()	Der ()-Operator dient zwei Zwecken: Er gibt die Reihenfolge der Operationen vor und wird auch zur Typkonvertierung eingesetzt.
?:	Der ?:-Operator gibt einen von zwei Werten in Abhängigkeit von einem dritten zurück. Er ist eine einfache Variante der `if`-Bedingungsprüfung.
new	Dient zur Instanziierung einer Klasse.
is	Prüft den Laufzeittyp eines Objekts mit einem angegebenen Typ.
typeof	Ruft das `System.Type`-Objekt für einen Typ ab.
checked/unchecked	Steuert die Reaktion der Laufzeitumgebung bei einem arithmetischen Überlauf.

Tabelle 2.12 Sonstige C#-Operatoren

2.4.8 Operator-Vorrangregeln

Enthält ein Ausdruck mehrere Operatoren, entscheiden die Operator-Vorrangregeln über die Reihenfolge der Ausführung der einzelnen Operationen. In Tabelle 2.13 sind die Operatoren so angeordnet, dass die weiter oben stehenden Vorrang vor den weiter unten stehenden haben.

Rang	Operator
1	x.y (Punktoperator), a[x], x++, x--, new, typeof, checked, unchecked
2	+ (unär), - (unär), !, ~, ++x, --x, (<Typ>)x
3	*, /, %
4	+ (additiv), - (subtraktiv)
5	<<, >>
6	<, >, <=, >=, is
7	==, !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Tabelle 2.13 Operator-Vorrangregeln

2.5 Datenfelder (Arrays)

Arrays, die manchmal auch als Datenfelder bezeichnet werden, ermöglichen es, eine nahezu beliebig große Anzahl von Variablen gleichen Namens und gleichen Datentyps zu definieren. Unterschieden werden die einzelnen Elemente nur anhand einer Indizierung. Arrays kommen insbesondere dann zum Einsatz, wenn in Programmschleifen dieselben Operationen auf alle oder einen Teil der Elemente ausgeführt werden sollen.

2.5.1 Die Deklaration und Initialisierung eines Arrays

Die Deklaration eines Arrays wird am besten an einem Beispiel verdeutlicht:

```
int[] elements;
```


Mit dieser Anweisung wird das Array `elements` deklariert, das Integer-Werte beschreibt. Um wie viele es sich handelt, ist noch nicht festgelegt. Die Kennzeichnung als Array erfolgt durch die eckigen Klammern, die hinter dem Datentyp angegeben werden. Danach folgt der Bezeichner des Arrays.

Das Array `elements` ist zwar deklariert, aber noch nicht initialisiert. Insbesondere benötigt die Laufzeitumgebung eine Angabe darüber, wie viele Elemente sich im Array befinden. Arrays werden von der .NET-Laufzeitumgebung als Objekt angesehen, deshalb unterscheidet sich die Initialisierung von der einer herkömmlichen Variablen:

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

Das Schlüsselwort `new` kennzeichnet die Erzeugung eines Objekts, dahinter wird der Datentyp genannt. Die Anzahl der Array-Elemente – man spricht auch von der Größe des Arrays – geht aus der Zahlenangabe in den eckigen Klammern hervor: In unserem Fall verwaltet das Array `elements` genau drei Integer. Die Angabe in den eckigen Klammern der Initialisierung ist immer eine Zahl vom Typ `int`.

Merke

Die Anzahl der Elemente eines Arrays ergibt sich aus der Angabe in den eckigen Klammern bei der Initialisierung mit `new`.

Eine alternative, gleichwertige Deklarations- und Initialisierungsanweisung ist einzeilig und bietet sich an, wenn bei der Deklaration bekannt ist, wie viele Elemente das Array haben soll:

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

Alle Elemente dieses Arrays sind danach mit dem Wert 0 vorinitialisiert. Steht zum Deklarationszeitpunkt bereits fest, welche Daten die Array-Elemente aufnehmen sollen, bietet sich auch die *literale Initialisierung* an, bei der die Daten in geschweiften Klammern bekannt gegeben werden:

```
int[] elements = new int[3]{23, 9, 7};
```

Gleichwertig ist auch diese Initialisierung:

```
int[] elements = new int[]{23, 9, 7};
```

Wer es ganz besonders kurz mag, darf auch die folgende Schreibweise einsetzen, bei der die Größe des Arrays automatisch anhand der Anzahl der zugewiesenen Elemente bestimmt wird:

```
int[] elements = {23, 9, 7};
```

Die literale Initialisierung setzt voraus, dass allen Elementen ein gültiger Wert übergeben wird. Deshalb ist die folgende Initialisierung falsch:

```
// falsche literale Initialisierung
int[] elements = new int[3] { 23 };
```

2.5.2 Der Zugriff auf die Array-Elemente

Bei der Initialisierung eines Arrays werden die einzelnen Elemente durchnummeriert. Dabei hat das erste Element den Index 0, das letzte Element den Index

Anzahl der Elemente - 1

Ein Array, das mit

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

deklariert und initialisiert worden ist, enthält somit drei Elemente: `elements[0]`, `elements[1]` und `elements[2]`. Beabsichtigen wir, dem ersten Element des Arrays die Zahl 55 zuzuweisen, müsste die Anweisung wie folgt lauten:

```
elements[0] = 55;
```

Analog erfolgt auch die Auswertung des Elementinhalts durch die Angabe des Index:

```
int value = elements[0];
```

Im folgenden Beispiel werden zwei Arrays deklariert und mit Werten initialisiert, die anschließend an der Konsole ausgegeben werden.

```
// Beispiel: ..\Kapitel 2\ArraySample
class Program
{
  static void Main(string[] args)
  {
    long[] lngVar = new long[4];
    string[] strArr = new String[2];
    // Wertzuweisungem
    lngVar[0] = 230;
    lngVar[1] = 4711;
    lngVar[3] = 77;
    strArr[0] = "C# ";
    strArr[1] = "macht Spaß!";
    // Konsolenausgaben
    Console.WriteLine("lngVar[0] = {0}", lngVar[0]);
    Console.WriteLine("lngVar[1] = {0}", lngVar[1]);
    Console.WriteLine("lngVar[2] = {0}", lngVar[2]);
    Console.WriteLine("lngVar[3] = {0}", lngVar[3]);
```



```
    Console.Write(strArr[0]);
    Console.WriteLine(strArr[1]);
    Console.ReadLine();
  }
}
```

Listing 2.21 Beispielprogramm mit einem Array

Das Array `lngVar` hat eine Größe von insgesamt vier Elementen und ist vom Typ `long`; das Array `strArr` vom Typ `string` enthält zwei Elemente. Bis auf das dritte Element des `long`-Arrays mit dem Index 2 wird allen Elementen ein Wert zugewiesen. Die Ausgabe des Programms zur Laufzeit lautet:

```
lngVar[0] = 230
lngVar[1] = 4711
lngVar[2] = 0
lngVar[3] = 77
C# macht Spass!
```

2.5.3 Mehrdimensionale Arrays

Die bisher behandelten Arrays können Sie sich als eine einfache Folge von Daten auf einer Geraden vorstellen. Sie werden als *eindimensionale Arrays* bezeichnet. Zur Darstellung komplexer Datenstrukturen, beispielsweise räumlicher, sind eindimensionale Arrays aber nicht besonders gut geeignet. Daher kommen in der Praxis auch häufig zweidimensionale oder noch höher dimensionierte Arrays zum Einsatz.

Ein *zweidimensionales Array* können Sie sich als Matrix oder Tabelle vorstellen. Bekanntermaßen ist jede Zelle einer Tabelle eindeutig durch die Position in einer Reihe und einer Spalte identifizierbar. Um den Inhalt einer Tabellenzelle durch ein bestimmtes Array-Element zu beschreiben, bietet sich ein zweidimensionales Array an: Eine Dimension beschreibt die Reihe, die andere Dimension die Spalte.

Angenommen, eine Tabelle hat vier Reihen und drei Spalten, dann könnte die Deklaration

```
int[,] zelle = new int[4,3];
```

lauten. Etwas schwieriger ist die literale Initialisierung eines *mehrdimensionalen Arrays*. Jede Dimensionsebene wird durch ein Paar geschweifte Klammern dargestellt, bei einem eindimensionalen Array also – wie oben eingangs gezeigt – durch ein Klammerpaar:

{Anzahl der Elemente der ersten Dimension}

Da ein zweidimensionales Array als ein Feld zu verstehen ist, bei dem jedes Array-Element selbst wieder ein eigenes Feld gleichen Typs definiert, wird jedes Element der Initialisierung

eines eindimensionalen Arrays durch ein Paar geschweifter Klammern ersetzt, in dem wiederum Werte des »Unterarrays« angegeben werden:

{{Anzahl der Elemente der zweiten Dimension}, { }, ...}

Die literale Zuweisung an ein zweidimensionales Array könnte demnach wie folgt aussehen:

```
int[,] point = new int[,]{{1,2,3},{4,5,6}};
```

Zulässig ist ebenfalls die kürzere Schreibweise mit:

```
int[,] point = {{1,2,3},{4,5,6}};
```

Diese Systematik setzt sich mit jeder weiteren Dimension fort. Beispielhaft sei das noch an einem dreidimensionalen Array gezeigt:

{{{Anzahl der Elemente der dritten Dimension}, { }, ...}, { }, ...}

Das folgende Codebeispiel zeigt anhand eines dreidimensionalen Arrays, dass die Initialisierung mit zunehmender Dimensionstiefe schon verhältnismäßig komplex und dazu auch noch schlecht lesbar ist:

```
int[,,] elements = {
                     { {1,2,3,4},{3,4,5,6},{6,7,8,9}},
                     { {3,4,6,1},{6,19,3,4},{4,1,8,7}}
                   };
```

Das Array `elements` entspricht einem Array `elements[2,3,4]`. Es weist in der dritten Dimension vier Elemente auf, in der zweiten drei und in der ersten zwei.

Beim Zugriff auf ein mehrdimensionales Array müssen Sie jede Dimension des entsprechenden Elements angeben. Beispielsweise schreibt die Anweisung

```
Console.WriteLine(elements[1,1,1]);
```

die Zahl 19 in das Konsolenfenster.

2.5.4 Festlegen der Array-Größe zur Laufzeit

Nicht immer sind wir in der glücklichen Lage, schon zur Entwicklungszeit die Größe eines Arrays zu kennen, da diese sich möglicherweise erst zur Laufzeit ergibt. In dieser Situation kann die Festlegung der Größe auch über eine Variable erfolgen, die zur Laufzeit mit einem konkreten Wert initialisiert wird. Das folgende Beispiel demonstriert das. Die Aufgabenstellung soll dabei sein, jedem Array-Element als Wert das Quadrat seines Index zuzuweisen.

```
// Beispiel: ..\Kapitel 2\ArrayInitialisierung
class Program
{
  static void Main(string[] args)
  {
```



```
    int[] liste;
    // Eingabe der Array-Größe
    Console.Write("Geben Sie die Anzahl der Elemente ein: ");
    int number = Convert.ToInt32(Console.ReadLine());
    // Initialisierung des Arrays
    liste = new int[number];
    // jedes Element des Arrays in einer Schleife durchlaufen
    // und jedem Array-Element einen Wert zuweisen und danach
    // an der Konsole ausgeben
    for (int i = 0; i < number; i++)
    {
      liste[i] = i * i;
      Console.WriteLine("liste[{0}] = {1}", i, liste[i]);
    }
    Console.ReadLine();
    }
  }
}
```

Listing 2.22 Das Beispielprogramm »ArrayInitialisierung«

Zuerst wird das Array `liste` deklariert, dessen Größe zunächst noch unbestimmt ist. Im nächsten Schritt wird der Anwender zur Angabe der gewünschten Elementanzahl aufgefordert. Die Eingabe wird von der Methode `ReadLine` entgegengenommen und als Rückgabewert vom Typ `string` geliefert. Da wir das Array mit einem Integer initialisieren müssen, muss die Benutzereingabe vor der Zuweisung an die Variable `number` zuerst in den richtigen Typ konvertiert werden. Wir benutzen dazu wieder die Methode `ToInt32` der Klasse `Convert`. Jetzt wissen wir, wie groß das Array `liste` tatsächlich werden soll, und können es mit

```
liste = new int[number];
```

initialisieren.

Thematisch noch nicht behandelt haben wir bisher Schleifen, die Anweisungen wiederholt ausführen. Das soll uns aber in diesem Beispiel nicht davon abhalten, schon einmal einen kurzen Blick auf die `for`-Schleife zu werfen, die solche Anforderungen erfüllt. Die Anzahl der Schleifendurchläufe muss dabei vor dem Eintreten in die Schleife bekannt sein. Auf die Details der Syntax kommen wir in Abschnitt 2.7.1 noch zu sprechen.

In unserem Beispiel wird die Schleife vom ersten Index (= 0) bis zum letzten Index, der erst zur Laufzeit der Anwendung festgelegt wird, durchlaufen. Innerhalb des Anweisungsblocks wird anforderungsgerecht zuerst das Quadrat des Index ermittelt und das Ergebnis dem entsprechenden Array-Element zugewiesen. Anschließend erfolgt die Ausgabe an der Konsole.

Wenn Sie zur Laufzeit auf Aufforderung hin die Zahl 4 eingeben, wird im Fenster der Eingabekonsole die folgende Ausgabe erscheinen:

```
liste[0] = 0
liste[1] = 1
liste[2] = 4
liste[3] = 9
```

2.5.5 Bestimmung der Array-Obergrenze

Es kommt häufig vor, dass Sie zur Laufzeit die Array-Obergrenze ermitteln müssen, bei einem mehrdimensionalen Array vielleicht sogar die Obergrenze einer bestimmten Dimension. Insbesondere bei Arrays, deren Größe ähnlich wie im vorhergehenden Abschnitt gezeigt erst zur Laufzeit festgelegt wird, kommt dieser Fragestellung Bedeutung zu.

Da ein Array ein Objekt ist, können auf dem Array-Bezeichner Methoden aufgerufen werden. Dazu gehört auch die Methode `GetLength`, die uns für jede beliebige Dimension eines vorgegebenen Arrays die Anzahl der Elemente zurückliefert. Auch wenn wir thematisch jetzt ein wenig vorgreifen, sollten wir uns kurz die Definition dieser Methode ansehen:

```
public int GetLength(int dimension)
```

Der Zugriffsmodifizierer `public` interessiert uns an dieser Stelle noch nicht. In einem anderen Zusammenhang werden wir uns mit ihm noch beschäftigen. Die Methode liefert einen `int` als Resultat zurück, gekennzeichnet durch die entsprechende Angabe vor dem Methodenbezeichner. In den runden Klammern ist ebenfalls ein `int` deklariert. Hier erwartet die Methode von uns die Angabe, von welcher Dimension wir die Elementanzahl, also die Größe, erfahren wollen. Dabei gilt, dass die erste Dimension mit 0 angegeben wird, die zweite mit 1 usw.

Haben wir ein zweidimensionales Array mit

```
int[,] elements = new int[20,45];
```

deklariert, wird uns die Anweisung

```
Console.WriteLine(elements.GetLength(1));
```

die Größe der zweiten Dimension ausgeben, also `45`.

2.5.6 Die Gesamtanzahl der Array-Elemente

Liegt ein mehrdimensionales Array vor, können wir die Gesamtanzahl der Elemente ermitteln, indem wir die Methode `GetLength` auf jeder Dimension aufrufen und anschließend die Rückgabewerte multiplizieren – aber es geht auch anders. Die Klasse `Array` bietet mit der Eigenschaft `Length` die Möglichkeit, auf einfache Art und Weise an die gewünschte Information zu gelangen:


```
int[,] elements = new int[20,45];
Console.WriteLine(elements.Length);
```

Die Ausgabe dieses Codefragments wird `900` sein, denn das Array enthält insgesamt 20 × 45 Elemente.

Bei einem eindimensionalen Array wird uns `Length` ebenfalls die Anzahl der Elemente liefern. In Schleifen, die Element für Element durchlaufen werden sollen, benötigen wir jedoch meist den letzten Index des Arrays. Dieser ist um genau eins niedriger als der Wert, der von `Length` zurückgegeben wird, also:

```
letzterArrayIndex = Array-Bezeichner.Length - 1;
```

2.5.7 Verzweigte Arrays

In allen bisherigen Ausführungen hatten unsere Arrays eine rechteckige Struktur. In C# haben Sie aber auch die Möglichkeit, ein Array zu deklarieren, dessen Elemente selbst wieder Arrays sind. Ein solches Array wird als *verzweigtes Array* bezeichnet. Da die Anzahl der Dimensionen eines verzweigten Arrays für jedes Element unterschiedlich groß sein kann, ist ein solches Array äußerst flexibel.

Die Deklaration und Initialisierung eines verzweigten Arrays ist nicht mehr so einfach wie die eines herkömmlichen mehrdimensionalen Arrays. Betrachten wir dazu zunächst ein Beispiel:

```
int[][] myArray = new int[4][];
```

Das Array `myArray` enthält insgesamt vier Elemente, die ihrerseits wieder Arrays sind. Kennzeichnend für verzweigte Arrays ist die doppelte Angabe der rechteckigen Klammern sowohl links vom Gleichheitszeichen bei der Deklaration als auch rechts bei der Initialisierung. Im ersten Moment mag das verwirrend erscheinen, aber vergleichen wir doch einmal: Würden wir ein eindimensionales Array deklarieren und initialisieren, müsste die Anweisung dazu wie folgt lauten:

```
int[] myArray = new int[4];
```

Durch das Hinzufügen einer zweiten Klammer, sowohl im deklarierenden als auch im initialisierenden Teil, machen wir deutlich, dass jedes Array-Element seinerseits ein Array repräsentiert.

Hätten wir es mit einem einfachen Array zu tun, würde es als initialisiert gelten. Nun ist der Sachverhalt aber anders, denn jedes Element eines verzweigten Arrays muss seinerseits selbst initialisiert werden. Bezogen auf das oben deklarierte Array `myArray` könnte das beispielsweise wie folgt aussehen:

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

Wenn die einzelnen Elemente aller Arrays bekannt sind, kann alternativ auch literal mit

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

oder mit

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

initialisiert werden.

