C# – first guide ISBN 3-8272-5936-3

Danksagung

Was würden Sie erwarten, wenn Ihnen ein Verleger per E-Mail mitteilt, dass er sich mit Ihnen über ein wirklich spannendes Buchprojekt unterhalten will? Als Chris Webb mich schließlich anrief, hatte ich alles Mögliche erwartet, nur nicht dieses Buch. Sein Angebot klang zu verlockend, um es ablehnen zu können: Wer hat schon die Chance, ein Buch über eine wirklich neue Programmiersprache zu schreiben?

Mein Dank geht natürlich an Chris Webb und Brad Jones, dafür, dass sie bei dem knapp bemessenen Zeitrahmen ihr Vertrauen in mich gesetzt haben. Beide, aber auch mein Lektor, Kevin Howard, mussten sich mit meinen konzeptuellen Änderungen im Inhalt abfinden, als das Buch zunehmend Gestalt annahm – man kann alte Gewohnheiten eben nicht ablegen.

Speziell danken möchte ich Christian Koller, dafür dass er meine Kapitel inhaltlich gegengelesen und mich darauf hingewiesen hat, wenn wieder einmal Details fehlten, die sich ohne weitere Erläuterung nur C++-ProgrammiererInnen erschließen. Obwohl ich bei Sams Publishing nicht jeden persönlich kenne, der an diesem Buch beteiligt war, danke ich trotzdem allen, die ihren Teil dazu beitragen haben, dass es letztlich seinen Weg in die Regale gefunden hat – trotz des engen Zeitrahmens.

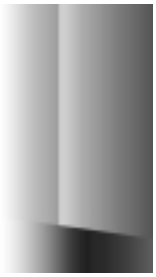
Darüber hinaus gilt mein letzter und wichtigster Dank meiner Familie für ihre unablässige Unterstützung während all meiner Buchprojekte.

Über den Autor

Christoph Wille, MCSE, MCSD, CNA und MCP-IT, arbeitet als Netzwerkberater und Programmierer speziell für Windows DNA. Von Microsoft als »Most Valuable Professional« (MVP) für den Bereich Active Server Pages eingestuft, zählt er zu den wenigen Entwicklern, die mit Microsoft zusammen an den ersten Versionen der Sprache C# gearbeitet haben.

Christoph Wille ist Autor oder Coautor verschiedener Bücher, darunter: Sams Teach Yourself ADO 2.5 in 21 Days, Sams Teach Yourself Active Server Pages in 24 Hours, MCSE Training Guide: SQL Server 7 Administration und Sams Teach Yourself MCSE TCP/IP in 14 Days.

Einführung



Dieses Buch ist Ihre Fahrkarte für den schnellen Einstieg in die neue komponentenorientierte Programmiersprache C# (gesprochen: C-Sharp), die Microsoft dem Produkt *Next Generation Windows Services Runtime* beilegt.

Bei dem Produkt, das hier als NGWS-Subsystem oder -Laufzeitumgebung bezeichnet wird, handelt es sich um eine Laufzeitumgebung, die nicht nur die Ausführung von Code verwaltet, sondern auch eine Reihe von Diensten bereitstellt, um die Programmierung zu vereinfachen. Ein Compiler, dessen Code in der NGWS-Laufzeitumgebung ausführbar sein soll, muss sogenannten *verwalteten Code* produzieren. Gewissermaßen im Gegenzug gibt es dafür die sprachübergreifende Ausnahmebehandlung sowie Komponentenintegration, verbesserte Sicherheit, Versionskontrolle und Profiling- sowie Debugging-Dienste.

C# ist die erste Sprache, die speziell für die NGWS-Laufzeitumgebung entwickelt wurde. Ja, ein guter Teil des gesamten NGWS-Subsystems ist bereits in C# geschrieben. Der C#-Compiler dürfte daher auch der am besten getestete und optimierte Compiler sein, der dem NGWS beigepackt ist. C# bezieht zwar einen großen Teil seiner Konzepte von C++, ist aber moderner und bietet wesentlich mehr Typensicherheit. Mit anderen Worten, die Sprache hat die besten Voraussetzungen, das Mittel der Wahl für die Implementation von Unternehmenslösungen zu werden.

Wer sollte dieses Buch lesen?

Wenn Sie gerade dabei sind, die Programmierung zu erlernen, ist dieses Buch wahrscheinlich eher nichts für Sie. Es wendet sich vielmehr an ProgrammiererInnen, die einen schnellen Überblick über die Möglichkeiten der Sprache erhalten wollen und dafür bereits das nötige Vorverständnis von anderen Programmiersprachen (vorzugsweise von C, C++ oder Java, aber auch Visual Basic) her mitbringen.

Am leichtesten wird sich Ihnen C# erschließen, wenn Sie C++ als Hintergrund haben; aber auch wenn Sie in einer anderen Sprache flüssig programmieren können, wird Ihnen das Buch die wichtigsten Aspekte und Konzepte von C# vermitteln. Am meisten werden Sie von dem Buch profitieren, wenn Sie auch ein wenig Ahnung von der COM-Programmierung haben – obwohl die COM-Programmierung beileibe nicht als Voraussetzung für ein generelles Verständnis der Sprache und des Buchs zu sehen ist.

Aufbau des Buchs

Das Buch umfasst zwölf Kapitel. Im Folgenden eine kurze Übersicht, was Sie in den einzelnen Kapiteln erwartet:

- Kapitel 1, »Einführung in C#« – nimmt Sie auf einen kurzen Rundgang durch die Sprache mit und beantwortet die Fragen, wann und warum es sich lohnt, C# zu erlernen.
- Kapitel 2, »Der Unterbau – das NGWS-Laufzeitsystem« – stellt Ihnen das Drumherum sowie die NGWS-Laufzeitumgebung als Infrastruktur für die Ausführung von C#-Code vor.
- Kapitel 3, »Die erste C#-Anwendung« – in diesem Kapitel lernen Sie Ihre erste C#-Anwendung kennen, sie trägt (wie sollte es anders sein) den Namen »Hello World«.
- Kapitel 4, »Die Typen von C#« – das Typensystem von C# ist reichhaltig; in diesem Kapitel lernen Sie neben den elementaren Datentypen der Sprache den wichtigen Unterschied zwischen wertbehafteten Typen und Referenztypen sowie den Mechanismus des *Boxing* und *Unboxing* kennen.
- Kapitel 5, »Klassen« – die in diesem Kapitel vorgestellte objektorientierte Programmierung mit Klassen vermittelt Ihnen einen Hauch der gewaltigen Ausdruckskraft der Sprache. Hier kommen Sie mit Konstruktoren, Destruktoren, Methoden, Eigenschaften, Indizierern und Ereignissen in Kontakt.
- Kapitel 6, »Kontrollstrukturen« – die Logik eines Programms spiegelt sich im Fluss der Anweisungen wieder. In diesem Kapitel lernen Sie die Mittel kennen, um in C# Fallunterscheidungen und Schleifen zu formulieren.
- Kapitel 7, »Ausnahmen behandeln« – in diesem Kapitel erfahren Sie, wie Sie in Ihrem Code Ausnahmen behandeln und selbst signalisieren, damit sich Ihr Programm in der NGWS-Welt als »gesellschaftstauglich« erweist.
- Kapitel 8, »Komponenten mit C# schreiben« – zeigt Ihnen, wie Sie Ihre eigenen Komponenten in C# für die sprachübergreifende Nutzung im Rahmenwerk des NGWS implementieren.
- Kapitel 9, »Konfiguration und Verbreitung« – hier erfahren Sie, was es in C# mit der bedingten Kompilierung auf sich hat und wie Sie Ihren Code auf der Basis von Kommentaren geeignet dokumentieren, damit der Compiler daraus eine Dokumentation im XML-Format erzeugen kann. Darüber hinaus stellt Ihnen dieses Kapitel das Konzept der Versionskontrolle der NGWS-Laufzeitumgebung vor.

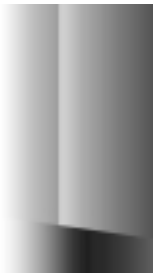
- Kapitel 10, »Integration von nicht verwaltetem Code« – auf die NGWS-Laufzeitumgebung angewiesen zu sein, heißt nicht, auf den Einsatz von existierendem (nicht verwaltetem) Code verzichten zu müssen. Dieses Kapitel zeigt, wie das Zusammenspiel funktioniert.
- Kapitel 11, »C#-Code debuggen« – stellt Ihnen den Einsatz und die Möglichkeiten des SDK-Debuggers für die Fehlersuche in C#-Anwendungen vor.
- Kapitel 12, »Sicherheit« – das NGWS-Laufzeitsystem verfügt über ein zeitgemäßes Sicherheitskonzept für die Ausführung von Code, der unterschiedlichen Sicherheitsanforderungen genügt. Hier erfahren Sie, wie »Codezugriffsrechte« funktionieren und was »rollenbasierte Sicherheit« ist.

Was brauchen Sie für die Arbeit mit diesem Buch?