Beim Zugriff auf das Element eines verzweigten Arrays muss zuerst berücksichtigt werden, in welchem Unterarray sich das gewünschte Element befindet. Danach wird die Position innerhalb des Unterarrays bekannt gegeben. Angenommen, Sie möchten den Inhalt des fünften Elements im Unterarray mit dem Index 3 auswerten, würde auf dieses Element wie folgt zugegriffen:

```
Console.WriteLine(myArray[3][4]);
```

Verzweigte Arrays sind nicht nur auf eindimensionale Arrays beschränkt, sondern können auch mit mehrdimensionalen kombiniert werden. Benötigen Sie zum Beispiel ein verzweigtes, zweidimensionales Array, müssen Sie das sowohl im Deklarations- als auch im Initialisierungsteil berücksichtigen. In jedem Teil dient die jeweils zweite eckige Klammer zur Angabe der Dimensionsgröße:

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

2.6 Kontrollstrukturen

Es gibt sicherlich kein Programm, das ohne die Steuerung des Programmablaufs zur Laufzeit auskommt. Das Programm muss Entscheidungen treffen, die vom aktuellen Zustand oder von den Benutzereingaben abhängen. Jede Programmiersprache kennt daher Kontrollstrukturen, um den Programmablauf der aktuellen Situation angepasst zu steuern. In diesem Abschnitt werden Sie die Möglichkeiten kennenlernen, die Sie unter C# nutzen können.

2.6.1 Die »if«-Anweisung

Die `if`-Anweisung bietet sich an, wenn bestimmte Programmteile nur beim Auftreten einer bestimmten Bedingung ausgeführt werden sollen. Betrachten wir dazu das folgende Beispiel:

```
static void Main(string[] args)
{
  Console.Write("Geben Sie Ihren Namen ein: ");
  string name = Console.ReadLine();
  if(name == "")
    Console.WriteLine("Haben Sie keinen Namen?");
  else
    Console.WriteLine($"Ihr Name ist \'{name}\'");
  Console.ReadLine();
}
```

Listing 2.23 Einfache »if«-Anweisung

Das Programm fordert den Anwender dazu auf, seinen Namen einzugeben. Die Benutzereingabe wird von der Methode `ReadLine` der Klasse `Console` entgegengenommen und als Rückgabewert des Aufrufs der Variablen `name` zugewiesen. Um sicherzustellen, dass der Anwender überhaupt eine Eingabe vorgenommen hat, die aus mindestens einem Zeichen besteht, wird der Inhalt der Stringvariablen `name` mit

```
if (name == "")
```

überprüft. Wenn `name` einen Leerstring enthält, wird an der Konsole

```
Haben Sie keinen Namen?
```

ausgegeben. Beachten Sie, dass die zu prüfende Bedingung hinter dem Schlüsselwort `if` grundsätzlich immer einen booleschen Wert, also `true` oder `false`, zurückliefert. Hat der Anwender eine Eingabe gemacht, wird die Eingabe mit einem entsprechenden Begleittext an der Konsole ausgegeben.

Das Kernkonstrukt der Überprüfung ist die `if`-Struktur, deren einfachste Variante wie folgt beschrieben wird:

```
if (Bedingung)
{
  // Anweisung1
}
[else
{
  // Anweisung2
}]
```

Die if-Anweisung dient dazu, in Abhängigkeit von der Bedingung entweder Anweisung1 oder Anweisung2 auszuführen. Ist die Bedingung wahr, wird Anweisung1 ausgeführt, ansonsten Anweisung2 hinter dem else-Zweig – falls ein solcher angegeben ist, denn der else-Zweig ist optional. Bei Anweisung1 und Anweisung2 kann es sich natürlich auch um beliebig viele Anweisungen handeln.

Hinweis

Hinter if und else können Sie auf die geschweiften Klammen verzichten, wenn es sich um genau eine Anweisung handelt. Sind es mehrere Anweisungen, sind die geschweiften Klammern Pflicht.

Beachten Sie, dass es sich bei der Bedingung in jedem Fall um einen booleschen Ausdruck handelt. Diese Anmerkung ist wichtig, denn wenn Sie bereits mit einer anderen Programmiersprache wie beispielsweise C/C++ gearbeitet haben, werden Sie wahrscheinlich zum Testen einer Bedingung einen von 0 verschiedenen Wert benutzt haben. In C# funktioniert das nicht! Nehmen wir an, Sie möchten feststellen, ob eine Zeichenfolge leer ist, dann müssten Sie die Bedingung wie folgt definieren:

```
// Deklaration und Initialisierung der Variablen text
string text = "";
[...]
if(text.Length != 0)
  Console.Write("Inhalt der Variablen = {0}", text);
```

Length liefert die Anzahl der Zeichen der Zeichenfolge zurück.

Da es in C# keine Standardkonvertierung von einem int in einen bool gibt, wäre es falsch, die Bedingung folgendermaßen zu formulieren:

```
// Achtung: In C# nicht zulässig
if (text.Length)...
```

In einer if-Bedingung können Sie beliebige Vergleichsoperatoren einsetzen, auch in Kombination mit den logischen Operatoren. Das kann zu verhältnismäßig komplexen Ausdrücken führen, beispielsweise:

```
if (a <= b && c != 0)...
if ((a > b && c < d)||(e != f && g < h))...
```

Bisher sind wir vereinfachend davon ausgegangen, dass unter einer bestimmten Bedingung immer nur eine Anweisung ausgeführt wird. Meistens müssen jedoch mehrere Anweisungen abgearbeitet werden. Um mehrere Anweisungen beim Auftreten einer bestimmten Bedingung auszuführen, müssen sie lediglich in einen Anweisungsblock zusammengefasst werden, beispielsweise:


```
static void Main(string[] args)
{
  Console.Write("Geben Sie eine Zahl zwischen 0 und 9 ein: ");
  int zahl = Convert.ToInt32(Console.ReadLine());
  if(zahl > 9 || zahl < 0)
  {
    Console.WriteLine("Ihre Zahl ist unzulässig");
    Console.Write("Versuchen Sie es erneut: ");
    zahl = Convert.ToInt32(Console.ReadLine());
  }
  else
  {
    Console.WriteLine("Korrekte Eingabe.");
    Console.WriteLine("Sie beherrschen das Zahlensystem!");
  }
  Console.WriteLine("Die Eingabe lautet:{0}", zahl);
  Console.ReadLine();
}
```

Listing 2.24 Mehrere Anweisungen zusammengefasst in einem Anweisungsblock

Eingebettete »if«-Statements

if-Anweisungen dürfen ineinander verschachtelt werden, d. h., dass innerhalb eines äußeren if-Statements eine oder auch mehrere weitere if-Anweisungen eingebettet werden können. Damit stehen wir aber zunächst vor einem Problem, wie im folgenden Codefragment gezeigt wird:

```
Console.Write("Geben Sie eine Zahl zwischen 0 und 9 ein: ");
int zahl = Convert.ToInt32(Console.ReadLine());
if(zahl >= 0 && zahl <= 9)
if(zahl <= 5)
Console.Write("Die Zahl ist 0,1,2,3,4 oder 5");
else
Console.Write("Die Zahl ist unzulässig.");
```

Listing 2.25 Eingebettetes »if«-Statement

Um die ganze Problematik anschaulich darzustellen, wurde auf sämtliche Tabulatoreinzüge verzichtet, denn Einzüge dienen nur der besseren Lesbarkeit des Programmcodes und haben keinen Einfluss auf die Interpretation der Ausführungsreihenfolge.

Die Frage, die aufgeworfen wird, lautet, ob else zum inneren oder zum äußeren if-Statement gehört. Wenn wir den Code betrachten, sind wir möglicherweise geneigt zu vermuten, else mit der Meldung

```
Die Zahl ist unzulässig.
```

dem äußeren `if` zuzuordnen, wenn eine Zahl kleiner 0 oder größer 9 eingegeben wird. Tatsächlich werden wir mit dieser Meldung genau dann konfrontiert, wenn eine Zahl zwischen 6 und 9 eingeben wird, denn der Compiler interpretiert den Code wie folgt:

```
if(zahl >= 0 && zahl <= 9)
{
  if(zahl <= 5)
    Console.Write("Die Zahl ist 0,1,2,3,4 oder 5");
  else
    Console.Write("Die Zahl ist unzulässig.");
}
```

Listing 2.26 Listing 2.25 nun mit Tabulatoreinzügen

Das war natürlich nicht unsere Absicht, denn rein logisch soll die `else`-Klausel der äußeren Bedingungsprüfung zugeordnet werden. Um das zu erreichen, müssen wir in unserem Programmcode das innere `if`-Statement als Block festlegen:

```
if(zahl >= 0 && zahl <= 9)
{
  if(zahl <= 5)
    Console.Write("Die Zahl ist 0,1,2,3,4 oder 5");
}
else
  Console.Write("Die Zahl ist unzulässig.");
```

Listing 2.27 Richtige Zuordnung des »else«-Zweigs

Unsere Erkenntnis können wir auch in eine allgemeingültige Regel formulieren:

> **Merke**
>
> Eine **else**-Klausel wird immer an das am nächsten stehende `if` gebunden. Dies kann nur durch das Festlegen von Anweisungsblöcken umgangen werden.

Das eben geschilderte Problem der `else`-Zuordnung ist unter dem Begriff *dangling else* bekannt, zu Deutsch »baumelndes else«. Es führt zu logischen Fehlern, die nur sehr schwer aufzuspüren sind.

Es kommt in der Praxis sehr häufig vor, dass mehrere Bedingungen der Reihe nach ausgewertet werden müssen. Unter Einbeziehung der Regel über die Zuordnung der `else`-Klausel könnte eine differenzierte Auswertung einer eingegebenen Zahl beispielsweise wie folgt lauten:


```
Console.Write("Geben Sie eine Zahl zwischen 0 und 9 ein: ");
int zahl = Convert.ToInt32(Console.ReadLine());
if(zahl == 0)
   Console.WriteLine("Die Zahl ist 0");
else
   if(zahl == 1)
      Console.WriteLine("Die Zahl ist 1");
   else
      if(zahl == 2)
         Console.WriteLine("Die Zahl ist 2");
      else
         if(zahl == 3)
            Console.WriteLine("Die Zahl ist 3");
         else
            Console.WriteLine("Zahl > 3");
```

Listing 2.28 Komplexeres »if«-Statement (1)

Um jedes `else` eindeutig zuordnen zu können, weist dieses Codefragment entsprechende Einzüge auf, die keinen Zweifel aufkommen lassen. Das täuscht dennoch nicht darüber hinweg, dass die Lesbarkeit leidet und der Code mit wachsender Anzahl der zu testenden Bedingungen unübersichtlich wird. Unter C# bietet es sich daher an, im Anschluss an das Schlüsselwort `else` sofort ein `if` anzugeben, wie im folgenden identischen Codefragment, das wesentlich überschaubarer wirkt und damit auch besser lesbar ist:

```
if(zahl == 0)
   Console.WriteLine("Die Zahl ist 0");
else if(zahl == 1)
   Console.WriteLine("Die Zahl ist 1");
else if(zahl == 2)
   Console.WriteLine("Die Zahl ist 2");
else if(zahl == 3)
   Console.WriteLine("Die Zahl ist 3");
else
   Console.WriteLine("Zahl > 3");
```

Listing 2.29 Komplexeres »if«-Statement (2)

Bedingte Zuweisung mit dem »?:«-Operator

Manchmal sehen wir uns mit der Aufgabe konfrontiert, eine Bedingung nur auf ihren booleschen Wert hin zu prüfen und in Abhängigkeit vom Testergebnis eine Zuweisung auszuführen. Eine `if`-Anweisung könnte dazu wie nachfolgend gezeigt aussehen:

```
int x, y;
Console.Write("Geben Sie eine Zahl ein: ");
x = Convert.ToInt32(Console.ReadLine());
if(x == 0)
   y = 1;
else
   y = x;
```

Gibt der Anwender die Zahl 0 ein, wird der Variablen `y` der Wert 1 zugewiesen. Weicht die Eingabe von 0 ab, ist der Inhalt der Variablen `x` mit der Variablen `y` identisch.

Es kann auch ein von C# angebotener spezieller Bedingungsoperator eingesetzt werden. Sehen wir uns zunächst dessen Syntax an:

<Variable> = <Bedingung> ? <Wert1> : <Wert2>

Zuerst wird die Bedingung ausgewertet. Ist deren Ergebnis `true`, wird `Wert1` der Variablen zugewiesen, andernfalls `Wert2`. Damit können wir das Beispiel von oben vollkommen äquivalent auch anders implementieren:

```
int x, y;
Console.Write("Geben Sie eine Zahl ein: ");
x = Convert.ToInt32(Console.ReadLine());
y = x == 0 ? 1 : x;
```

Im ersten Moment sieht der Code schlecht lesbar aus. Wenn wir allerdings zusätzliche Klammern setzen, wird die entsprechende Codezeile schon deutlicher:

```
y = (x == 0 ? 1 : x);
```

Zuerst wird die Bedingung

```
x == 0
```

geprüft. Ist das Ergebnis `true`, wird `y` die Zahl 1 zugewiesen. Ist das Ergebnis `false`, werden die beiden Variablen gleichgesetzt.

2.6.2 Das »switch«-Statement

Mit der `if`-Anweisung können Bedingungen auf Basis sowohl verschiedener Vergleichsoperatoren als auch verschiedener Operanden formuliert werden. In der Praxis muss jedoch häufig derselbe Operand überprüft werden. Nehmen wir beispielsweise an, eine Konsolenanwendung bietet dem Anwender eine Auswahl diverser Optionen an, die den weiteren Ablauf des Programms steuert:

```
static void Main(string[] args)
{
  string message = "Treffen Sie eine Wahl:\n\n";
```



```
  message += "(N) - Neues Spiel\n";
  message += "(A) - Altes Spiel fortsetzen\n";
  message += "(E) - Beenden\n";
  Console.WriteLine(message);
  Console.Write("Ihre Wahl lautet: ");
  string choice = Console.ReadLine().ToUpper();
  if (choice == "N")
  {
    Console.Write("Neues Spiel...");
    // Anweisungen, die ein neues Spiel starten
  }
  else if (choice == "A")
  {
    Console.Write("Altes Spiel laden ...");
    // Anweisungen, die einen alten Spielstand abrufen
  }
  else if (choice == "E")
  {
    Console.Write("Spiel beenden ...");
    // Anweisungen, um das Spiel zu beenden
  }
  else
  {
    Console.Write("Ungültige Eingabe ...");
    // weitere Anweisungen
  }
  Console.ReadLine();
}
```

Listing 2.30 Komplexe Bedingungsprüfung

Der Ablauf des Programms wird über die Eingabe »N«, »A« oder »E« festgelegt. Stellvertretend wird in unserem Fall dazu eine Konsolenausgabe angezeigt. Vor der Eingabeüberprüfung sollten wir berücksichtigen, dass der Anwender möglicherweise der geforderten Großschreibweise der Buchstaben keine Beachtung schenkt. Um diesem Umstand Rechnung zu tragen, wird die Eingabe mit

```
string choice = Console.ReadLine().ToUpper();
```

in jedem Fall in einen Großbuchstaben umgewandelt. Verantwortlich dafür ist die Methode `ToUpper` der Klasse `String`, die direkt auf dem Rückgabewert aufgerufen wird.

Alternativ zur `if`-Struktur könnte die Programmlogik auch mit einer `switch`-Anweisung realisiert werden. Im obigen Beispiel müsste der `if`-Programmteil dann durch den folgenden ersetzt werden:

```
// Beispiel: ..\Kapitel 2\SwitchSample
[...]
switch (choice)
{
  case "N":
    Console.Write("Neues Spiel...");
    // Anweisungen, die ein neues Spiel starten
    break;
  case "A":
    Console.Write("Altes Spiel laden...");
    // Anweisungen, die einen alten Spielstand laden
    break;
  case "E":
    Console.Write("Spiel beenden...");
    // Anweisungen, um das Spiel zu beenden
    break;
  default:
    Console.Write("Ungültige Eingabe...");
    // weitere Anweisungen
    break;
}
[...]
```

Listing 2.31 Das »switch«-Statement

Sehen wir uns nun die allgemeine Syntax der `switch`-Anweisung an:

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

Mit der `switch`-Anweisung lässt sich der Programmablauf ähnlich wie mit der `if`-Anweisung steuern. Dabei wird überprüft, ob der hinter `switch` aufgeführte Ausdruck, der im Regelfall

entweder eine Ganzzahl oder eine Zeichenfolge sein muss, mit einer der hinter `case` angegebenen Konstanten übereinstimmt. Nacheinander wird dabei zuerst mit `Konstante1` verglichen, danach mit `Konstante2` usw. Stimmen Ausdruck und Konstante überein, werden alle folgenden Anweisungen bis zur Sprunganweisung `break` ausgeführt. Wird zwischen dem Ausdruck und einer der Konstanten keine Übereinstimmung festgestellt, werden die Anweisungen hinter der `default`-Marke ausgeführt – falls eine solche angegeben ist, denn `default` ist optional. Achten Sie auch darauf, hinter jeder Konstanten und hinter `default` einen Doppelpunkt zu setzen.

Anmerkung

Tatsächlich muss es sich bei dem hinter `switch` angegeben Ausdruck seit der Version C# 7.0 nicht mehr nur um eine Ganzzahl oder einen String handeln. Mit der Einführung des *Pattern Matchings* wurde diese Regel gebrochen und erweitert. Allerdings werde ich erst in Kapitel 10, »Weitere C#-Sprachfeatures«, auf das Pattern Matching eingehen, da einige damit im Zusammenhang stehende Sprachgrundlagen an dieser Stelle noch nicht bekannt sind.

Die Sprunganweisung `break` ist in jedem Fall erforderlich, wenn hinter dem `case`-Statement eine oder mehrere Anweisungen codiert sind, ansonsten meldet der Compiler einen Syntaxfehler. Die `break`-Anweisung signalisiert, die Programmausführung mit der Anweisung fortzusetzen, die dem `switch`-Anweisungsblock folgt.