Um die Beispiele aus diesem Buch auf praktischer Ebene nachzuvollziehen, benötigen Sie das *Next Generation Windows Services* (NGWS) *Software Development Kit* (SDK) und eine Maschine, auf der (mindestens) Windows 2000 installiert ist. Obwohl Sie vom Prinzip her bereits mit der Kombination NGWS-Laufzeitumgebung und C#-Compiler auskommen, empfiehlt es sich doch zum Ausloten der Möglichkeiten dieser neuen Technologie, das gesamte SDK mit allen Werkzeugen (insbesondere den Debugger) sowie der Sprachdokumentation zu installieren.

Das Buch verlangt nicht, dass auf Ihrer Maschine irgendwelche Werkzeuge aus Visual Studio 7 installiert sein müssen. Die einzige Empfehlung in dieser Richtung ist der Quelltext-Editor; andere Editoren mit einer gewissen Basisfunktionalität sind aber genauso gut geeignet.

Kapitel 1



Einführung in C#

Herzlich willkommen in der Welt von C#! Dieses Kapitel nimmt Sie auf einen kurzen Rundgang durch die Sprache mit und beantwortet Fragen wie »Warum lohnt es sich, C# zu erlernen?«, »Was sind die wichtigsten Unterschiede zwischen C++ und C#?« und »In welchem Ausmaß wird C# die Entwicklung Ihrer Software vereinfachen und Ihnen mehr Spaß bereiten?«.

1.1 Warum eine neue Programmiersprache?

Sicher werden Sie sich diese Frage schon gestellt haben: »Warum soll ich eine andere Programmiersprache erlernen, wenn ich für meine Unternehmenslösungen mit C++ oder Visual Basic als Entwicklungssprache bestens zu Rande komme?« Jemand vom Marketing wird Ihnen darauf die prägnante Antwort geben: »Unsere Zielvorgabe ist es, dass C# die *lingua franca* für die NGWS-Anwendungsprogrammierung im Bereich der Unternehmenslösungen wird«. Der klare Unterton dieser Aussage ist, dass man bereits in naher Zukunft an dieser Sprache wohl eher schlecht vorbeikommen wird. Dieses Kapitel wird diese Aussage durch einige Sachargumente stützen sowie einen Überblick über die interessantesten Sprach-Features geben – gewissermaßen als Appetitmacher.

Die Programmiersprache C# stammt von C und C++ ab, versteht sich aber als moderne, einfache, typensichere und vor allem vollständig objektorientierte Programmiersprache. Falls Ihnen C/C++ vertraut ist, wird es ein Kinderspiel für Sie sein, C# zu erlernen. So sind viele C#-Anweisungen direkt C++ entlehnt, darunter das gesamte Ausdrucks- und Operatorwesen. Ja, wer nicht allzu genau hinsieht, wird ein C#- leicht für ein C++-Programm halten.

Das Attribut »moderne Programmiersprache« ist für C# keine Floskel. C# vereinfacht C++ in vielerlei Hinsicht, so beim Klassenbegriff, in den Namensbereichen und beim Überladen von Methoden. Was die Komplexität betrifft, so wurde C# gegenüber C++ um einiges abgespeckt, mit dem Ergebnis, dass die Sprache leichter und weniger fehleranfällig geworden ist. So gibt es keine Makros und keine Mehrfachvererbung mehr; diese Features haben im Allgemeinen mehr Ärger bereitet als genützt – speziell Entwicklern von Unternehmenslösungen.

Auf der anderen Seite sind aber auch ein ganze Reihe von Features hinzugekommen, die einem Programmierer das Leben leichter machen sollen: strikte Typensicherheit, Versionskontrolle, Garbage Collection und vieles mehr. All diese Konzepte zielen auf die Entwicklung komponentenorientierter Software. Trotzdem – oder gerade weil – man in C# nicht mehr das volle Ausdruckspotenzial von C++ zur Verfügung hat, verbessert sich die Produktivität.

Um nicht zu viele Merkmale der Sprache auf einmal vorzustellen, werden die folgenden Abschnitte einzeln auf die genannten Attribute eingehen und die Spracheigenschaften, die sie rechtfertigen, kurz vorstellen:

- einfach
- modern
- objektorientiert
- typensicher
- mit Versionskontrolle
- kompatibel
- flexibel

1.1.1 Einfach

Was einem im Zusammenhang mit C++ sicher nicht in den Sinn kommen wird, ist das Attribut »einfach«. Mit C# ist das anders: »Einfachheit« war in der Tat mit das wichtigste Ziel bei der Entwicklung dieser Programmiersprache. Ihrethalben sind eine ganze Reihe von Sprach-Features aus der C/C++-Welt auf der Strecke geblieben, aber es sind auch welche hinzugekommen.