Auf `break` kann man verzichten, wenn mehrere `case`-Anweisungen direkt hintereinanderstehen. Die Folge ist dann, dass die Kette so lange durchlaufen wird, bis ein `break` erscheint. Daher wird im folgenden Codefragment die erste Ausgabeanweisung ausgeführt, wenn `value` den Wert 1, 2 oder 3 hat.

```
int value = ...;
switch(value)
{
  case 1:
  case 2:
  case 3:
    Console.Write("value = 1, 2 oder 3");
    break;
  case 4:
    Console.Write("value = 4");
    break;
}
```

Neben `break` gibt es mit `goto` eine weitere Sprunganweisung, hinter der eine Marke angegeben werden kann, beispielsweise:

```
goto case "E";
```

Die goto-Anweisung bietet sich insbesondere an, wenn für mehrere Konstanten dieselben Anweisungsfolgen ausgeführt werden müssen, z. B.:

```
int value = ...;
switch(value)
{
  case 1:
    Console.WriteLine("Im case 1-Zweig");
    goto case 3;
  case 2:
  case 3:
    Console.Write("value = 1, 2 oder 3");
    break;
  case 4:
    Console.Write("value = 4");
    break;
}
```

Nehmen wir an, value hätte den Wert 1. Das Programm reagiert wie folgt: Zuerst wird der case 1-Zweig ausgeführt und danach die Steuerung des Programms an den case 3-Zweig übergeben. Zwei Konsolenausgaben sind also die Folge:

```
Im case 1-Zweig
value = 1, 2 oder 3
```

Einschränkungen der »switch«-Anweisung

In C# gibt es keine Möglichkeit, einen zusammenhängenden Konstantenbereich hinter dem case-Statement anzugeben, wie es in einigen anderen Sprachen möglich ist. Wollen Sie beispielsweise für einen Ausdruck alle Zahlen im Bereich von 0 bis 10 gleichermaßen behandeln, müssen Sie für jede einzelne eine case-Anweisung implementieren. In solchen Fällen empfiehlt es sich, anstelle der switch-Anweisung das if-Statement zu verwenden.

Die »goto«-Anweisung

Die goto-Anweisung kann nicht nur innerhalb eines switch-Blocks angegeben, sondern auch generell dazu benutzt werden, eine beliebige Marke im Code anzusteuern. Solche Sprünge werden auch als *unbedingte Sprünge* bezeichnet, weil sie an keine besondere Bedingung geknüpft sind. Eine Marke ist ein Bezeichner, der mit einem Doppelpunkt abgeschlossen wird. Im folgenden Beispiel wird die Marke meineMarke definiert. Trifft das Programm zur Laufzeit auf das goto-Statement, verzweigt es zu den Anweisungen, die sich hinter der benutzerdefinierten Marke befinden.


```
static void Main(string[] args)
{
  int value = 4711;
  Console.WriteLine("Programmstart");
  goto meineMarke;
  Console.WriteLine("value = {0}",value);
meineMarke:
  Console.WriteLine("Programmende");
  Console.ReadLine();
}
```

Listing 2.32 Allgemeine Verwendung von »goto«

In diesem Listing wird es niemals zu der Ausgabe des Variableninhalts von value kommen. Das ist natürlich kein Fehler, sondern mehr eine programmiertechnische Unsauberkeit, die der Compiler sogar erkennt und im Fenster *Fehlerliste* als Warnhinweis anzeigt.

Neben der Möglichkeit, eine Sprunganweisung innerhalb einer switch-Anweisung zu codieren, bietet sich die goto-Anweisung auch dazu an, tief verschachtelte Schleifen zu verlassen (mehr dazu im folgenden Abschnitt). In allen anderen Fällen sollten Sie jedoch prinzipiell auf goto verzichten, denn es zeugt im Allgemeinen von einem schlechten Programmierstil.

2.7 Programmschleifen

Schleifen dienen dazu, Anweisungsfolgen wiederholt auszuführen. Dabei wird zwischen zwei Schleifentypen unterschieden:

- bestimmte Schleifen
- unbestimmte Schleifen

Ist beim Schleifeneintritt bekannt, wie oft die Anweisungsfolge durchlaufen werden muss, wird von einer *bestimmten Schleife* gesprochen. Ergibt sich erst während des Schleifendurchlaufs, wann die zyklische Bearbeitung abgebrochen werden kann oder muss, spricht man von *unbestimmten Schleifen*. Die Grenzen zwischen diesen beiden Typen sind dabei nicht eindeutig, sondern können durchaus verwischen. Eine bestimmte Schleife kann wie eine unbestimmte agieren, eine unbestimmte wie eine bestimmte.

2.7.1 Die »for«-Schleife

Man setzt eine for-Schleife meistens dann ein, wenn bekannt ist, wie oft bestimmte Anweisungen ausgeführt werden müssen. Die allgemeine Syntax des for-Schleifenkonstrukts sieht dabei wie folgt aus:

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

Die `for`-Schleife setzt sich aus zwei Komponenten zusammen: aus dem Schleifenkopf, der die Eigenschaft der Schleife beschreibt, und aus dem sich daran anschließenden Schleifenblock in geschweiften Klammern, der die wiederholt auszuführenden Anweisungen enthält. Handelt es sich dabei nur um eine Anweisung, können Sie auf die geschweiften Klammern verzichten.

Um die Anzahl der Durchläufe einer `for`-Schleife festzulegen, bedarf es eines Schleifenzählers, dessen Anfangswert durch *`Ausdruck1`* beschrieben wird. Der Endwert wird in *`Ausdruck2`* festgelegt, und *`Ausdruck3`* schließlich bestimmt, auf welchen Betrag der Schleifenzähler bei jedem Schleifendurchlauf erhöht werden soll. Dazu ein Beispiel:

```
for(int counter = 0; counter < 10; counter++)
{
  Console.WriteLine($"Zählerstand = {counter}");
}
```

Listing 2.33 Konstruktion einer einfachen »for«-Schleife

Der Schleifenzähler heißt hier `counter`. Sein Startwert beträgt 0, und er wird bei jedem Schleifendurchlauf um den Wert 1 erhöht. Erreicht `counter` den Wert 10, wird das Programm mit der Anweisung fortgesetzt, die dem Anweisungsblock der Schleife folgt.

Hinweis

Genauso wie bei einer `if`-Anweisung gilt auch für eine `for`-Schleife, dass Sie auf die geschweiften Klammern des Anweisungsblocks verzichten können, wenn die Operation in der Schleife nur durch eine einzige Anweisung beschrieben wird. Daher dürfte die `for`-Schleife aus Listing 2.33 auch wie folgt codiert werden:

```
for(int counter = 0; counter < 10; counter++)
  Console.WriteLine($"Zählerstand = {counter}");
```

Führen wir den Code aus, werden wir an der Konsole die folgende Ausgabe erhalten:

```
Zählerstand = 0
Zählerstand = 1
Zählerstand = 2
Zählerstand = 3
```



```
[...]
Zählerstand = 8
Zählerstand = 9
```

Weil der Schleifenblock nur eine Anweisung enthält, könnte die `for`-Schleife auch wie folgt codiert werden:

```
for(int counter = 0; counter < 10; counter++)
  Console.WriteLine("Zählerstand = {0}",counter);
```

Die Arbeitsweise der »for«-Schleife

Stößt der Programmablauf auf eine `for`-Schleife, wird zuerst `Ausdruck1` – auch *Initialisierungsausdruck* genannt – ausgewertet. Dieser initialisiert den Zähler der Schleife mit einem Startwert. Der Zähler der Schleife in unserem Beispiel wird mit dem Startwert 0 initialisiert.

`Ausdruck2`, der *Bedingungsausdruck*, wertet vor jedem Schleifendurchlauf den aktuellen Stand des Zählers aus. Im Beispiel von oben lautet die Bedingung:

```
counter < 10
```

Der Bedingungsausdruck kann unter Einbeziehung der diversen Operatoren beliebig komplex werden, muss aber immer ein boolesches Ergebnis haben. Der Anweisungsblock wird nur dann ausgeführt, wenn `Ausdruck2` `true` ist, ansonsten setzt das Programm seine Ausführung mit der Anweisung fort, die dem Schleifenblock folgt.

`Ausdruck3` (*Reinitialisierungsausdruck*) übernimmt die Steuerung des Schleifenzählers. Er wird dazu benutzt, den Schleifenzähler entweder zu inkrementieren oder zu dekrementieren. In unserem Fall wird der Zähler jeweils um +1 erhöht. Die Erhöhung erfolgt immer dann, wenn der Anweisungsblock der Schleife durchlaufen ist. Danach bewertet der Bedingungsausdruck den neuen Zählerstand.

Die Zählervariable

Grundsätzlich gibt es zwei Möglichkeiten, die Zählervariable zu deklarieren, die für das Abbruchkriterium herangezogen wird:

- innerhalb des Schleifenkopfs
- vor der Schleife

Welcher Notation Sie den Vorzug geben, hängt davon ab, über welche Sichtbarkeit der Zähler verfügen soll. Betrachten Sie dazu zunächst das folgende Codefragment:

```
static void Main(string[] args)
{
  for (int index = 0; index <= 10; index++)
  {
    Console.WriteLine("Zählerstand = {0}", index);
```

```
  }
  // die folgende Anweisung verursacht einen Kompilierfehler
  Console.WriteLine(index);
}
```

Listing 2.34 Deklaration der Zählervariablen im Schleifenkopf

Eine Zählervariable, die im Schleifenkopf deklariert wird, gilt als lokale Variable der Schleife und ist deshalb auch nur innerhalb des Anweisungsblocks der `for`-Schleife gültig. Der Zugriff auf den Zähler von außerhalb der Schleife führt deshalb auch zu einem Kompilierfehler.

Implementieren Sie mehrere Schleifen, müssen Sie daher auch jedes Mal den Zähler neu deklarieren:

```
for(int index = 0; index <= 10 ;index++) { [...] }
[...]
for(int index = 12; index <= 100 ;index += 3) { [...] }
```

Die bessere Lösung wäre in diesem Fall die Deklaration der Zählervariablen vor dem Auftreten der ersten Schleife:

```
int index;
for(index = 0; index <= 10 ;index++) { [...] }
[...]
for(index = 12; index <= 100 ;index += 3){ [...] }
```

Wenn wir an diesem Punkt angekommen sind, stellt sich die Frage, ob beim Vorliegen einer einzigen `for`-Schleife die gleichzeitige Deklaration und Initialisierung im Schleifenkopf der vorgezogenen Deklaration der Zählervariablen vor dem Schleifenkopf vorzuziehen ist. Eine klare Antwort darauf gibt es nicht. Der besseren Übersichtlichkeit wegen scheint es jedoch vorteilhaft zu sein, die Deklaration im Initialisierungsausdruck vorzunehmen.

»for«-Schleifen mit beliebiger Veränderung des Zählers

In den meisten Fällen erfüllt eine ganzzahlige Schrittweite die Anforderungen vollkommen. Das ist aber nicht immer so. Manchmal werden auch kleinere Schrittweiten benötigt, also im Bereich von Fließkommazahlen. Fließkommazahlen sind naturgemäß systembedingt immer ungenau. Das kann bei Schleifen besonders fatale Folgen haben. Sehen Sie sich dazu Listing 2.35 an:

```
static void Main(string[] args)
{
  int value = 0;
  for(double counter = 0; counter <= 2 ;counter += 0.1)
  {
    value++;
```



```
    Console.WriteLine($"{value}. Zählerstand = {counter}");
  }
  Console.ReadLine();
}
```

Listing 2.35 »for«-Schleife mit Zähler vom Typ einer Dezimalzahl

Normalerweise würden wir auf den ersten Blick keinen Haken vermuten – erst wenn wir das Programm ausführen, werden wir feststellen, dass der letzte Zählerwert fehlt:

```
1. Zählerstand = 0
2. Zählerstand = 0,1
[...]
18. Zählerstand = 1,7
19. Zählerstand = 1,8
20. Zählerstand = 1,9
```

Die systembedingte Ungenauigkeit der Fließkommazahlen bewirkt, dass der Zählerstand im letzten Schritt nicht exakt 2 ist, sondern ein wenig größer. Damit wird der zweite Ausdruck des Schleifenkopfs zu `false` und bewirkt den vorzeitigen Ausstieg aus der Schleife – der letzte erforderliche Schleifendurchlauf wird überhaupt nicht ausgeführt. Diese These lässt sich beweisen, wenn wir die Anweisung zur Ausgabe an der Konsole durch die folgende ersetzen:

```
Console.WriteLine($"{value}. Zählerstand = {counter:E16}");
```

Wir erzwingen nun die Ausgabe in Exponentialschreibweise und geben eine Genauigkeit von 16 Nachkommastellen an – denn der Typ `double` wird an der 16. Nachkommastelle ungenau. Die Ausgabe an der Konsole sieht dann wie in Abbildung 2.10 gezeigt aus.

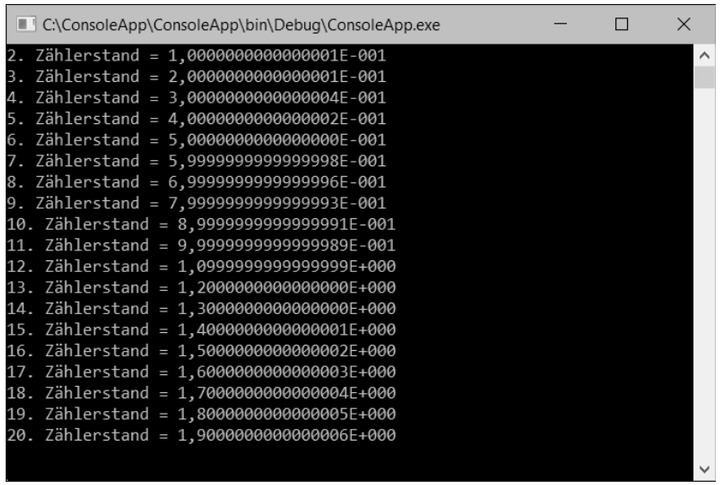

Abbildung 2.10 Fließkommazahl als Zähler – die Ausgabe an der Konsole

Diesen Fehler können Sie vermeiden, indem Sie sowohl den Zähler als auch die Schrittweite ganzzahlig machen. In unserem Beispiel wird mit dem Faktor 10 die Schrittweite auf +1 gesetzt. Analog muss die Ausstiegsbedingung angepasst werden. Um den Effekt bei der Ausgabe rückgängig zu machen, dividieren wir das auszugebende Datum am Ende durch denselben Faktor.

```
static void Main(string[] args)
{
  int value = 0;
  for(double counter = 0; counter <= 20 ;counter++)
  {
    value++;
    Console.WriteLine($"{value}. Zählerstand = {counter/10}");
  }
  Console.ReadLine();
}
```

Listing 2.36 Anpassung des Codes aus Listing 2.35 an Ganzzahlen

Natürlich bewirkt die Division ihrerseits auch wieder eine Ungenauigkeit, aber das liegt in der Natur der Fließkommazahlen, was wir akzeptieren müssen. Andererseits haben wir aber die Gewissheit, dass zumindest die Anzahl der Schleifendurchläufe korrekt ist.

Die Initialisierung von Arrays in einer »for«-Schleife

Sie haben gesehen, dass mit `for`-Schleifen Anweisungssequenzen wiederholt ausgeführt werden. Dieser Schleifentyp eignet sich besonders dazu, Array-Elemente mit bestimmten Werten zu initialisieren. Machen wir uns das an einem einfachen Beispiel deutlich. Das Array `liste` soll mit Zahlen initialisiert werden, die dem Quadrat des Index des Elements entsprechen. Den höchsten vertretenen Index soll der Anwender an der Konsole eingeben. Der Code dazu sieht wie folgt aus:

```
static void Main(string[] args)
{
  int[] liste;
  Console.Write("Geben Sie den höchsten Array-Index ein: ");
  liste = new int[Convert.ToInt32(Console.ReadLine()) + 1];
  for(int i = 0; i < liste.Length; i++)
  {
    liste[i] = i * i;
    Console.WriteLine(liste[i]);
  }
```



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

Listing 2.37 Ein Array in einer Schleife initialisieren

Nach der Deklaration des Arrays und der sich anschließenden Aufforderung, die Größe des Arrays festzulegen, wird das Array entsprechend der Eingabe des Anwenders initialisiert. Die Anweisung dazu erscheint im ersten Moment verhältnismäßig komplex, ist aber recht einfach zu interpretieren. Dabei geht man – genauso wie es auch die Laufzeit macht – von der innersten Klammerebene aus, im vorliegenden Fall also von der Entgegennahme der Benutzereingabe:

```
Console.ReadLine()
```

Die Eingabe des Anwenders ist eine Zeichenfolge, also vom Typ `string`. Da die Indexangabe eines Arrays immer ein `int` sein muss, sind wir zu einer Konvertierung gezwungen:

```
Convert.ToInt32(Console.ReadLine())
```

Jetzt gilt es noch zu bedenken, dass per Vorgabe die Eingabe den höchsten Index des Arrays darstellt, wir aber bei einer Array-Initialisierung immer die Anzahl der Elemente angeben. Um unser Array endgültig richtig zu dimensionieren, müssen wir die konvertierte Benutzereingabe noch um 1 erhöhen, also:

```
Convert.ToInt32(Console.ReadLine()) + 1
```

Mit der daraus resultierenden Zahl kann das Array nun endgültig in der vom Anwender gewünschten Kapazität initialisiert werden.

Jetzt folgt die `for`-Schleife. Da wir jedem Array-Element im Schleifenblock das Quadrat seines Index zuweisen wollen, lassen wir den Schleifenzähler über alle vertretenen Indizes laufen – also von 0 bis zum höchsten Index. Letzteren ermitteln wir aus der Eigenschaft `Length` unseres Arrays, die uns die Gesamtanzahl der Elemente liefert. Diese ist immer um 1 höher als der letzte Index im Array. Daher entspricht die Bedingung

```
i < liste.Length
```

immer den Forderungen, denn die Schleife wird jetzt so lange durchlaufen, bis die Zahl erreicht ist, die kleiner ist als die Anzahl der Elemente. Gleichwertig könnten wir auch Folgendes formulieren:

```
i <= liste.Length - 1
```

Der Schleifenkopf ist nun anforderungsgerecht formuliert, die Anweisungen des Schleifenblocks werden genauso oft durchlaufen, wie das Array Elemente aufweist. Da bei jedem Schleifendurchlauf der Schleifenzähler ein Pendant in Form eines Array-Index aufweist, können wir den Zähler dazu benutzen, jedes einzelne Array-Element anzusprechen:

```
liste[i] = i * i;
```

Beim ersten Durchlauf mit `i = 0` wird demnach `liste[0]` die Zahl 0 zugewiesen, beim zweiten Durchlauf mit `i = 1` dem Element `liste[1]` der Wert 1 usw.