Ein wirkliches prominentes Konzept, das in C# fehlt, sind Zeiger. Nachdem man es bei C# standardmäßig mit verwaltetem Code zu tun hat, sind potenziell unsichere Operationen wie direkte Speicherzugriffe oder Adressmanipulationen nicht nur »nicht erlaubt«, es fehlen schlicht die sprachlichen Mittel, sie zu formulieren. Segen oder Fluch? Ersteres natürlich, denn welcher C++-Programmierer kann schon von sich behaupten, noch nie auf Speicheradressen zugegriffen zu haben, für die der benutzte Zeiger nicht gedacht war.

In engem Zusammenhang mit dem für Entgleisungen anfälligen Zeigerkonzept stehen die Operatoren `::`, `.` und `->` für die Auflösung von Namensbereichen, Elementen und Referenzen. Diese Operatoren sind ein weiteres komplexes Kapitel von C++, für deren Verständnis gut und gerne ein zusätzlicher Tag harter Auseinandersetzung mit der Sprache zu veranschlagen ist. C# räumt mit dieser verwirrenden Vielfalt auf und packt die gesamte Funktionalität in einen einzigen Operator, den Punktoperator. Der Programmierer muss jetzt nur noch das Konzept der verschachtelten Bezeichner verstehen.

Auch muss man sich bei C# nicht mehr länger mit irgendwelchen kryptischen Typen herumschlagen, die auf verschiedene Prozessorarchitekturen gemünzt sind. C# benutzt ein vereinheitlichtes Typensystem, das es gestattet, wirklich *jeden* Wert als Objekt zu betrachten, gleich, ob er einen elementaren Typ trägt oder einer ausgewachsenen Klasse angehört. Im Gegensatz zu anderen Programmiersprachen ist die Verpackung elementarer Typen als Objekte bei C# nicht mit erheblichen Laufzeitverlusten behaftet, da die Sprache speziell für diesen Zweck den sogenannten Boxing-Mechanismus vorstellt. Auf Boxing und Unboxing geht

das Buch an späterer Stelle natürlich noch ausführlicher ein – jetzt nur soviel: Der Mechanismus erlaubt es, einfache Typen nach Belieben als Objekte zu verpacken und bei Bedarf wieder auszupacken.

Erfahreneren Programmierern wird es zwar nicht gefallen, die Vorteile sind aber nicht von der Hand zu weisen: C# behandelt Ganzzahlen (Integerwerte) und Wahrheitswerte (Boolesche Werte) als von Grund auf unterschiedliche Datentypen (zwischen denen es insbesondere keine implizite Typumwandlung gibt). Unbeabsichtigte Zuweisungen in `if`-Bedingungen oder in Schleifenkriterien, wie sie in C/C++ häufig Ursache verzwickter Fehler sind, sind in C# nicht möglich, weil der Compiler einen Fehler meldet, sobald Sie ein Kriterium formulieren, das keinen Booleschen Wert liefert.

Weiterhin räumt C# mit Redundanzen auf, die sich im Lauf der Jahre in C++ zur Aufrechterhaltung der C-Kompatibilität eingeschlichen haben – so beispielsweise mit `const` vs. `#define` und der Zweigleisigkeit bei Zeichen. In C# findet man ausschließlich die jeweils am weitesten verbreitete Alternative dieser Konzepte.

1.1.2 Modern

Der für das Erlernen von C# verbundene Aufwand ist eine wirklich zukunftsträchtige Investition, da C# eigens als *die* Programmiersprache für NGWS-Anwendungen entwickelt wurde. Sie werden daher vieles bereits als »zur Spache gehörig« vorfinden, was Sie in C++ noch mühsam haben implementieren müssen oder dort auch gar nicht verfügbar war.

Eine besonders für den Bereich der Unternehmenslösungen willkommene Erweiterung des Typensystems sind die finanzmathematischen Typen. Sie haben nun den Datentyp `decimal` zur Verfügung, der speziell für das Rechnen mit Geldbeträgen ausgelegt ist, weil er die im Zusammenhang mit Gleitkommawerten auftretenden Rundungsfehler vermeidet. Darüber hinaus können Sie natürlich jederzeit Ihre eigenen Datentypen implementieren, falls Sie für Ihre Problemstellung mit den elementaren Datentypen der Sprache nicht auskommen.

Nachdem Sie – sicherlich ein wenig skeptisch – erfahren mussten, dass C# keine Zeiger kennt, wird es Sie wahrscheinlich nicht mehr überraschen, dass Sie die Sprache insbesondere auch von der gesamten Speicherverwaltung entlastet. Das Speichermanagement übernimmt komplett der Garbage Collector des NGWS-Laufzeitsystems. Und weil sowohl der Code als auch der Speicher eines C#-Programms »verwaltet« ist, muss die Sprache schon aus Gründen der Stabilität absolute Typensicherheit garantieren.

Obwohl die Ausnahmebehandlung für einen C++-Programmierer sicher nichts Neues ist, wird ihm die zentrale Bedeutung dieses Konzepts unter C# auf den ersten Blick doch etwas ungewohnt erscheinen. Im Unterschied zu C++ ist der Ausnahmebehandlungsmechanismus von C# sprachübergreifend konzipiert und wird

gleichermaßen vom NGWS-Laufzeitsystem gestellt. Die spitzfindigen `HRESULT`-Werte von C++ haben ausgedient, die Fehlerbehandlung auf der Basis von Ausnahmen wird in C# zu einem robusten Mittel für den Programmieralltag.

Im Zeitalter der globalen Vernetzung ist Sicherheit zu einer zentralen Anforderung für Programmiersprachen geworden – insbesondere für solche, die für sich das Attribut »modern« in Anspruch nehmen wollen. C# lässt Sie hier nicht im Regen stehen: Die Sprache bietet von der Syntax her in Form der sogenannten Metadaten Unterstützung für das zentrale Sicherheitsmodell der NGWS-Laufzeitumgebung. Eine detailliertere Diskussion dieses Konzepts finden Sie gleich im nächsten Kapitel.

1.1.3 Objektorientiert

Ein Verzicht auf Objektorientierung dürfte so ziemlich das Letzte sein, was man heute von einer neuen Programmiersprache erwartet, oder? Selbstverständlich unterstützt C# sämtliche Schlüsselkonzepte der objektorientierten Programmierung, darunter Features wie Verkapselung, Vererbung und Polymorphie. Das gesamte in C# vorgegebene Klassenmodell gründet auf das von der NGWS-Laufzeitumgebung bereitgestellte virtuelle Objektsystem (VOS), das im nächsten Kapitel genauer beschrieben wird. Mit anderen Worten, das Objektmodell ist nun Bestandteil der Infrastruktur und nicht mehr nur der Programmiersprache.

Wie Sie gleich zu Anfang bemerken werden, wenn Sie sich mit der Sprache als solcher auseinandersetzen, gibt es keine globalen Funktionen, Variablen und Konstanten mehr. Alles muss in einer Klasse enthalten sein – entweder als statisches Element (zu einem Typ gehörig) oder als dynamisches Element (zur Instanz eines Typs gehörig). Das verbessert nicht nur die Lesbarkeit Ihres Codes, sondern beugt auch potenziellen Namenskonflikten vor.

Die Methoden einer Klasse sind standardmäßig nicht virtuell, das heißt, sie lassen sich in einer abgeleiteten Klasse nicht überschreiben. Der Hintergedanke dabei ist, das unbeabsichtigte Überschreiben von Methoden zu verhindern. Damit eine Methode überschrieben werden kann, muss sie explizit mit dem Modifizierer `virtual` deklariert sein. Das hält nicht nur den Umfang der virtuellen Methodentabelle (*vtable*) klein, sondern garantiert auch, dass sich Ihr Code versionsgerecht verhalten kann.

Wer von C++ kommt, ist es gewohnt, die Sichtbarkeit von Elementen über Zugriffsmodifizierer zu gestalten. Diese Modifizierer, `private`, `protected` und `public`, finden Sie natürlich auch in C#. Darüber hinaus kennt C# aber noch einen vierten Modifizierer: `internal`. Details dazu diskutiert Kapitel 5, »Klassen«.

Haben Sie in C++ jemals bewusst eine Klasse von mehreren Basisklassen abgeleitet? (ATL-Programmierer bitte weghören, Ihre Stimme zählt hier nicht!) In fast allen Fällen kommt man mit einer einzige Basisklasse aus. Und wenn man doch

mehrere Basisklassen verwendet, sind die Scherereien im Allgemeinen größer als der Nutzen. Nicht zuletzt aus diesem Grund ist C# auf eine einzige Basisklasse beschränkt. Und wenn Sie glauben, auf Mehrfachvererbung nicht verzichten zu können, bleibt Ihnen immer noch die Möglichkeit, Ihr Polymorphiekonzept über Schnittstellen zu implementieren.

Nachdem es in C# keine Zeiger gibt, drängt sich die Frage auf, wie C# mit Funktionszeigern verfährt. Die Anwort lautet: vermittels *Delegationen*, die gleichzeitig den Unterbau für das gesamte Ereignismodell der NGWS-Laufzeitumgebung bilden. Mehr darüber gleichfalls in Kapitel 5, »Klassen«.