Die Argumente der »Main«-Prozedur

Bisher haben wir unsere Programme immer nur durch einen einfachen Aufruf gestartet, entweder direkt aus der Entwicklungsumgebung heraus oder durch die Angabe des Dateinamens an der Eingabekonsole. Verteilen wir eine Anwendung, wird ein Anwender jedoch niemals aus der Entwicklungsumgebung heraus die Applikation starten, sondern entweder durch Doppelklick auf die EXE-Datei im Explorer, durch die Eingabe des Namens der ausführbaren Datei an der Eingabekonsole oder über die Option START • AUSFÜHREN...

Die beiden letztgenannten Punkte eröffnen noch weitere Möglichkeiten: Es können der `Main`-Methode auch Befehlszeilenparameter als zusätzliche Informationen übergeben werden, die im Array `args` der Parameterliste der `Main`-Methode entgegengenommen werden:

```
static void Main(string[] args)
```

Nehmen wir an, wir würden eine Anwendung namens *MyApplication.exe* an der Konsole wie folgt starten:

```
MyApplication Peter Willi Udo
```

Die drei Übergabeparameter `Peter`, `Willi` und `Udo` werden von `Main` im `string`-Array `args` empfangen und können von der Anwendung für weitere Operationen benutzt werden. Da das Programm zur Laufzeit jedoch nicht weiß, ob und wie viele Parameter übergeben worden sind, wird das Array `args` zunächst dahingehend abgefragt, ob überhaupt ein gültiges Element enthalten ist. Wenn die Anzahl der Elemente größer 0 ist, können Sie mit einer `for`-Schleife in bekannter Weise auf jedes Array-Element zugreifen. Sehen wir uns das an einem konkreten Beispiel an:

```
// Beispiel: ..\Kapitel 2\Befehlszeilenparameter
static void Main(string[] args)
{
  // prüfen, ob beim Programmaufruf eine oder mehere Strings übergeben worden sind
  if (args.Length > 0)
  {
    // die Zeichenfolgen in der Konsole anzeigen
    for (int i = 0; i < args.Length; i++)
      Console.WriteLine(args[i]);
  }
  else
    Console.WriteLine("Kein Übergabestring");
```



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

Listing 2.38 Auswerten der Übergabeargumente an die Methode »Main«

Das `if`-Statement stellt durch Auswertung der `Length`-Eigenschaft auf `args` fest, ob das Array leer ist oder nicht. Hat der Anwender zumindest einen Parameter übergeben, wird die `for`-Schleife ausgeführt, die den Inhalt des Parameters an der Konsole ausgibt.

Grundsätzlich werden alle übergebenen Parameter als Zeichenfolgen empfangen. Das soll uns aber nicht davon abhalten, im Bedarfsfall der Laufzeitumgebung auch Zahlen zu übergeben. Allerdings dürfen wir dann nicht vergessen, mit einer der Methoden der Klasse `Convert` die Zeichenfolge in den erforderlichen Datentyp zu konvertieren.

Verschachtelte Schleifen

`for`-Schleifen können praktisch beliebig verschachtelt werden. Im nächsten Beispiel werde ich zeigen, wie Sie eine verschachtelte Schleife dazu benutzen können, einen Baum beliebiger Größe – hier durch Buchstaben dargestellt – an der Konsole auszugeben.

```
1:      M
2:     MMM
3:    MMMMM
4:   MMMMMMM
5:  MMMMMMMMM
6: MMMMMMMMMMM
```

Jede Ausgabezeile setzt sich aus einer Anzahl von Leerzeichen und Buchstaben zusammen und hängt von der Größe der Darstellung ab. Für die Leerzeichen gilt:

Anzahl Leerzeichen = Gesamtanzahl der Zeilen – aktuelle Zeilennummer

Die auszugebenden Buchstaben folgen der Beziehung:

*Anzahl der Buchstaben = aktuelle Zeilennummer * 2 – 1*

Um die gewünschte Ausgabe zu erhalten, wird in einer äußeren `for`-Schleife jede Stufe (Zeile) des Baums separat behandelt. Darin eingebettet sind zwei weitere Schleifen implementiert, von denen jede für sich zuerst vollständig ausgeführt wird – wir haben es also mit zwei parallelen inneren Schleifen zu tun. Dabei werden in der ersten inneren Schleife zuerst die Leerzeichen geschrieben und in der zweiten die Buchstaben. Die Struktur der Schleifen sieht demnach wie folgt aus:

```
// Äußere Schleife beschreibt bei jedem Durchlauf eine Zeile
for (...)
{
```

```
  // Leerzeichen schreiben
  for (...) { [...] }
  // Buchstaben schreiben
  for (...) { [...] }
}
```

Sehen wir uns nun den Programmcode an, der den gestellten Anforderungen genügt. Das Programm verlangt nach dem Start, dass der Anwender die Anzahl der Stufen angibt.

```
// Beispiel: ..\Kapitel 2\Baumstruktur
static void Main(string[] args)
{
  Console.Write("Geben Sie die Anzahl der Stufen ein: ");
  int zeile = Convert.ToInt32(Console.ReadLine());
    // jede Stufe des Baums aufbauen
    for (int i = 1; i <= zeile; i++)
    {
      // Leerzeichen schreiben
      for (int j = 1; j <= zeile - i; j++)
        Console.Write(" ");
      // Buchstaben schreiben
      for (int j = 1; j <= i * 2 - 1; j++)
        Console.Write("M");
      Console.WriteLine();
    }
  }
  Console.ReadLine();
}
```

Listing 2.39 Verschachtelte »for«-Schleifen

Vorzeitiges Beenden einer Schleife mit »break«

Es kann sich zur Laufzeit als erforderlich erweisen, nicht auf das Erfüllen der Abbruchbedingung zu warten, sondern den Schleifendurchlauf vorzeitig zu beenden. C# stellt ein Schlüsselwort zur Verfügung, das uns dazu in die Lage versetzt: `break`.

```
for(int i = 0; i <= 10; i++)
{
  if(i == 3)
    break;
  Console.WriteLine("Zähler = {0}", i);
}
```

Listing 2.40 »for«-Schleife mit »break« vorzeitig abbrechen

Dieses Codefragment wird zu der folgenden Ausgabe an der Konsole führen:

```
Zähler = 0
Zähler = 1
Zähler = 2
```

`break` beendet die Schleife unabhängig von der im Schleifenkopf formulierten Abbruchbedingung und setzt den Programmablauf hinter dem Anweisungsblock der `for`-Schleife fort.

Sie können `break` auch in einer verschachtelten Schleife einsetzen. Das wirkt sich nur auf die `for`-Schleife aus, in deren direktem Anweisungsblock der Abbruch programmiert ist. Die äußeren Schleifen sind davon nicht betroffen.

Abbruch der Anweisungen im Schleifenblock mit »continue«

Sehr ähnlich wie `break` verhält sich auch die Anweisung `continue`. Die Bearbeitung des Codes in der Schleife wird zwar abgebrochen, aber die Steuerung wieder an den Schleifenkopf übergeben. Mit anderen Worten: Alle Anweisungen, die zwischen `continue` und dem Ende des Anweisungsblocks stehen, werden übersprungen. Das wollen wir uns ebenfalls an einem Codefragment ansehen:

```
for(int i = 0; i <= 10; i++)
{
  if(i == 3)
    continue;
  Console.WriteLine("Zähler = {0}", i);
}
```

Listing 2.41 Abbruch eines Schleifendurchlaufs mir »break«

Die Ausgabe an der Konsole sieht wie folgt aus:

```
Zähler = 0
Zähler = 1
Zähler = 2
Zähler = 4
Zähler = 5
[...]
```

Steht der Zähler auf 3, ist die Abbruchbedingung erfüllt. Es wird `continue` ausgeführt mit der Folge, dass die Laufzeitumgebung die folgende Ausgabeanweisung überspringt und die Schleife mit dem Zählerstand 4 fortgesetzt wird.

Die Ausdrücke der »for«-Schleife

Zum Abschluss der Ausführungen über die Möglichkeiten der `for`-Schleife unter C# kommen wir noch einmal auf die drei Ausdrücke im Schleifenkopf zurück. Was ich bisher noch

nicht erwähnt habe, sei an dieser Stelle nachgeholt: Alle drei Ausdrücke sind optional, müssen also nicht angegeben werden. Fehlt aber ein Ausdruck, gilt er stets als »erfüllt«. Im Extremfall lässt sich eine Schleife sogar ganz ohne explizit ausformulierten Schleifenkopf konstruieren. Wir erhalten dann die kürzeste `for`-Schleife überhaupt – allerdings handelt es sich dann um eine Endlosschleife, da das Abbruchkriterium in dem Sinne als erfüllt gilt, dass die Schleife nicht beendet werden soll:

```
// Endlosschleife
for (; ; );
```

2.7.2 Die »foreach«-Schleife

Die `for`-Schleife setzt drei Ausdrücke voraus, die erst in Kombination die gewünschte Iteration ermöglichen. C# kennt noch ein weiteres Konstrukt, das ein Array vom ersten bis zum letzten Element durchläuft: die `foreach`-Schleife. Sehen wir uns dazu ein Beispiel an, das genauso wie das oben gezeigte operiert:

```
int[] elements = {2,4,6,8};
foreach(int item in elements)
{
  Console.WriteLine(item);
}
```

Anstatt jedes Element über seinen Index anzusprechen, wird nun das Array als eine Einheit angesehen, die aus mehreren typgleichen Elementen gebildet wird. Das Array wird vom ersten bis zum letzten Mitglied durchlaufen, wobei die Adressierung nun über eine Laufvariable als temporäres Element erfolgt, das hier als `item` bezeichnet wird. Der Bezeichner ist natürlich frei wählbar. Bei der Iteration wird `item` jedes Mal auf ein anderes Array-Element verweisen. Daher ist die Indexangabe auch überflüssig.

Die allgemeine Syntax der `foreach`-Schleife lautet:

```
// Syntax: foreach-Schleife
foreach (Datentyp Bezeichner in Array - Bezeichner) { [...] }
```

Beachten Sie, dass die Deklaration der Laufvariablen in den Klammern nicht optional ist. Daher führt das folgende Codefragment zu einem Fehler:

```
int item;
// Fehler im foreach-Statement
foreach (item in intArr) { [...] }
```

Wenn Sie ein Array von Elementen eines einfachen Datentyps durchlaufen, sind die Daten schreibgeschützt, können also nicht verändert werden, z. B.:

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



```
foreach(int item in elements)
  // Unerlaubte Änderung
   item = 33;
```

Hinweis

Möglicherweise lesen Sie diesen Hinweis erst, wenn Sie sich bereits beim Lesen dieses Buches in einem späteren Kapitel befinden. Daher muss ich an dieser Stelle der Vollständigkeit halber darauf hinweisen, dass ein Array nur schreibgeschützt ist, wenn es Wertetypen beschreibt. Zu diesen werden fast alle elementaren Datentypen gezählt. Ein Array von Objekten, die auf Referenztypen basieren, verhält sich anders: Die Objektdaten können durchaus in einer `foreach`-Schleife manipuliert werden. Für alle Leser, die noch nicht weiter in diesem Buch gelesen haben: Über Werte- und Referenztypen erfahren Sie alles Notwendige in Kapitel 3, »Das Klassendesign«.

2.7.3 Die »do«- und die »while«-Schleife

Ist die Anzahl der Iterationen bereits beim Eintritt in die Schleife bekannt, wird zumeist das `for`-Schleifenkonstrukt verwendet. Ergibt sich jedoch erst zur Laufzeit der Anwendung, wie oft der Schleifenkörper durchlaufen werden muss, bietet sich eher eine `do`- oder `while`-Schleife an. Grundsätzlich können alle auftretenden Anforderungen an wiederholt auszuführende Anweisungen mit einem dieser beiden Typen formuliert werden – sie können also die `for`-Schleife durchaus gleichwertig ersetzen.

Die »while«-Schleife

In eine Schleife wird dann eingetreten, wenn bestimmte Bedingungen erfüllt sind. Bei der `for`-Schleife wird diese Bedingung durch den Schleifenzähler festgelegt, bei einer `while`-Schleife wird die Bedingung hinter dem Schlüsselwort `while` in runden Klammern angegeben. Da sich die Anweisungen der Bedingungsprüfung anschließen, spricht man auch von einer kopfgesteuerten Schleife. Sehen wir uns daher zunächst die Syntax dieses Schleifentyps an:

```
while(Bedingung)
{
  [...]
}
```

Bei der Bedingung handelt es sich um einen booleschen Ausdruck, der aus den Vergleichsoperatoren gebildet wird und entweder `true` oder `false` liefert. Eine `while`-Schleife wird ausgeführt, solange die Bedingung wahr, also `true` ist. Die Schleife wird beendet, wenn

die Bedingung `false` ist. Ist die Bedingung schon bei der ersten Überprüfung falsch, werden die Anweisungen im Schleifenkörper überhaupt nicht ausgeführt.

Da im Gegensatz zur `for`-Schleife die Bedingung zum Austritt aus der `while`-Schleife nicht automatisch verändert wird, muss innerhalb des Schleifenkörpers eine Anweisung stehen, die es ermöglicht, die Schleife zu einem vordefinierten Zeitpunkt zu verlassen. Wenn Sie eine solche Anweisung vergessen, liegt der klassische Fall einer Endlosschleife vor.

Hinweis

Wenn Sie beim Testen eines Programms aus der Entwicklungsumgebung heraus in eine Endlosschleife geraten, können Sie mit der Tastenkombination Strg+Pause die Laufzeitumgebung unterbrechen und wieder zur Entwicklungsumgebung zurückkehren.

Im folgenden Beispiel muss der Anwender zur Laufzeit eine Zahl angeben, mit der er die Anzahl der Schleifendurchläufe festlegt. Die zusätzliche Zählervariable `counter` dient als Hilfsvariable, um die Austrittsbedingung zu formulieren. Sie wird innerhalb der Schleife bei jedem Schleifendurchlauf um 1 erhöht und bewirkt, dass die `while`-Schleife zum gewünschten Zeitpunkt verlassen wird.

```
// Beispiel: ..\Kapitel 2\WhileSample
static void Main(string[] args)
{
  Console.Write("Geben Sie eine Zahl zwischen\n");
  Console.Write("0 und einschließlich 10 ein: ");
  int number = Convert.ToInt32(Console.ReadLine());
  int counter = 1;
  while (counter <= number)
  {
    Console.WriteLine($"{counter}.Schleifendurchlauf");
    counter++;
  }
  Console.ReadLine();
}
```

Listing 2.42 Beispielprogramm zu einer »while«-Schleife

Genauso wie eine `for`-Schleife kann auch eine `while`-Schleife entweder mit `break` oder mit `continue` unterbrochen werden. Die Auswirkungen sind bekannt:

- Mit `break` wird die gesamte Schleife als beendet angesehen. Das Programm setzt seine Ausführung mit der Anweisung fort, die dem Anweisungsblock der Schleife folgt.

- Mit `continue` wird der aktuelle Iterationsvorgang abgebrochen. Anweisungen, die innerhalb des Schleifenblocks auf `continue` folgen, werden nicht mehr ausgeführt. Die Steuerung wird an die Schleife zurückgegeben.

Daher würde

```
int value = 0;
while(value < 5)
{
  value++;
  if(value == 3)
    break;
  Console.WriteLine(value);
}
```

die Ausgabe `1`, `2` haben, während der Austausch von `break` gegen `continue` die Zahlenwerte `1`, `2`, `4`, `5` ausgibt.

Die »do«-Schleife

Die `do`-Schleife unterscheidet sich dahingehend von der `while`-Schleife, dass die Schleifenbedingung am Ende der Schleife ausgewertet wird. Die `do`-Schleife ist eine fußgesteuerte Schleife. Die Folge ist, dass die Anweisungen innerhalb des Anweisungsblocks zumindest einmal durchlaufen werden.

```
do
{
  [...]
}while(<Bedingung>);
```

Der Anweisungsblock wird so lange wiederholt ausgeführt, bis die Bedingung `false` ist. Danach wird mit der Anweisung fortgefahren, die sich unmittelbar anschließt.

Die Tatsache, dass die Laufzeit einer Anwendung mindestens einmal in den Anweisungsblock der `do`-Schleife eintaucht, können Sie sich zunutze machen, wenn eine bestimmte Eingabe vom Anwender erforderlich wird. Ist die Eingabe unzulässig, wird eine Schleife so lange durchlaufen, bis sich der Anwender »überzeugen« lässt. Im folgenden Beispiel wird das demonstriert.

```
// Beispiel: ..\Kapitel 2\DoSample
static void Main(string[] args)
{
  // Informationsanzeige
  Console.Write("W - Programm fortsetzen\n");
```

```
  Console.Write("E - Programm beenden\n");
  Console.Write("----------------------\n");
  // Schleife wird so oft durchlaufen, bis der Anwender eine gültige Eingabe macht
  do
  {
    Console.Write("Ihre Wahl: ");
    string eingabe = Console.ReadLine();
    if (eingabe == "W")
      // das Programm nach dem Schleifenende fortsetzen
      break;
    else if (eingabe == "E")
      // das Programm beenden
      return;
    else
    {
      // Fehleingabe
      Console.Write("Falsche Eingabe - ");
      Console.Write("Neueingabe erforderlich\n");
      Console.Write("----------------------\n");
    }
  } while (true);
  Console.WriteLine("...es geht weiter.");
  Console.ReadLine();
}
```

Listing 2.43 Beispiel einer »do«-Schleife

Zugelassen sind nur die beiden Eingaben »W« und »E«. Jede andere Eingabe führt zu einer erneuten Iteration. Die `do`-Schleife ist wegen ihrer Austrittsbedingung

```
while(true)
```

als Endlosschleife konstruiert, aus der es ein kontrolliertes Beenden nur mit der Sprunganweisung `break` gibt, wenn der Anwender mit der Eingabe »W« eine Fortsetzung des Programms wünscht.

Mit der Anweisung `return` wird das laufende Programm vorzeitig beendet. Diese Anweisung dient per Definition dazu, die aktuell ausgeführte Methode zu verlassen. Handelt es sich dabei aber um die `Main`-Methode einer Konsolenanwendung, kommt das dem Beenden der Anwendung gleich.