1.1.4 Typensicher

Noch einmal zurück zu den Zeigern: Wenn Sie in C++ einen Zeiger haben, steht es Ihnen frei, für diesen eine Typumwandlung in jeden anderen Datentyp zu erzwingen. Das heißt, es steht Ihnen vom Prinzip her auch frei, wirklich idiotische Dinge zu tun, wie einen Wert vom Typ `int*` (`int`-Zeiger) in einen Wert vom Typ `double*` (`double`-Zeiger) zu verwandeln. Solange hinter einer solchen Operation noch ein realer Speicherwert steckt, mag das zwar funktionieren, sinnvoll ist es aber sicher nicht. Von einer eigens für Unternehmenslösungen entwickelten Sprache sollte man schon etwas mehr Sicherheit erwarten können.

Nicht zuletzt aus diesem Grund, aber auch um den Garbage Collector zu schützen, steht die Typensicherheit im Anforderungskatalog von C# ganz weit oben. Das bedeutet natürlich für Sie, dass Sie sich bei C# an ein paar Regeln halten müssen, die den Umgang mit Variablen betreffen:

- Variablen müssen grundsätzlich initialisiert werden. Die Elementvariablen von Objekten setzt zwar der Compiler implizit auf 0, um lokale Variablen von Methoden müssen Sie sich aber in jedem Fall selbst kümmern. Da der C#-Compiler nichtinitialisierte Variablen bemängelt, werden schwer aufzufindende Fehler, wie sie in C/C++ häufig auftreten, bereits im Keim erstickt.
- C# räumt mit unsicheren Typumwandlungen ein für alle Mal auf. Der C#-Compiler schließt die Typumwandlung von einem wertbehafteten Typ in einen Referenztyp aus (das Boxing ausgenommen) und überprüft bei der Typumwandlung von Objektreferenzen rigide, ob das jeweilige Objekt wirklich von der Klasse abstammt, die für die als Linkswert fungierende Objektvariable vereinbart wurde.
- C# stellt die Einhaltung von Array-Grenzen sicher. Im Gegensatz zu C/C++ ist es nicht mehr möglich, »Elemente« jenseits der Arraygrenzen anzusprechen und in Speicherbereiche zu schreiben oder daraus zu lesen, für die keine Belegung erfolgt ist.
- Arithmetische Operationen können Überläufe produzieren. C# bietet eine Überlaufkontrolle auf Anwendungsebene oder auf der Ebene einzelner Anwei-

sungen. Bei eingeschalteter Überlaufkontrolle werden Überlauffehler als Ausnahmen signalisiert, ansonsten jedoch ignoriert.

- C# stellt sicher, dass Referenzparameter (call-by-reference) typensicher sind.

1.1.5 Mit Versionskontrolle

Das Problem, dass bei älteren Programmen nach dem Versions-Update einer gemeinsam genutzten DLL oft gar nichts mehr läuft, ist nicht neu. Sicher werden auch Sie ein Lied davon singen können. Es kommt daher, dass jedes Programm eine neue Version einer DLL installieren kann, mit der dann alle anderen Programme leben müssen, ob sie können oder nicht. Versionsabhängigkeit ist ein echtes Problem, das nach einer systemseitigen Lösung verlangt.

Wie in Kapitel 9, »Konfiguration und Verbreitung« näher ausgeführt, ist die NGWS-Laufzeitumgebung auf eine Versionskontrolle für gemeinsam genutzte Komponenten eingerichtet, und C# wartet natürlich mit der nötigen Unterstützung für diesen Mechanismus auf. Obwohl C# selbst keine Garantie bieten kann, dass eine Anwendung die richtigen Versionen gemeinsam genutzter Komponenten zu sehen bekommt, so schafft es doch zumindest die Voraussetzungen, dass eine Versionskontrolle überhaupt stattfinden kann. Die Versionskontrolle bietet einem Entwickler eine weitgehende Sicherheit, dass die Pflege seines Codes die binäre Kompatibilität mit bestehenden älteren Anwendungen nicht gefährdet.

1.1.6 Kompatibel

C# ist keine geschlossene Welt. Die Sprache erlaubt den Zugriff auf verschiedene APIs, allem voran auf solche, die der NGWS Common Language Specification (CLS) genügen. Die CLS definiert einen Standard für die sprachübergreifende Verwendung von Code. Um die CLS-Verträglichkeit eines Datentyps (Klasse) durchzusetzen, überprüft der C#-Compiler alle als öffentlich exportierten Größen und erzeugt eine Fehlermeldung, wenn die CLS-Verträglichkeit nicht gegeben ist.

Natürlich werden Sie auch mit älteren COM-Objekten arbeiten wollen. Das ist ohne weiteres möglich, ja, der Zugriff auf COM-Objekte im Rahmen der NGWS-Laufzeitumgebung ist sogar vollständig transparent. Auf die Integration älteren Codes geht Kapitel 10, »Integration von nicht verwaltetem Code«, näher ein.