Kapitel 22

Einführung in die WPF und XAML

Mit der Einführung von *.NET 1.0* wurde auch eine neue Technologie zur Entwicklung von Windows-Anwendungen eingeführt: die *WinForm-API*. Im Grunde genommen war die WinForm-API keine Neuentwicklung, da sie die *Windows-API* nutzte. Trotz der Erweiterung der Windows-API wurde an der grundlegenden Architektur nichts verändert.

Unter dem Codenamen *Avalon* startete Microsoft Mitte des letzten Jahrzehnts die Entwicklung einer neuen Bibliothek für grafische Benutzeroberflächen, die im Jahr 2006 unter dem Bezeichner *Windows Presentation Foundation* (WPF) als Teil von *.NET 3.0* veröffentlicht wurde. Bereits in *Visual Studio 2008* wurde die WPF neben der traditionellen Technologie für die WinForm-API fest in die Entwicklungsumgebung integriert. Nach einigen Startschwierigkeiten, die nicht nur auf Mängel zurückzuführen waren, fand die WPF eine breite Akzeptanz in der Entwicklergemeinde und gehört nunmehr zur bevorzugten Technologie zur Entwicklung von Windows-Anwendungen.

In den kommenden Kapiteln werden wir uns mit der WPF beschäftigen. Dabei können wir nicht alle Aspekte und Konzepte berücksichtigen. Aber ich möchte Ihnen einen Einstieg in die Technologie geben und Ihnen zeigen, wie Sie Windows-Anwendungen mit der WPF entwickeln. Sie werden feststellen, dass die Lernkurve nicht so steil ist wie bei der nun auf das Abstellgleis geschobenen WinForm-API.

Anmerkung

Nach Aussagen von Microsoft wird die WinForm-API nicht mehr weiterentwickelt. Dennoch ist sie auch weiterhin elementarer Bestandteil von Visual Studio und wird es meiner Ansicht nach auch noch länger bleiben. Im Grunde genommen brauchen Sie nicht zu fürchten, ein neues Projekt mit der WinForm-API zu starten, aber andererseits sind Sie dann nicht mehr *up to date* und können die vielen interessanten und herausragenden Programmiertechniken der WPF nicht nutzen. Sollten Sie in Zukunft, aus welchen Gründen auch immer, planen, Ihre WinForm-Anwendung auf WPF umzustellen, fangen Sie wegen der doch sehr unterschiedlichen Techniken von vorn an. Deshalb sollte sich die Frage, ob WinForms oder WPF, eigentlich gar nicht stellen.

22.1 Die Merkmale einer WPF-Anwendung

Am Anfang stellt sich zuerst die Frage, welche typischen Charakteristika eine WPF-Anwendung auszeichnen und wo die Vorteile im Vergleich zu den WinForms zu suchen sind. Die folgende Liste stellt Ihnen vorab eine Reihe von Vorzügen der WPF vor.

- Die Benutzeroberfläche wird mit einer an XML angelehnten Sprache beschrieben: mit *XAML* (*eXtensible Application Markup Language*, gesprochen »Xemmel«). XAML ist ausgesprochen mächtig und offenbart erstaunliche Fähigkeiten. Zusammen mit den in der WPF eingeführten Konzepten werden Sie vergleichsweise wenig C#-Programmcode schreiben und sich mehr auf XAML konzentrieren. Die Folge ist, dass der XAML-Code relativ umfangreich werden kann, während sich der C#-Code reduziert.
- WPF-Anwendungen bieten eine umfangreiche Unterstützung von 2D- und 3D-Grafiken. Dabei wird die Grafikausgabe durch *DirectX* genutzt mit der Folge, dass die GPU der Grafikkarte zur Berechnung der grafischen Elemente herangezogen wird und nicht die CPU. Das führt zu einem deutlich besserem Leistungsverhalten.
- WPF-Anwendungen bieten vielfältige Datenbindungsmöglichkeiten für die Komponenten. Das ist in den Augen vieler Entwickler eine der Stärken der WPF. Deshalb werden wir uns mit dieser Thematik noch intensiv auseinandersetzen.
- Die WPF-Ausgabe ist *vektorbasiert*. Das bedeutet, dass auch beim Skalieren keine hässlichen Pixel zu sehen sind, sondern immer ein fließender Verlauf der grafischen Darstellung.
- WPF-Anwendungen bieten vielfältige grafische Unterstützung, z. B. zur Darstellung der Steuerelemente, grafische Animationen, Unterstützung von Videos, Bildern und Audiodateien.
- Die sich hinter der WPF verbergende Technologie wird in einer Vielzahl verschiedener Projekte verwendet (Windows Workflow Foundation WF, Universal Windows Platform UWP usw.).
- Im Vergleich zur WinForm-API gibt es vielfältige Gestaltungsmöglichkeiten, nicht nur durch das mögliche Verschachteln der Elemente, sondern auch durch die einfachen Umgestaltungsmöglichkeiten der visuellen Komponenten.
- Nicht unerwähnt bleiben sollte auch, dass die WPF ein altes Problem der WinForms gelöst hat: Früher war es kaum möglich, das Fenster automatisch an die Monitorauflösung (DPI) anzupassen. Das führte dazu, dass Randbereiche von Dialogen möglicherweise nicht mehr angezeigt wurden und im Extremfall einige Bedienelemente der Oberfläche unerreichbar waren.

Summa summarum stellen die grafischen Fähigkeiten der WPF alles Vergangene in den Schatten. Wollen Sie runde Buttons? Kein Problem. Wollen Sie runde Fenster? Ebenfalls kein Problem. Neben den Vorteilen der grafischen Gestaltung können auch andere Gesichtspunkte wie die der umfangreichen Datenbindungsmöglichkeiten die Entscheidung für die WPF und somit gegen die WinForm-API beeinflussen. Darüber hinaus müssen Sie bei der Entscheidungsfindung berücksichtigen, dass Microsoft die WinForm-API nicht mehr weiterentwickelt und voll auf die WPF setzt.

Die Trennung zwischen Oberflächenbeschreibung mit XAML und dem C#-Code gestattet es, dass die Oberfläche von einem Grafiker gestaltet wird, während der Entwickler den Code dazu schreibt. Speziell für Grafiker ist das Tool *Blend* gedacht, dessen Oberfläche stark an die Software *Photoshop* erinnert. Blend wird zusammen mit Visual Studio installiert.

22.1.1 Anwendungstypen

Visual Studio 2019 bietet Ihnen im Zusammenhang mit der WPF vier verschiedene Projektvorlagen an:

- WPF-App (.NET Framework)
- WPF-Browser-App (.NET Framework)
- WPF-Benutzersteuerelementbibliothek (.NET Framework)
- Benutzerdefinierte WPF-Steuerelemente (.NET Framework)

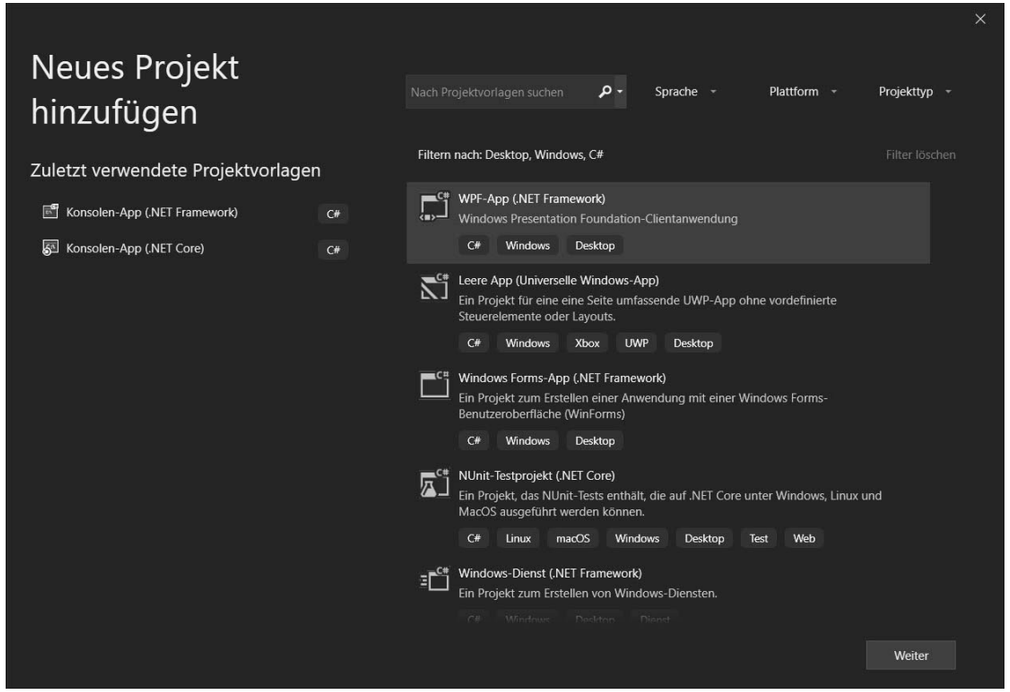

Abbildung 22.1 WPF-Projektvorlage

WPF-App: Dieser Anwendungstyp entspricht im Wesentlichen einer herkömmlichen Windows-Anwendung. Die charakteristischen Eigenschaften gleichen denen einer Win-Form-Anwendung. WPF-Anwendungen werden in einem eigenen Fenster ausgeführt.

WPF-Browser-App: Im Gegensatz zu klassischen WPF-Anwendungen stellen WPF-Browseranwendungen keine eigenen Fenster bereit – die Ausgabe erfolgt im Browser. Außerdem werden WPF-Browseranwendungen nicht auf der lokalen Maschine installiert, was zur Folge hat, dass es nicht möglich ist, einen Verweis auf die Anwendung im Startmenü zu hinterlegen.

Benutzerdefinierte WPF-Steuerelementbibliothek (User Control): Visual Studio 2019 stellt zwei Projektvorlagen zum Entwickeln eigener Steuerelemente bereit. Die Variante mit einem User Control ist die einfachere von beiden. Etwas vereinfacht gesagt, wird dabei ein neues Steuerelement aus mehreren bestehenden Steuerelementen gebildet.

WPF-Benutzersteuerelementbibliothek (Custom Control): Der Aufwand, ein Custom Control zu entwickeln, ist deutlich größer, hat aber im Vergleich zu den User Controls auch Vorteile. Beispielsweise kann ein Custom Control durch Templates angepasst werden.

22.1.2 Eine WPF-Anwendung und ihre Dateien

Wir wollen nun ein erstes Projekt vom Typ *WPF-App* starten und uns zuerst die Entwicklungsumgebung ansehen.

Auf der linken Seite sehen Sie die Toolbox. Darin werden alle standardmäßig zur Verfügung gestellten Steuerelemente der WPF aufgelistet. Diejenigen, die bereits mit ähnlichen Entwicklertools gearbeitet haben, werden nun sofort versuchen, in der Toolbox eine *Control* auszuwählen und sie mittels *Drag & Drop* in den Designer zu ziehen. Verwerfen Sie diese Idee am besten sofort! So wird keine Oberfläche in der WPF gestaltet – zumindest, solange Sie eine grafische Benutzeroberfläche unter Einbeziehung aller Vorteile der WPF abliefern wollen. Zu dieser Erkenntnis werden Sie auch kommen, sobald Sie im folgenden Kapitel gelernt haben, Layoutcontainer einzusetzen. Wenn ich an einem WPF-Projekt arbeite, ist die Toolbox fast immer geschlossen.

Im rechten unteren Bereich sehen Sie das Eigenschaftsfenster. Im Eigenschaftsfenster werden die Eigenschaften der im XAML-Code selektierten Komponente angezeigt. Leider ist die Vorgabe so, dass die Eigenschaften nach Kategorien angeordnet werden. Sie können aber auf eine alphabetische Sortierung umschalten. Vermutlich werden Sie nur selten auf die Unterstützung des Eigenschaftsfensters zurückgreifen. Wenn Sie nämlich die wichtigsten und gängigsten Eigenschaften der WPF-Controls kennen, können Sie Eigenschaften dank der *IntelliSense*-Unterstützung viel schneller im XAML-Code festlegen als im Eigenschaftsfenster selbst.

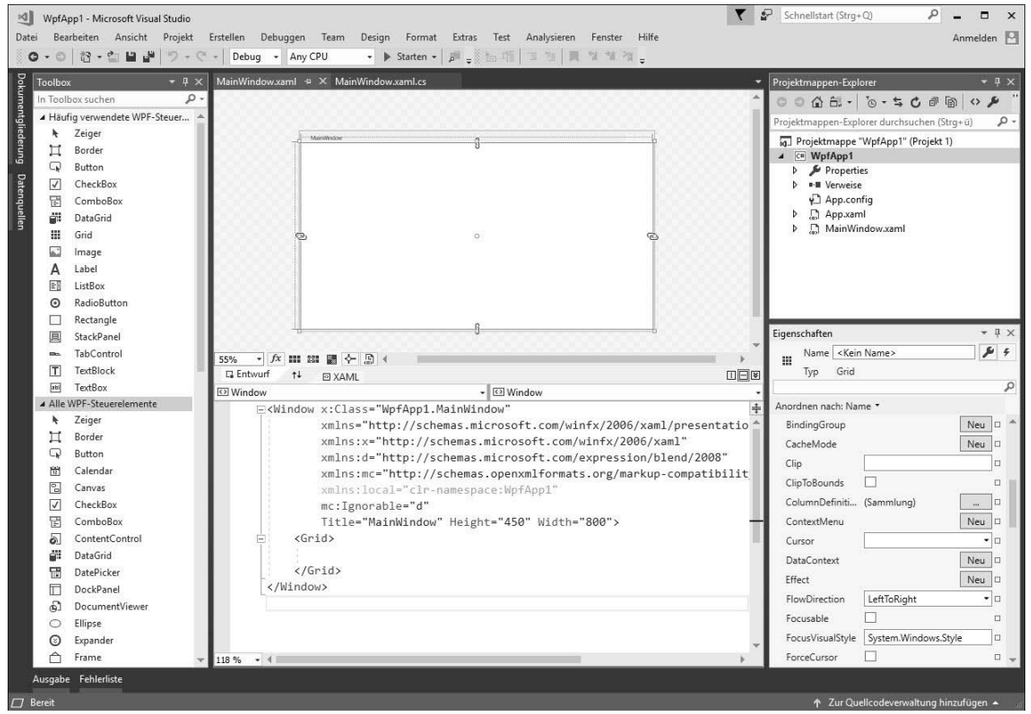

Abbildung 22.2 Die Entwicklungsumgebung einer WPF-Anwendung

Im mittleren oberen Bereich des Editors wird das Fensterdesign angezeigt, der dazugehörige XAML-Code erscheint darunter. Im XAML-Code gestalten Sie das Fenster, nicht im Designer selbst. Beide Teilbereiche, der Designer und der XAML-Code, synchronisieren sich bei Änderung gegenseitig.

Werfen Sie nun einen Blick in den Projektmappen-Explorer. Hier finden Sie unter anderem mit *App.xaml*, *App.xaml.cs*, *MainWindow.xaml* und *MainWindow.xaml.cs* vier Dateien, die wir uns nun genauer ansehen.

Die Datei »MainWindow.xaml«

In der Datei *MainWindow.xaml* steckt der XAML-Code des Fensters, den Sie im unteren Bereich des Code-Editors sehen.

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
    </Grid>
</Window>
```

Listing 22.1 Der standardmäßig erzeugte XAML-Code eines Fensters

In der ersten Zeile wird mit `x:Class="WpfApplication1.MainWindow` der Bezug zum C#-Code hergestellt. Danach sind mit `xmlns` mehrere XML-Namespaces angegeben, denen mit `Title` die Beschriftung der Titelleiste folgt. `Height` und `Width` geben die Ausgangsgröße des Fensters an.

Alle Steuerelemente einer WPF-Anwendung positionieren sich innerhalb eines Layoutcontainers. Mit `Grid` wird sofort ein Vorschlag gemacht, den Sie aber nach eigenem Ermessen durch einen anderen Container ersetzen können. Die Layoutcontainer werde ich in Kapitel 23 vorstellen.

Die Datei »MainWindow.xaml.cs«

MainWindow.xaml ist die Datei, die zunächst einmal nur die Oberfläche des Fensters beschreibt. Bei *MainWindow.xaml.cs* handelt es sich um die Datei, in der Sie den C#-Programmcode schreiben. Diese Datei wird auch als *Code-Behind-Datei* bezeichnet. Wie üblich werden Sie hier die Ereignishandler implementieren, Eigenschaften und Felder beschreiben usw. Die Datei weist nicht viel Inhalt auf. Nur der parameterlose Standardkonstruktor, in dem die Methode `InitializeComponent` aufgerufen wird, ist dort zu finden.

```
using System;
[...]
namespace WpfApplication1
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }
  }
}
```

Listing 22.2 Der Inhalt der Datei »MainWindow.xaml.cs«

Die Methode `InitializeComponent` lädt die für das Fenster benötigten Komponenten, die in der Datei *MainWindow.xaml* definiert sind, indem die statische Methode `LoadComponent` der Klasse `Application` aufgerufen wird.

Die Datei »App.xaml«

Auch zu dieser Datei gehört eine Code-Behind-Datei, die mit dem Attribut `x:Class` angegeben wird. Mit `xmlns` werden hier zwei Namespaces eingebunden, und das Attribut `StartupUri` gibt an, mit welchem Fenster die Anwendung gestartet werden soll.

```
<Application x:Class="WpfApplication1.App"
             xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
             xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
             xmlns:local="clr-namespace:WpfApplication1"
             StartupUri="MainWindow.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

Listing 22.3 Der XAML-Code in der Datei »App.xaml«

Im Bereich zwischen dem einleitenden Tag `<Application.Resources>` und dem ausleitenden `</Application.Resources>` können Sie anwendungsweite Ressourcen eintragen.