Bei der OLE-Automatisierung liegt der Fall anders. Wer jemals ein OLE-Automatisierungs-Projekt in C++ realisiert hat, wird von den Automatisierungsdatentypen seine Finger nicht mehr lassen wollen. Erfreulicherweise bietet C# dafür eine brauchbare Unterstützung – und zwar ohne Sie großartig mit den Details zu belästigen.

Schließlich bietet Ihnen C# auch noch die Möglichkeit, im C-Stil vorliegende API-Funktionen anzusprechen. In der Tat können Sie von C# aus jeden Eintritts-

punkt in eine DLL nutzen, solange er der C-Aufrufkonvention folgt. Der Mechanismus, der den Zugriff auf solchen Code ermöglicht, trägt den Namen Platform Invocation Services (PInvoke) und wird in Kapitel 10 anhand einiger Beispiele, die den Aufruf von C-API-Funktionen demonstrieren, näher vorgestellt.

1.1.7 Flexibel

Der letzte Absatz des vorigen Abschnitts wird bei einem C-Programmierer sicher die Frage aufgeworfen haben, was eigentlich mit API-Funktionen ist, deren Aufruf die Angabe von Zeigern erforderlich macht. Das ist in der Tat ein Problem. Und es beschränkt sich leider nicht nur auf einige wenige API-Funktionen, sondern gilt dummerweise für den Löwenanteil. Mit anderen Worten: Der Zugriff auf Win32-Code ist ohne das unsichere klassische Zeigerkonzept schlicht nicht zu bewerkstelligen (obwohl einige API-Funktionen auch über die COM- und PInvoke-Unterstützung zugänglich sind).

Die Lösung ist nicht gerade schön, aber sie funktioniert: Obwohl C#-Code standardmäßig im sogenannten *sicheren Modus* übersetzt wird, kennt der C#-Compiler auch einen unsicheren Modus, in dem die Deklaration von Zeigern, Strukturen und statischen Arrays im klassischen Stil von C erlaubt ist. Sowohl der sichere Code als auch der unsichere Code kommen unterschiedslos im verwalteten Bereich zur Ausführung, ohne dass zwischen beiden ein Marshaling stattfinden würde.

Was aber sind die Implikationen für den Umgang mit Speicher, den Sie im unsicheren Modus anfordern? Nun, der Garbage Collector wird natürlich die Finger von diesem Speicher lassen. »Unsichere« Variablen aber steckt der Garbage Collector in denselben Speicherpool wie die »sicheren«.

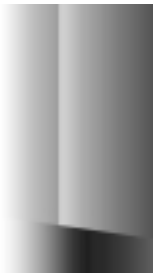
1.2 Zusammenfassung

Die Sprache C# stammt von C und C++ ab und ist speziell für Entwickler von Unternehmenslösungen gedacht, die gewillt sind, etwas von der gewaltigen Ausdruckskraft der Sprache C++ gegen erheblich mehr Sicherheit, Bequemlichkeit und verbesserte Produktivität einzutauschen. C# ist eine moderne, objektorientierte und typensichere Sprache, die sich zwar eine Menge von C und C++ ausborgt, jedoch für spezifische Konzepte, wie Namensräume, Klassen, Methoden und Ausnahmebehandlung, durchaus eigene Wege geht.

C# kann mit einer Reihe an bequemen Features aufwarten, darunter Garbage Collection, Typensicherheit, Versionskontrolle und vieles mehr. Gewissermaßen im Gegenzug bleibt dabei das Zeigerkonzept auf der Strecke, sofern Sie Ihren Code entsprechend der Standardvorgabe für den sicheren Modus übersetzen. Insbesondere für die Typensicherheit macht sich das natürlich bezahlt. Falls Sie aber dennoch Zeiger benötigen, haben Sie keine andere Wahl, als auf den unsicheren

Modus auszuweichen. Beachten Sie jedoch, dass dann kein Marshaling zwischen sicherem und unsicherem Code stattfindet.

Kapitel 2



Der Unterbau – das NGWS-Laufzeitsystem

Nachdem Sie nun bereits einen ersten Eindruck von C# bekommen haben, stellt Ihnen dieses Kapitel das NGWS-Laufzeitsystem vor. C# ist für die Ausführung von Programmen auf dieses Laufzeitsystem angewiesen, weshalb es sich empfiehlt, die Arbeitsweise und Konzeption dieses Systems besser kennen zu lernen.