Die Datei »App.xaml.cs«

Hierbei handelt es sich um die *Code-Behind-Datei* zu *App.xaml*. Sie definiert die Klasse `App`, die von `Application` abgeleitet ist, und weist keinen Code auf. Die Klasse `App` veröffentlicht Eigenschaften und löst Ereignisse aus, auf die Sie hier reagieren können.

```
using System;
[...]
namespace WpfApplication1
{
  public partial class App : Application
  {
  }
}
```

Listing 22.4 Der Code der Datei »App.xaml.cs«

22.1.3 Ein erstes WPF-Beispiel

Ein paar allgemeine Dinge rund um WPF-Anwendungen haben Sie nun erfahren. Wahrscheinlich sind Sie auch schon gespannt, wie die Oberfläche einer WPF-Anwendung tatsäch-

lich aussehen kann. Um ein Gefühl dafür zu bekommen, sehen Sie sich bitte die zugegebenermaßen sehr einfache Oberfläche in Abbildung 22.3 an. Mit Funktionalitäten ist das Fenster nicht ausgestattet.

Abbildung 22.3 Eine einfache WPF-Oberfläche

Dieser Benutzeroberfläche (auch als *GUI*, für *Graphical User Interface*, bezeichnet) liegt der folgende XAML-Code zugrunde:

```
<Window x:Class="WpfApp1.MainWindow"
   [...] >
  <Window.Background>
    <LinearGradientBrush EndPoint="0,1" StartPoint="0,0">
      <GradientStop Color="#FF1E1E1E" Offset="0" />
      <GradientStop Color="#FF646464" Offset="1" />
      <GradientStop Color="#FF828282" Offset="0.987"/>
    </LinearGradientBrush>
  </Window.Background>
  <Window.Resources>
    <Style x:Key="style1">
      <Setter Property="Control.Foreground" Value="White" />
      <Setter Property="Control.VerticalAlignment" Value="Center" />
    </Style>
    <Style TargetType="TextBox">
      <Setter Property="Grid.Column" Value="1" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Window.Resources>
  <DockPanel>
    <StackPanel DockPanel.Dock="Top" Background="#FF49494D">
      <Label Foreground="White" FontSize="16" HorizontalAlignment="Center">
         Personendaten
      </Label>
    </StackPanel>
```



```
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
                HorizontalAlignment="Right">
      <Button Height="24" Width="100" Margin="5">OK</Button>
      <Button  Width="100" Margin="5">Abbrechen</Button>
    </StackPanel>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <Label Style="{StaticResource style1}" >Name:</Label>
      <Label Style="{StaticResource style1}" Grid.Row="1">Alter:</Label>
      <Label Style="{StaticResource style1}" Grid.Row="2">Adresse:</Label>
      <TextBox Grid.Column="1" Text="{Binding Path=Name}" />
      <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Alter}" />
      <TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=Adresse}" />
    </Grid>
  </DockPanel>
</Window>
```

Listing 22.5 Der XAML-Code des Fensters aus Listing 22.3

Eingerahmt wird der XAML-Code durch das Element `Window`. In diesem ist alles im Zusammenhang mit der Darstellung des Fensters (und mehr) enthalten.

Im `Window`-Element sehen Sie ein direkt untergeordnetes `DockPanel`-Element. Ein `DockPanel` gehört der Gruppe der sogenannten *Container-Controls* an. Container-Controls teilen den Arbeitsbereich auf, arrangieren und positionieren die in ihnen enthaltenen Steuerelemente. `Grid`, `StackPanel`, `Canvas` oder auch `DockPanel` sind ebenfalls typische Vertreter solcher Container-Controls.

Ein `Window` kann nur ein direkt untergeordnetes Element haben, indirekt können es beliebig viele sein. Dem `DockPanel` sind drei weitere Layoutcontainer untergeordnet: Zwei `StackPanel`-Elemente und ein `Grid`. Standardmäßig beschreibt ein `Grid` nur eine Zeile und eine Spalte. Im `Grid` werden drei Zeilen vom Typ `RowDefinition` sowie zwei Spalten vom Typ `ColumnDefinition` definiert, um die Personendaten anzuzeigen.

Im oberen `StackPanel` ist ein `Label` enthalten, um eine allgemeine Überschrift darzustellen, im unteren `StackPanel` sind zwei Schaltflächen horizontal ausgerichtet.

In der Sektion `<Window.BackGround>...</Window.BackGround` wird der farbliche Hintergrund des Fensters festgelegt. Hier sehen Sie, dass im Grunde genommen einfache Eigenschaften (hier `BackGround`) durch Elementverschachtelung zu interessanten Effekten führen können. Die Elementverschachtelung beschränkt sich jedoch nicht nur auf grafische Komponenten. Auch Steuerelemente lassen sich in gleicher Weise verschachteln. Mit dieser Technik können Sie letztendlich Steuerelemente nach eigenen Vorstellungen aufbauen und ihnen eine eigene Charakteristik geben.

Ein weiterer wichtiger Abschnitt innerhalb des Fensters wird durch `Window.Resources` beschrieben. In vielen WPF-Fenstern ist das der Bereich, der den meisten XAML-Code enthält. In unserem Beispiel werden hier zwei `Style`-Objekte beschrieben. Es handelt sich um einen allgemein verfügbaren, untypisierten Stil (`x:Key="style1"`) und um einen typisierten Stil für alle `TextBox`-Steuerelemente des Fensters. Der untypisierte Stil wird von einigen `Label`-Steuerelementen benutzt, um die im entsprechenden `Style` angegebenen Eigenschaftseinstellungen gemeinsam zu verwenden. Das Prinzip erinnert sehr an die *Cascading Style Sheets* (*CCS*) in HTML-Seiten.

Der XAML-Code dient aber nicht nur dazu, die Oberfläche zu beschreiben. Die Fähigkeiten gehen deutlich darüber hinaus. Verhaltensweisen und Techniken, die früher nur durch Programmcode ausgedrückt werden konnten, lassen sich mit XAML umsetzen – hier sei ohne weitere Erläuterung der Begriff *Trigger* ins Rennen geschickt. Im Endeffekt führt das dazu, dass der Code der im Hintergrund ablaufenden Operationen reduziert werden kann. Selbstverständlich können Sie alles das, was mit XAML beschrieben wird, auch mit C#-Code erreichen.

Hinweis

Wie Sie an diesem Beispiel erkennen, ist der XAML-Code relativ umfangreich. Ich werde daher bei vielen der folgenden Beispielprogramme nicht alle Einzelheiten des XAML-Codes angeben, sondern mich oft auf die entscheidenden Merkmale beschränken.

Hinweis

Sie finden das komplette Beispiel (Download von *www.rheinwerk-verlag.de/4699*, Materialien zum Buch) im Unterordner *..\Kapitel 22\WpfApp1*

22.1.4 Wichtige WPF-Features

Die WPF wartet noch mit zahlreichen Features auf, die ich bisher noch nicht erwähnt habe. Alle möchte und kann ich Ihnen an dieser Stelle noch nicht aufzählen, aber auf zwei muss ich

hier bereits eingehen, da ich sie erst in einem späteren Kapitel ausführlich erörtern werde, die Begriffe aber bereits im Vorfeld immer wieder fallen werden. Dabei handelt es sich um die *Dependency Properties* und die *Routed Events*.

Dependency Properties

Rufen wir uns das Konzept der sogenannten CLR-Eigenschaften in Erinnerung: Eine `private`-deklarierte Variable innerhalb einer Klasse wird durch eine Property, die einen `get`- und einen `set`-Accessor enthält, veröffentlicht. Jedes Objekt besitzt einen Pool von Eigenschaften, bei gleichen Typen ist auch die Anzahl der Eigenschaften gleich.

Die WPF führt einen anderen Eigenschaftstypus ein: die *Dependency Properties*, im deutschen auch als *Abhängigkeitseigenschaften* bezeichnet. Dieses Konzept ist ganz anders, denn alle Eigenschaften eines bestimmten Typs liegen in einem gemeinsam nutzbaren Container – zumindest, solange die Dependency Property nicht von einem definierten Standardwert abweicht. Wird eine Abhängigkeitseigenschaft eines Objekts individuell für das Objekt festgelegt, wird die Eigenschaft zu einer objektspezifischen.

Dependency Properties sind mit zahlreichen eigenen Verhaltensmerkmalen ausgestattet. Dazu zählt beispielsweise, dass nur Dependency Properties im Sinne der WPF an eine Datenquelle gebunden werden können. Styles können nur Eigenschaften beschreiben, wenn es sich um eine Abhängigkeitseigenschaft handelt. Animationen, ebenfalls ein starkes Feature der WPF, setzen auch Abhängigkeitseigenschaften voraus.

Das sollte in diesem Moment als Information genügen. In Kapitel 26, »Dependency Properties«, werde ich Ihnen zeigen, wie Dependency Properties codiert werden.

Im Zusammenhang mit den Dependency Properties muss ich noch eine verwandte Eigenschaftsgruppe erwähnen: Die *Attached Properties* (im Deutschen auch als *angehängte Eigenschaften* bezeichnet). Als angehängte Eigenschaften werden solche bezeichnet, die ein Steuerelement von seinem hierarchisch übergeordneten Container erhält. Auch dazu finden sich in Listing 22.5 Beispiele: Es handelt sich um die Angaben von `Grid.Column` und `Grid.Row`. Mit diesen Eigenschaften wird die Position hinsichtlich Spalte und Zeile des betreffenden Steuerelements im übergeordneten `Grid`-Control beschrieben. Das `Grid` stattet automatisch alle ihm direkt untergeordneten Controls mit diesen beiden Eigenschaften aus. Lägen die betroffenen Steuerelemente in einem anderen Container, gäbe es diese beiden Eigenschaften nicht.

Routed Events

In Listing 22.6 erkennen Sie sehr schön, dass XAML es ermöglicht, verschiedene Elemente ineinander zu verschachteln. Dieses Konzept kann in Konsequenz dazu führen, dass innerhalb eines `Button`-Elements ein `Label` positioniert ist, innerhalb dessen wiederum ein `Image`-Ele-

ment zu finden ist. Berücksichtigen wir dazu noch, dass der `Button` innerhalb einer `Grid`-Zelle positioniert ist, die ihrerseits dem `Window`-Element untergeordnet ist, ergibt sich die folgende Elementhierarchie:

```
<Window>
  <Grid>
    <Button ...>
      <Label ...>
        <Image ... />
      </Label>
    </Button>
  </Grid>
</Window>
```

Listing 22.6 Verschachtelung von WPF-Elementen

Nehmen wir an, der Anwender würde auf das `Image`-Element klicken. Betrachten wir die Situation ganz nüchtern, müssen wir uns die Frage stellen, ob mit der Aktion tatsächlich das `Image` angeklickt werden sollte oder der `Button` reagieren soll. Vermutlich ist die Reaktion der Schaltfläche gewünscht.

In einer klassischen, nicht WPF-basierten Oberfläche würde das oberste Element auf das Ereignis reagieren, in unserem fiktiven Beispiel wäre das demnach das `Image`. Eine Weiterleitung an den `Button` zu codieren ist natürlich möglich, aber auch mit einem gewissen Aufwand verbunden.

Genau an dieser Stelle kommen die *Routed Events* ins Spiel. Routed Events leiten die Ereignisse an über- oder untergeordnete Elemente weiter. Dabei kommt es zu zwei Ereignisketten: Zuerst werden die getunnelten Events (englisch: *tunneled events*) ausgelöst, die beim `Window` starten und über das `Grid`, den `Button` und das `Label` am Ende beim `Image` landen. Anschließend werden die Ereignisse *zurückgebubbelt* (englisch *bubbled events*). Diesmal geht die Ereigniskette beim `Image` los und setzt sich über das `Label`, den `Button`, das `Grid` bis zurück zum Ausgangspunkt `Window` fort. Routed Events werde ich in Kapitel 27, »Ereignisse in der WPF«, genauer beleuchten.

22.1.5 Der logische und der visuelle Elementbaum

Erstellen Sie eine Benutzeroberfläche mit der WPF, erzeugen Sie eine Hierarchie ineinander verschachtelter Elemente. Dabei gibt es mit `Window` immer ein Wurzelelement, in dem die anderen Elemente enthalten sind. Jedes Element kann seinerseits wieder praktisch unbegrenzt untergeordnete Elemente enthalten. Auf diese Weise bildet sich eine durchaus tiefgehende Elementstruktur, die als *Elementbaum* bezeichnet wird.

Aufgrund der Architektur der WPF wird zwischen zwei Baumstrukturen unterschieden:

- logischer Elementbaum
- visueller Elementbaum

Zur Verdeutlichung der Unterschiede zwischen den genannten beiden Elementbäumen dient der folgende XAML-Code, der innerhalb eines `Grid`-Controls ein `Label`- und ein `Button`-Steuerelement beschreibt.

```
<Window x:Class="ElementTree.MainWindow" ...>
  <Stackpanel>
    <Label Content="Label" />
    <Button Content="Button" />
  </Stackpanel>
</Window>
```

Der *logische Elementbaum* wird in diesem XAML-Beispiel durch die Elemente `MainWindow`, `StackPanel`, `Label` und `Button` gebildet. Er enthält demnach alle Elemente, die in XAML bzw. im Code definiert sind. Zum logischen Baum gehören beispielsweise weder Füllmuster noch Animationen.

Jedes WPF-Steuerelement ist selbst durch eine mehr oder weniger große Anzahl visueller Einzelkomponenten aufgebaut. Alle Einzelkomponenten, die als Basis eine der beiden Klassen

- `System.Windows.Media.Visual` oder
- `System.Windows.Media.Media3D.Visual3D`

angehören, bilden zusammen den *visuellen Elementbaum*. Andere Elemente, beispielsweise `String`-Objekte, gehören nicht zum visuellen Elementbaum, weil sie kein eigenes Renderverhalten benötigen. Zu den Elementen des visuellen Elementbaums hingegen sind die Klassen `Button` und `Label` zu zählen. In Abbildung 22.4 ist die Zugehörigkeit verschiedener Einzelkomponenten zu den verschiedenen Elementbäumen dargestellt.

Warum wird zwischen den Elementbäumen unterschieden?

WPF-Steuerelemente haben kein eigenes, festes Layout. Beim Rendern wird der visuelle Elementbaum jeder einzelnen Komponente durchlaufen. Beispielsweise ist der Rahmen einer Schaltfläche zwar Bestandteil der Schaltfläche, kann aber jederzeit durch ein anderes Element ersetzt werden. Infolgedessen kommt es aber auch zu einem Problem hinsichtlich Ereignisauslösung: Da der Rahmen einer Schaltfläche praktisch beliebig beschrieben werden kann, muss der `Button` in der Lage sein, zu erkennen, ob er innerhalb oder außerhalb des Rahmens angeklickt wird. Dazu ist ein Ansatz notwendig, der sich am tatsächlichen Layout orientiert. In der WPF werden auch aus diesem Grund die Routed Events beschrieben, die sich nicht am logischen, sondern am visuellen Elementbaum orientieren.

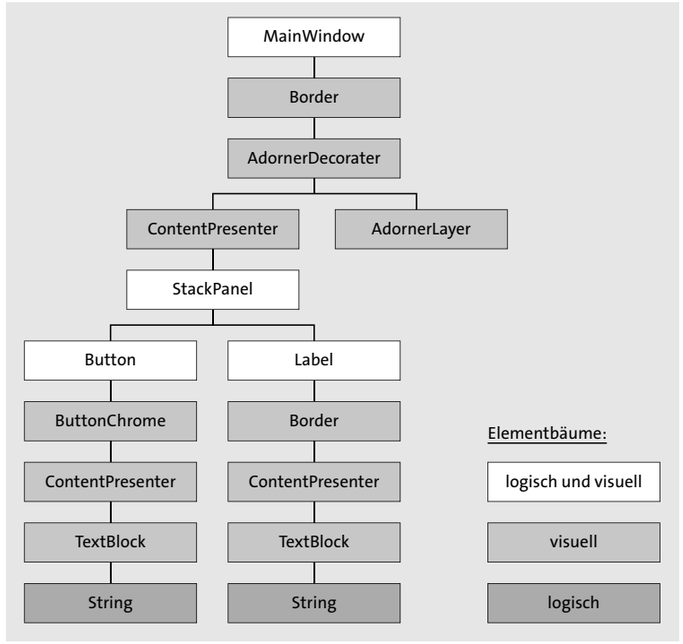

Abbildung 22.4 Logischer und visueller Elementbaum

22.2 XAML (Extended Application Markup Language)

XAML ist eine deklarative Programmiersprache, deren Wurzeln auf *XML* zurückzuführen sind. XAML unterliegt damit auch denselben strengen Regeln wie XML:

- Elemente werden durch Tags beschrieben.
- Jedes Starttag bedarf zwingend eines Endtags.
- Die Groß-/Kleinschreibung muss berücksichtigt werden.

XAML ist im Grunde genommen eine Erweiterung der XML-Spezifikation. Sie werden im XAML-Code viele bekannte Regeln der XML wiederfinden, aber auch mit Neuerungen oder Ergänzungen konfrontiert werden, die das Gesamtkonzept von XAML im Vergleich zu XML besser unterstützen und auch aufwerten.

22.2.1 Die Struktur einer XAML-Datei

Sehen wir uns den XAML-Code an, den Visual Studio 2019 beim Erstellen eines neuen Fensters erzeugt:


```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
  <Grid>
  </Grid>
</Window>
```

Listing 22.7 Struktur der XAML-Datei eines Fensters

Jede XML-Datei hat ein Wurzelelement, das alle anderen Elemente einschließt. Das gilt natürlich für die XAML-Datei einer WPF-Anwendung. Hier handelt es sich um das Element `Window`. Es weist von Anfang an einige Attribute auf, die eine gewisse Grundcharakteristik sicherstellen:

- Das erste, `x:Class`, gibt die Code-Behind-Datei an, die den C#-Code des aktuellen XAML-Dokuments enthält.
- Mit `xmlns` werden mehrere XML-Namespaces bekanntgegeben, damit die Elemente im XAML-Code einwandfrei identifiziert werden können.
- Mit dem Attribut `Title` wird anschließend die Zeichenfolge beschrieben, die in der Titelleiste des Fensters angezeigt wird, und `Height` und `Width` legen die Gesamthöhe bzw. -breite des Fensters fest.

Im Wurzelelement `Window` sind alle Komponenten enthalten, aus denen sich das Fenster zusammensetzt: Schaltflächen, Textboxen, Listenfelder usw. Da `Window` jedoch grundsätzlich nur ein direkt untergeordnetes Element haben kann, handelt es dabei um ein Containersteuerelement, das seinerseits selbst beliebig viele Steuerelemente aufnehmen kann. Per Vorgabe wird immer ein `Grid`-Element erzeugt. Es gibt noch ein paar Layoutcontainer mehr, die alle in irgendeiner Weise auf eine bestimmte Darstellung oder Ausrichtung der in ihnen enthaltenen Steuerelemente spezialisiert sind. Neben dem `Grid` ist beispielsweise das `StackPanel` ein häufig verwendeter Container. Ein charakteristisches Merkmal aller Containersteuerelemente ist, dass sie über die Eigenschaft `Children` verfügen, die eine Auflistung von `UIElement`-Objekten verwaltet.

Im Codefragment in Listing 22.8 sehen Sie das Stammelement `Window` nebst dem untergeordneten Container vom Typ `Grid`. Das `Grid` wird nur durch eine Zeile und eine Spalte, also eine Zelle beschrieben, die ein `Button`-Element enthält.

```
<Window x:Class="WpfApplication1.MainWindow"
        [...]
        Title="MainWindow" Height="163" Width="300">
  <Grid>
    <Button FontSize="18" Background="LightGray" Name="btnButton1">
       Der erste Button
    </Button>
  </Grid>
</Window>
```

Listing 22.8 »Grid« mit einem eingebetteten Button

Beachten Sie, dass die Schaltfläche keine Angaben zu ihren Abmessungen enthält. Es liegt in der Natur des übergeordneten `Grid`-Containers, dass der dem `Grid` als einziges Element untergeordnete `Button` dann den gesamten Containerbereich für sich beansprucht (siehe Abbildung 22.5). Wie Sie in den Bereich des `Grid`-Objekts auch mehrere Steuerelemente unterbringen können, werden sie später noch sehen.

Abbildung 22.5 Die grafische Benutzeroberfläche aus Listing 22.8

Das Element `Button` entspricht der gleichnamigen Klassendefinition im Namespace `System.Windows.Control`. Die Angabe `<Button>` im XAML-Code bewirkt die Instanziierung des entsprechenden Elements über den parameterlosen Konstruktor. Die Attribute `FontSize`, `Background`, `Content` und `Name` sind Eigenschaften der Klasse `Button`.

Alternativ können Sie die Schaltfläche im Programmcode der Code-Behind-Datei erzeugen, beispielsweise nach dem Aufruf der Methode `InitializeComponent` im Konstruktor:

```
public MainWindow()
{
  InitializeComponent();
  Button btnButton1 = new Button();
  btnButton1.FontSize = 18;
  btnButton1.Background = new SolidColorBrush(Colors.LightGray);
  btnButton1.Content = "Der erste Button";
  grid1.Children.Add(btnButton1);
}
```

Listing 22.9 Button aus Listing 22.5 mit Programmcode erzeugen

Mit der letzten Anweisung im Listing wird die Schaltfläche ihrem übergeordneten `Grid`-Container zugeordnet.

22.2.2 Eigenschaften eines XAML-Elements in Attribut-Schreibweise festlegen

Jedes WPF-Element hat zahlreiche Eigenschaften. Eine Möglichkeit ist es, diese im XAML-Code durch Attribute anzugeben. Beabsichtigen Sie, beispielsweise die Breite und die Höhe einer Schaltfläche festzulegen, müssen Sie die Eigenschaften `Height` und `Width` als Attribute angeben:

```
<Button Height="50" Width="100"></Button>
```

oder gleichwertig

```
<Button Height="50" Width="100" />
```

Allen Attributen werden die Werte grundsätzlich immer als `String` übergeben. Die Zeichenfolge wird von einem Typkonverter anschließend in den von der Eigenschaft beschriebenen Datentyp umgewandelt. Bei `Height` und `Width` ist das die Umwandlung in den Datentyp `Double`.

Bei dem Zieldatentyp muss es sich nicht unbedingt um einen elementaren Datentyp handeln. Legen Sie zum Beispiel die Hintergrundfarbe `Background` fest, verbirgt sich dahinter die Konvertierung in den schon verhältnismäßig komplexen Typ `Brush`.

22.2.3 Eigenschaften im Eigenschaftsfenster festlegen

Sie müssen die Eigenschaften eines Elements nicht im XAML-Code direkt angeben – auch wenn es dank der IntelliSense-Unterstützung nicht weiter schwierig ist. Sie können stattdessen die Eigenschaften auch im Eigenschaftsfenster von Visual Studio für die aktuell im Designer ausgewählte Komponente (siehe Abbildung 22.6) definieren.

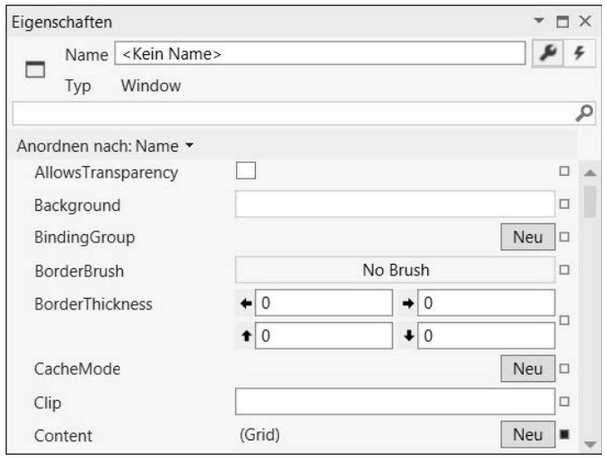

Abbildung 22.6 Das Eigenschaftsfenster einer WPF-Komponente

22.2.4 Die Eigenschaft-Element-Syntax

Die Attribut-Schreibweise ist zwar kompakt, birgt aber den Nachteil, dass einer Eigenschaft nur eine Zeichenfolge zugewiesen werden kann. Diese wird meistens auf einen elementaren Datentyp zurückgeführt. Eigenschaften können aber auch komplexer Natur sein. Nehmen wir exemplarisch die Eigenschaft `Background`, die die Hintergrundfarbe beschreibt. Soll diese einheitlich Blau sein, ist die Attribut-Schreibweise vollkommen ausreichend:

```
<Button Background="Blue" />
```

Was ist aber, wenn die Hintergrundfarbe nicht durch eine konkrete Farbe, sondern durch einen Farbverlauf beschrieben werden soll? Dafür sind mehrere Objekte notwendig. In solchen Fällen kann man die Attribut-Schreibweise nicht mehr anwenden, und es kommt die sogenannte *Eigenschaft-Element-Syntax* ins Spiel. Bei dieser Schreibweise wird zuerst der Typ des Elements genannt und dahinter, durch einen Punkt getrennt, die Eigenschaft.

```
<Button Height="100" Width="200" Foreground="White">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="Black" Offset="0.0" />
      <GradientStop Color="LightGray" Offset="1.0" />
    </LinearGradientBrush>
  </Button.Background>
    Mein erster Button
</Button>
```

Listing 22.10 Button mit Farbverlauf

Das Codefragment beschreibt mit einem Objekt vom Typ `LinearGradientBrush` den linearen Farbverlauf des Hintergrunds einer Schaltfläche. Die beiden `GradientStop`-Objekte geben Position und Farbe des Farbverlaufs an.

Abbildung 22.7 Button mit linearem Farbverlauf

Die Eigenschaft-Element-Schreibweise können Sie auch bei einfachen Eigenschaften wie zum Beispiel `Height` und `Width` verwenden. Sie geben dann den gewünschten Wert zwischen den beiden Tags an, z. B.:


```
<Button>
  <Button.Height>50</Button.Height>
  <Button.Width>100</Button.Width>
  Mein erster Button
</Button>
```

Listing 22.11 Eigenschaft-Element-Syntax elementarer Properties

22.2.5 Inhaltseigenschaften

Sie kennen nun die *Attribut-Schreibweise* und die *Eigenschaft-Element-Syntax*. Doch wie verhält es sich mit der Beschriftung einer Schaltfläche? Diese ist tatsächlich ein Sonderfall unter den Eigenschaften, da die Beschriftung auf zweierlei Art und Weise vorgenommen werden kann. Geben Sie dazu die Eigenschaft `Content` entweder als Attribut des `<Button>`-Tags an oder zwischen dem ein- und ausleitendem Element. Sie schreiben also entweder

```
<Button Content="Mein erster Button" />
```

oder

```
<Button>Mein erster Button</Button>
```

Beide Varianten sind gleichwertig. Es steht nun die Frage im Raum, warum wir in der letztgenannten Form nicht ausdrücklich die Eigenschaft angeben müssen, also:

```
<Button>
  <Button.Content>
    Mein erster Button
  </Button.Content>
</Button>
```

Die Antwort lautet: Weil `Content` als sogenannte *Inhaltseigenschaft* für das `Button`-Element definiert ist. Bei einer Inhaltseigenschaft können Sie auf die explizite Angabe der Eigenschaft in der Eigenschaft-Element-Schreibweise verzichten. Bei einer Schaltfläche ist die Eigenschaft `Content` gleichzeitig die Inhaltseigenschaft, weil das Steuerelement `Button` von der Basis `ContentControl` abgeleitet ist. Hintergrund dabei ist, dass `ContentControl` mit dem Attribut `ContentPropertyAttribute` verknüpft ist, mit dem die Inhaltseigenschaft aller ableitenden Komponenten angegeben wird:

```
[ContentProperty("Content")]
public class ContentControl : Control, IAddChild { [...] }
```

Das Besondere an der Inhaltseigenschaft `Content` ist ihr Datentyp `Object`. Dadurch lässt sich der Inhaltseigenschaft des betreffenden Elements, also auch dem `Button`, ein beliebiges Objekt zuordnen, beispielsweise ein `Rectangle`-Element, wie in Listing 22.12 gezeigt:

```
<Button Height="100" Width="200" Background="LightGray">
  <Rectangle Height="30" Width="100" Fill="DarkBlue" />
</Button>
```

Listing 22.12 Eigenschaft »Content« einer Schaltfläche beschreibt ein »Rectangle«.

Verschachteln von Elementen

Natürlich muss es sich bei dem eingebetteten Element nicht unbedingt um ein grafisches Element handeln, es kann auch ein Steuerelement sein, zum Beispiel eine `ListBox`:

```
<Button Height="100" Width="200">
  <ListBox Width="80">
    <ListBoxItem>Niederlande</ListBoxItem>
    <ListBoxItem>Belgien</ListBoxItem>
    <ListBoxItem>Frankreich</ListBoxItem>
  </ListBox>
</Button>
```

Listing 22.13 Das Steuerelement »ListBox« mit mehreren eingebetteten Elementen

Wollen Sie mehrere Elemente der Schaltfläche unterordnen, beispielsweise ein `Label`, eine `ListBox` und darüber hinaus ein `Image`, scheint das im ersten Moment nicht möglich zu sein, weil die `Content`-Eigenschaft nur ein Element zulässt. Dennoch ist die Lösung ganz simpel: Wählen Sie als dem `Button` untergeordnetes Element eines der Containersteuerelemente (z. B. ein `Grid`), in dem Sie die gewünschten Steuerelemente entsprechend positionieren.

Klassen mit beliebigen Inhalten

`Content` ist die Inhaltseigenschaft aller Steuerelemente, die von `ContentControl` abgeleitet sind. Dazu gehört die Klasse `Button`. In der WPF sind insgesamt sogar vier Klassen definiert, die eine Inhaltseigenschaft vorschreiben. Jede dieser Basisklassen zwingt ihren Ableitungen eine bestimmte Charakteristik hinsichtlich der Inhaltsbeschreibung auf. Bei einer `ListBox` sind es zum Beispiel gleich mehrere Elemente, die angegeben werden können, ohne dass die entsprechende Inhaltseigenschaft `Items` angegeben werden muss:

```
<ListBox>
  <ListBoxItem>Freitag</ListBoxItem>
  <ListBoxItem>Samstag</ListBoxItem>
  <ListBoxItem>Sonntag</ListBoxItem>
</ListBox>
```

Listing 22.14 »ListBox«-Control mit mehreren Einträgen

Tabelle 22.1 können Sie die vier Klassen entnehmen, die eine Inhaltseigenschaft vorschreiben.

Klasse	Beschreibung
`ContentControl`	Legt die Eigenschaft `Content` als Inhaltseigenschaft fest. Diese Eigenschaft kann nur ein Element eines beliebigen Typs beschreiben (z. B. `Button`, `CheckBox`, `Label`).
`HeaderedContentControl`	Diese Klasse ist selbst von `ContentControl` abgeleitet und ergänzt die Fähigkeit der Inhaltseigenschaft um die Möglichkeit eines beschreibenden Headers (z. B. `Expander`, `GroupBox`, `TabItem`).
`ItemsControl`	Legt die Eigenschaft `Items` als Inhaltseigenschaft fest. `Items` kann eine Auflistung mehrerer Elemente beschreiben (z. B. `ListBox`, `ComboBox`, `Menu`).
`HeaderedItemsControl`	Diese von `ItemsControl` abgeleitete Klasse dient als Basisklasse aller Steuerelemente, die mehrere Elemente enthalten, die durch genau einen Header beschrieben werden (z. B. `MenuItem`, `ToolBar`).

Tabelle 22.1 Klassen, die Inhaltseigenschaften beschreiben

Klassen, die von »UIElement« abgeleitet sind

Einen Sonderfall hinsichtlich der Inhaltseigenschaften bilden die Klassen, die selbst Steuerelemente enthalten und verwalten. Dabei handelt es sich um alle Layoutcontainer wie beispielsweise das `Grid`, das `StackPanel` oder das `DockPanel`. An dieser Stelle sei im Zusammenhang mit den Inhaltseigenschaften schon das Folgende verraten: Alle diese Klassen leiten sich von der Basis `Panel` ab, in der die Eigenschaft `Children` als Inhaltseigenschaft festgelegt ist. Die Eigenschaft `Children` selbst beschreibt eine `UIElementCollection`, in der Objekte verwaltet werden, die sich auf den Typ `UIElement` zurückführen lassen. Dazu gehören alle Steuerelemente.

Sehen wir uns nun am Beispiel des `StackPanel`-Objekts an, wie Sie der Inhaltseigenschaft `Children` Elemente hinzufügen können:

```
<StackPanel>
  <Button>OK</Button>
  <Button>Übernehmen</Button>
  <Button>Abbrechen</Button>
</StackPanel>
```

Listing 22.15 Hinzufügen von Steuerelementen zur Eigenschaft »Children« in XAML

Entwickeln Sie eine WPF-Anwendung, wird Ihr Schwerpunkt die XAML-Codierung sein. Trotzdem können Sie alles, was Sie im XAML-Code schreiben, auch mit C#-Code erreichen.

Um beispielsweise den XAML-Code aus Listing 22.15 durch C#-Code abzubilden, sind die folgenden Anweisungen notwendig:

```
StackPanel stackPanel = new StackPanel();
Button btnOK = new Button();
btnOK.Content = "OK";
Button btnUebernehmen = new Button();
btnUebernehmen.Content = "Übernehmen";
Button btnAbbrechen = new Button();
btnAbbrechen.Content = "Abbrechen";
stackPanel.Children.Add(btnOK);
stackPanel.Children.Add(btnUebernehmen);
stackPanel.Children.Add(btnAbbrechen);
this.AddChild(stackPanel);
```

Listing 22.16 Die Elemente aus Listing 22.15 durch C#-Code erzeugt

Dieses Beispiel zeigt sehr eindrucksvoll, dass XAML deutlich kürzer und kompakter ist als die Beschreibung der Oberfläche mit Programmcode.

22.2.6 Typkonvertierung

Die im XAML-Code definierten Elemente entsprechen einer Klasse, die Attribute einer Eigenschaft. Dabei wird den Attributen der Wert immer als Zeichenfolge übergeben. Die meisten Eigenschaften sind aber nicht vom Datentyp `String` und müssen daher in den tatsächlichen Datentyp konvertiert werden. Betrachten wir dazu das einfache Beispiel der Eigenschaften `Width` und `Height` eines Buttons:

```
<Button Height="100" Width="200">Beenden</Button>
```

Beide Eigenschaften sind vom Typ `Double`. Das bedeutet, dass die angegebenen Werte in die Fließkommazahlen 100,0 bzw. 200,0 konvertiert werden müssen. In der WPF sind zahlreiche Typkonvertierungen vordefiniert. Ein komplexer Fall liegt vor, wenn wir der Eigenschaft `Background` eine Farbe übergeben:

```
<Button Background="Blue" />
```

Tatsächlich ist die Eigenschaft `Background` vom Typ `Brush`. Da die Klasse `Brush` abstrakt definiert ist, muss sie in eine ihrer Ableitungen konvertiert werden. Solange wir es mit einer monochromen Farbe zu tun haben, wird es eine Konvertierung in den Typ `SolidColorBrush` sein. Das wird deutlich, wenn Sie sich die entsprechende Eigenschaft-Element-Schreibweise ansehen:


```
<Button>
  <Button.Background>
    <SolidColorBrush Color="Blue" />
  </Button.Background>
</Button>
```

Hier muss natürlich noch der Farbwert `Blue` konvertiert werden, da der Typ einer Farbe nicht `String`, sondern `Color` ist.

Die Technik der Typkonvertierung wird in XAML auch benutzt, um mehrere ähnliche Eigenschaften zusammenzufassen und auf diese Weise den XAML-Code kompakter zu gestalten. `Margin` gehört zu dieser Gruppe von Eigenschaften. `Margin` ist vom Datentyp `Thickness` und legt den äußeren Rand eines Elements fest. In der Eigenschaft-Element-Schreibweise können Sie `Margin` wie folgt in einem `StackPanel` einsetzen:

```
<StackPanel>
  <StackPanel.Margin>
    <Thickness Left="100" Top="30" Right="50" Bottom="10" />
  </StackPanel.Margin>
  <Button>Button1</Button>
</StackPanel>
```

Listing 22.17 Festlegen der Eigenschaft »Margin« (aufwendige Schreibweise)

Typkonvertierung ist hierbei noch nicht im Spiel. Sie können aber die Eigenschaft `Margin` auch wie folgt festlegen:

```
<StackPanel Margin="100, 30, 50, 10">
  <Button>Button1</Button>
</StackPanel>
```

Eine andere Variante gestattet es sogar, auf die Kommata zu verzichten:

```
<StackPanel Margin="100 30 50 10">
  <Button>Button1</Button>
</StackPanel>
```

Diese Form des Einsatzes von `Margin` setzt voraus, dass die Zeichenfolge von einem Typkonvertierer passend umgesetzt wird (was natürlich der Fall ist). Selbstverständlich ist die Eigenschaft mit der erforderlichen Verhaltensweise ausgestattet, die Werte entsprechend zu verarbeiten.

22.2.7 Markup-Erweiterungen (Markup Extensions)

Die Eigenschaften eines Elements werden in XAML weitestgehend durch Attribute beschrieben, die als Zeichenfolge angegeben und passend ausgewertet werden. Dieses Konzept wird

auch in XML benutzt, hat aber seine Grenzen, wenn ein Eigenschaftswert durch einen Objektverweis beschrieben werden muss. An dieser Stelle kommen Markup-Erweiterungen ins Spiel, die uns das ermöglichen.

Im Grunde genommen stellen auch Typkonvertierer einen Weg dar, ein Objekt an eine Eigenschaft zu binden. Der Unterschied zwischen einem Typkonvertierer und einer Markup-Erweiterung ist jedoch, dass Typkonvertierer nach einer festgelegten Regel arbeiten und im Hintergrund agieren, während Markup-Erweiterungen allgemein verwendbar sind.

Markup-Erweiterungen sind Attributwerte, die in geschweiften Klammen eingeschlossen angegeben sind. Im Beispielcode von Listing 22.18 wird das Konzept der Markup-Erweiterung dazu benutzt, den Inhalt der Eigenschaft `Text` der `TextBox` *txtUnten* an die Eigenschaft `Text` der `TextBox` *txtOben* zu binden. Das hat zur Folge, dass zur Laufzeit eine Eingabe in der oberen `TextBox` sofort von der unteren `TextBox` übernommen wird.

```
<StackPanel>
  <TextBox Name="txtOben" />
  <TextBox Name="txtUnten" Text="{Binding ElementName=txtOben, Path=Text}" />
</StackPanel>
```

Listing 22.18 Die Eigenschaft »Text« mit einer Markup-Erweiterung

Mit den geschweiften Klammern geben wir an, dass der Attributwert weder ein Literal noch ein über einen Typkonvertierer umwandelbarer Wert ist. Die Klasse innerhalb der Markup-Erweiterung ist in diesem Beispiel `Binding`, die zum Namespace `System.Windows.Data` gehört. Der Parameter `ElementName` gibt das Element an, an das gebunden wird; `Path` beschreibt die Eigenschaft des Quellelements, aus der der Wert bezogen werden soll. Beachten Sie, dass Sie zwischen den Parametern ein Komma setzen müssen.

Hinweis

Eine Steuerelementeigenschaft, die mit einer Markup Extension an eine andere Eigenschaft mit `Binding` gebunden wird, muss als Abhängigkeitseigenschaft (Dependency Property) implementiert sein.

Werden den Parametern einer Markup-Erweiterung mit dem Zuweisungsoperator `=` Werte übergeben (wie `ElementName` und `Path` in Listing 22.18), wird die Markup-Erweiterung `Binding` mit dem parameterlosen Konstruktor instanziiert. Die Parameterwerte für `ElementName` und `Path` werden den gleichnamigen Eigenschaften des `Binding`-Objekts übergeben.

Enthalten die Parameter hingegen kein `=`-Zeichen, werden sie an einen parametrisierten Konstruktor weitergeleitet. Selbstverständlich müssen dann Anzahl und Typ mit denen des Konstruktors übereinstimmen. Da die Klasse `Binding` einen Konstruktor beschreibt, der

einen `String` für die Eigenschaft `Path` entgegennehmen kann, wäre auch die folgende Markup Extension zulässig:

```
<TextBox Name="txtUnten" Text="{Binding Text, ElementName=txtOben}" />
```

Markup-Erweiterungen werden nicht nur durch den Typ `Binding` beschrieben. `StaticResource`, `DynamicResource` oder auch `x:` sind weitere wichtige Erweiterungen, die uns noch beschäftigen werden. Nicht unerwähnt bleiben sollte auch, dass das enorme Potential der WPF hinsichtlich der Datenbindung erst durch Markup-Erweiterungen voll ausgeschöpft wird.

Die Eigenschaft-Element-Schreibweise ist auch im Zusammenhang mit Markup-Erweiterungen möglich. Listing 22.19 zeigt das anhand des Beispiels aus Listing 22.18:

```
<StackPanel>
  <TextBox Name="txtOben"></TextBox>
  <TextBox Name="txtUnten">
    <TextBox.Text>
      <Binding ElementName="txtOben" Path="Text" />
    </TextBox.Text>
  </TextBox>
</StackPanel>
```

Listing 22.19 Markup-Erweiterung mit der Eigenschaft-Element-Syntax

Später werden Sie noch sehen, dass zudem mehrfach verschachtelte Markup-Erweiterungen möglich sind.

Markup-Erweiterungen durch C#-Code beschreiben

Markup-Erweiterungen lassen sich nicht nur deklarativ im XAML-Code festlegen, sondern auch durch Programmcode beschreiben. Das sehen Sie im folgenden Programmcode:

```
public MainWindow()
{
  InitializeComponent();
  // StackPanel erstellen
  StackPanel panel = new StackPanel();
  this.AddChild(panel);
  // TextBox oben erstellen
  TextBox txtOben = new TextBox();
  panel.Children.Add(txtOben);
  // TextBox unten erstellen
  TextBox txtUnten = new TextBox();
  panel.Children.Add(txtUnten);
  // Bindung erzeugen
  Binding binding = new Binding("Text");
```

```
  binding.Source = txtOben;
  txtUnten.SetBinding(TextBox.TextProperty, binding);
}
```

Listing 22.20 Bindung mittels Programmcode

Anmerkung

Dieser Programmcode kann nur dann fehlerfrei ausgeführt werden, wenn das Fenster im XAML-Code keinen Layoutcontainer enthält.

Der Code ist im Konstruktor des Fensters nach dem Aufruf der Methode `InitializeComponent` implementiert. Zuerst wird das `StackPanel`-Objekt erzeugt und dem `Window` durch Aufruf von `AddChild` als untergeordnetes Element übergeben. Die beiden Textboxen werden nach der Instanziierung zu untergeordneten Elementen des `StackPanel`-Elements. Dieser Container liefert durch Aufruf der Eigenschaft `Children` die Referenz auf ein `UIElementCollection`-Objekt, dem mit der Methode `Add` die Textboxen hinzugefügt werden.

Die Bindung der unteren an die obere `TextBox` erfordert ein `Binding`-Objekt, dessen Konstruktor Sie die Eigenschaft des Elements bekanntgeben, an die gebunden werden soll. Das Element, an das gebunden wird, geben Sie mit der Eigenschaft `Source` an.

Aktiviert wird die Bindung der unteren an die obere `TextBox` mit der Methode `SetBinding`. Sehen Sie sich bitte noch einmal genau den Aufruf dieser Methode und hier insbesondere die Argumente an. Während das zweite übergebene Argument keiner besonderen Erklärung bedarf, erscheint das erste mit `TextBox.TextProperty` ungewöhnlich. Aber genau so wird eine Abhängigkeitseigenschaft abgerufen.

22.2.8 XML-Namespaces

Bei einem XML-Namespace handelt es sich um eine Vorschrift, um Elemente im XAML-Code eindeutig zuordnen und interpretieren zu können. Beispielsweise könnte das Element `<Button>` in einem XAML-Dokument zwei unterschiedliche Schaltflächen, also Klassen, beschreiben. Erst die Zuordnung zu einem Namespace gestattet die eindeutige Identifizierbarkeit. Prinzipiell kommt den XML-Namespaces somit die gleiche Bedeutung zu wie den CLR-Namespaces.

Im Wurzelelement `Window` einer XAML-Datei werden bereits beim Anlegen eines Fensters mehrere Namespaces verfügbar gemacht:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```



```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfApp1"
```

Listing 22.21 Standardvorgabe der XML-Namespaces

Namespaces werden mit dem Attribut `xmlns` eingeleitet. Optional können Sie dahinter, durch einen Doppelpunkt getrennt, ein Namespace-Präfix angeben. Dem wird meist ein URI zugeordnet.

Sehen wir uns die beiden ersten Namespace-Angaben im einleitenden `Window`-Element an. Der erste weist kein Präfix auf und ist der sogenannte *Standard-Namespace*. Alle diesem Namespace zugeordneten Elemente werden ohne Präfixangabe in XAML verwendet. Dazu gehören beispielsweise `<Grid>`, `<Button>` oder `<TextBox>`.

Da nur ein XML-Namespace ohne Präfix als Standard-Namespace angegeben werden darf, müssen alle Elemente, die nicht dem Standard-Namespace zugeordnet werden, ein Präfix aufweisen. Die zweite Namespace-Angabe definiert ein solches mit `x`. Dieser Namespace dient den XAML-Spracherweiterungen. Verwenden Sie ein Element dieses Namespace, müssen Sie vor dem Element das Namespace-Präfix, gefolgt von einem Doppelpunkt, angeben. Ein gutes Beispiel liefert bereits die XAML-Struktur eines Fensters mit

```
<Window x:Class="WpfApp1.MainWindow" ...>
```

Die Namespaces, die durch die Präfixe `d` und `mc` beschrieben werden, haben für uns keine wesentliche Bedeutung. Ganz anders aber der mit `local` beschriebene Namespace. Damit wird von Anfang der CLR-Namespace bekanntgegeben, dem die Anwendung zugeordnet ist. Dieser Namespace ist immer dann notwendig, wenn innerhalb des XAML-Codes eine Klasse der aktuellen Anwendung genutzt werden soll.

Hinweis

Von Hause aus ist XAML-Code dumm. Sie müssen über XML-Namespaces wirklich alles und jeden Typ bekannt machen. Das bezieht auch die elementaren Datentypen ein. Angenommen, Sie möchten eine Integer-Ressource erzeugen. Ohne die Angabe von

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

werden Sie scheitern, weil XAML-Code auch nicht die *mscorlib.dll* kennt, in der unter anderem die elementaren Datentypen enthalten sind.

CLR-Namespaces verwenden

Viele WPF-Anwendungen benötigen über die Standardvorgaben hinaus weitere Namespaces. Dabei kann es sich durchaus auch um CLR-Namespaces handeln. Angenommen, wir möchten in einer WPF-Anwendung mit XAML auf die Klasse `Circle` der Assembly *GeometricObjectsSolution.dll* zugreifen. Nachdem Sie in der WPF-Anwendung einen Verweis auf die

Klassenbibliothek gelegt haben, geben Sie den Namespace im Wurzelelement `Window` wie folgt an:

```
<Window x:Class="WpfApplication1.Window1"
  [...]
  xmlns:geo="clr-namespace:GeometricObjects;assembly=GeometricObjectsSolution"
  Title="Window1" Height="300" Width="300">
</Window>
```

Das Präfix, hier `geo`, ist frei wählbar, muss aber in der aktuellen XAML-Datei eindeutig sein. Danach folgen zwei Name-Wert-Paare, die durch ein Semikolon voneinander getrennt sind. Das erste Paar wird mit `clr-namespace` eingeleitet. Dahinter folgt ein Doppelpunkt und anschließend der CLR-Namespace, dem die Klasse `Circle` zugeordnet ist. Das zweite Name-Wert-Paar wird mit `assembly` eingeleitet. Nach dem `=`-Zeichen wird die Assembly angegeben, jedoch ohne ihre Dateiendung *.dll*.

Jetzt können Sie die Klasse `Circle` im XAML-Code verwenden und den Eigenschaften die gewünschten Werte übergeben. Der Elementangabe müssen Sie dabei das gewählte Präfix voranstellen, hier also `geo`.

```
<geo:Circle x:Kex="kreis" Radius="77" XCoordinate="100" YCoordinate="-250" />
```

Anmerkung

Auf das Erzeugen eines Objekts im XAML-Code möchte ich an dieser Stelle nicht weiter eingehen. Damit werden wir uns später noch sehr ausführlich beschäftigen.

Etwas einfacher ist die Bekanntgabe des zum aktuellen Projekt gehörenden Namespace, der bekanntlich das Präfix `local` hat. Hier kann auf die Angabe von `assembly` verzichtet werden.

Mehrere CLR-Namespaces zusammenfassen

Entwickeln Sie Klassenbibliotheken, sollten Sie daran denken, die Klassen für den Einsatz in XAML vorzubereiten. Dazu gehört die Berücksichtigung der Namespaces. Obwohl sich im XAML-Code auch CLR-Namespaces angeben lassen, sollte man der üblichen W3C-konformen Notation, also die Angabe eines URLs/URIs, den Vorzug geben.

Das gilt insbesondere dann, wenn in einer Klassenbibliothek mehrere CLR-Namespaces definiert sind. Nehmen wir exemplarisch an, die beiden Klassen `Rectangle` und `Circle` seien unterschiedlichen CLR-Namespaces zugeordnet.

```
namespace Namespace1 { public class Circle {[...]} }
namespace Namespace2 { public class Rectangle {[...]} }
```


Ohne weitere Maßnahmen zu ergreifen, können Sie beide Typen im XAML-Code nutzen, wenn Sie beide Namespaces im Wurzelelement angeben, z. B.:

```
xmlns:geo1="clr-namespace:Namespace1;assembly=GeometricObjectsSolution"
xmlns:geo2="clr-namespace:Namespace2;assembly=GeometricObjectsSolution"
```

Beide Namespaces lassen sich auf einen gemeinsamen XML-Namespace abbilden. Dazu muss die entsprechende Vorkehrung bereits in der Klassenbibliothek erfolgen. Die notwendigen Angaben sind in der Datei *AssemblyInfo.cs* der Klassenbibliothek zu machen. Ergänzen Sie die Datei dazu wie nachfolgend gezeigt um zwei Attributangaben:

```
[assembly: XmlnsDefinition("http://www.tollsoft.de", "Namespace1")]
[assembly: XmlnsDefinition("http://www.tollsoft.de", "Namespace2")]
```

Um auf das Attribut `XmlnsDefinition` zugreifen zu können, müssen Sie zuerst einen Verweis auf die Bibliothek *System.Xaml.dll* legen und den Namespace `System.Windows.Markup` mit `using` bekanntgeben.

Das Attribut beschreibt zwei Parameter. Dem ersten übergeben Sie den gewünschten XML-Namespace, dem zweiten Parameter teilen Sie mit, welcher CLR-Namespace auf diesen XML-Namespace abgebildet werden soll. Der XAML-Code reduziert sich daraufhin auf die folgende Angabe im Wurzelelement:

```
xmlns:geo="http://www.tollsoft.de"
```

Sie können anschließend mit dem Präfix `geo` Elemente sowohl vom Typ `Circle` als auch vom Typ `Rectangle` in Ihren XAML-Code einbetten, z. B.:

```
<geo:Circle ... />
```

22.2.9 XAML-Spracherweiterungen

XAML definiert eine Reihe von Attributen, die besondere Aspekte bei der Entwicklung berücksichtigen und keine Entsprechungen in einer Klasse besitzen. Hiermit werden nur Zusatzinformationen geliefert, die einer besonderen Verarbeitung bedürfen. Tabelle 22.2 stellt Ihnen einige davon vor. In den folgenden Kapiteln werden Sie in den Beispielen auf einige dieser Schlüsselwörter stoßen.

Schlüsselwort	Bedeutung
`x:Class`	Dieses Attribut stellt die Beziehung zwischen dem Wurzelelement im XAML-Code und der Code-Behind-Datei her.
`x:Code`	Die Trennung von Code und Oberflächenbeschreibung ist keine strikte Vorgabe. Sie können auch Code innerhalb einer XAML-Datei implementieren. Mit `x:Code` wird ein Codebereich im XAML-Code definiert.

Tabelle 22.2 Schlüsselwörter von XAML (Auszug)

Schlüsselwort	Bedeutung
x:Key	Gibt den eindeutigen Namen eines Elements in einer Ressource an.
x:Name	Mit diesem Attribut können Sie einem Element einen Namen geben, wenn das Element selbst nicht über eine Eigenschaft Name verfügt.

Tabelle 22.2 Schlüsselwörter von XAML (Auszug) (Forts.)

Markup-Erweiterungen

Es gibt mehrere Markup-Erweiterungen, die nicht spezifisch für die WPF-Anwendung sind, sondern Implementierungen für Funktionen von XAML als Sprache sind. Auch diese Markup-Erweiterungen sind durch das x:-Präfix in der Verwendung identifizierbar.

Erweiterung	Beschreibung

x:Array	Hiermit lassen sich Arrays in XAML definieren. Beispiel:
	`<x:Array Type="clr:Int32" x:Key="intListe" >`
	`  <clr:Int32>36</clr:Int32>`
	`  <clr:Int32>1270</clr:Int32>`
	`  <clr:Int32>5</clr:Int32>`
	`</x:Array>`
	Beachten Sie bitte, dass für die Verwendung des Integers der CLR-Namespace System bekanntgegeben werden muss. Im Beispiel habe dazu das Präfix clr verwendet.

x:Null	Wird verwendet, um einem Element null zuzuweisen. Beispiel: `<Button Background="{x:Null}" />`
x:Static	Referenziert eine statische Variable oder Eigenschaft eines Objekts oder eine Konstante oder einen Enumerationswert. Beispiel: `<Button Background="{x:Static Brushes.Red}" />`
x:Type	Wird beispielsweise in Stildefinitionen benutzt, um einen Typ anzugeben. Beispiel: `<Style TargetType="{x:Type TextBox}" />`

Tabelle 22.3 Markup-Erweiterungen von XAML

22.2.10 Die Direktive »#region« nutzen

Im C#-Code können Sie mit der Direktiven #region-#endregion Ihren Programmcode übersichtlicher gestalten und bestimmte Codeabschnitte mit Hilfe von Markern am linken Rand des Code-Editors reduzieren oder erweitern.

Auch im XAML-Code können Sie von diesem Hilfsmittel profitieren. Die Syntax ähnelt der im C#-Code, muss aber, wie in Listing 22.22 gezeigt, in spitzen Klammern geschrieben werden.

```
<!--#region Beschreibung-->
<Button />
<ListBox />
<!--#endregion-->
```

Listing 22.22 »#region« im XAML-Code

#region-Bereiche können eine große Hilfe sein, den XAML-Code, der bei aufwendigen Oberflächen schnell sehr unübersichtlich wird, optisch besser zu strukturieren und damit lesbarer zu machen.