Dieses Kapitel organisiert sich in zwei große Abschnitte – einen Grundlagenteil sowie einen Teil, der die Implikationen für den alltäglichen Umgang vorstellt. Die Teile überschneiden sich inhaltlich ein wenig, was ein Verständnis der Konzepte sicher erleichtert.

2.1 Das NGWS-Laufzeitsystem

Das Subsystem der Next Generation Windows Services, kurz NGWS, umfasst ein vollständiges Laufzeitsystem für die Ausführung von Code und bietet darüber hinaus eine Reihe von Diensten, um die Programmierung zu vereinfachen. Sofern der verwendete Compiler auf die Benutzung dieses Laufzeitsystems eingerichtet ist, genießen Sie für Ihren Code alle Vorzüge dieser systemseitig verwalteten Laufzeitumgebung.

Wie wohl auch nicht anders zu erwarten bietet der C#-Compiler Unterstützung für das NGWS-Laufzeitsystem. Und er ist dabei nicht der einzige; Visual Basic und C++ sind mit von der Partie. Der Code, den diese Compiler für das NGWS-Laufzeitsystem generieren, wird als *verwalteter Code* bezeichnet. Die Vorteile, die Ihr Code vom NGWS-Laufzeitsystem erhält, sind im Einzelnen:

- Integration über die Sprachgrenzen hinweg (über eine gemeinsame Sprachenspezifikation)
- automatische Speicherverwaltung (Garbage Collector)
- behandlung von Ausnahmezuständen über Sprachgrenzen hinweg
- erhöhte Sicherheit inklusive Typprüfungen
- Unterstützung von Versionsnummern – das Ende des »DLL-Friedhofs«
- vereinfachtes Modell für die Interaktion von Komponenten

Damit das NGWS-Laufzeitsystem diese Unterstützung leisten kann, muss der Compiler zusammen mit dem verwalteten Code Metadaten erzeugen, die unter anderem die verwendeten Typen beschreiben und zusammen mit dem Code in der ausführbaren Datei gespeichert werden. Die Dateien sind wie gewohnt im PE-Format (»portable executable«) gehalten.

Einer der Schwerpunkte des NGWS-Laufzeitsystems liegt offensichtlich in der Integration mehrerer Programmiersprachen. Tatsächlich geht diese Integration so weit, dass Sie C#-Klassen von Visual-Basic-Objekten ableiten können. (Natürlich gibt es dafür einige Voraussetzungen, die weiter hinten in diesem Kapitel erläutert werden.)

Ein Feature, das vor allem das Herz von C/C++-ProgrammiererInnen erfreuen wird, ist die Speicherverwaltung. Die allseits berüchtigten Speicherlecks sind in C# kein Thema, weil sich das Laufzeitsystem um alle Objekte kümmert: Sein Garbage Collector gibt sie automatisch wieder frei, wenn sie nicht länger benötigt werden (d.h. keine Referenzen mehr auf sie vorhanden sind). Wer jemals das Vergnügen hatte, sich mit COM-Objekten in C++ herumzuschlagen, wird das besonders zu schätzen wissen.

Bei der Installation verwalteter Komponenten und Anwendungen auf anderen Systemen hilft das NGWS-Laufzeitsystem ebenfalls kräftig: Anhand der in den Dateien gespeicherten Metadaten lässt sich zuverlässig ermitteln, ob Bibliotheken und andere Dateien in exakt der Version vorliegen, die Ihre Anwendung bzw. Komponente erwartet. In der Praxis bedeutet das, dass es wesentlich seltener Probleme mit unpassenden Versionen und nur scheinbar erfüllten Abhängigkeiten geben wird. Ein weiterer, ebenfalls nicht zu unterschätzender Vorteil der Speicherung von Metadaten in der ausführbaren Datei: Typinformationen sind am selben Ort wie der Code – die Registrierung des Systems ist damit als potenzielle Problemquelle weitgehend ausgeschaltet.

Die beiden nächsten Abschnitte sind den Features des NGWS-Laufzeitsystems gewidmet, die in erster Linie vor der Ausführung Ihrer C#-Anwendungen zum Tragen kommen:

- Intermediate Language Code (IL-Code) und Metadaten
- JITter

2.1.1 Intermediate Language Code (IL-Code) und Metadaten

C#-Compiler erzeugen keine direkt ausführbaren Maschinenbefehle, sondern einen Zwischencode (IL für Intermediate Language), der seinerseits als Input für einen verwalteten Ausführungsprozess des NGWS-Laufzeitsystems dient. Einer der großen Vorteile dieses Zwischencodes liegt darin, dass er unabhängig vom Prozessortyp ist – was aber natürlich auch bedeutet, dass auf dem jeweiligen Zielsystem ein Compiler existieren muss, der diesen Code in Maschinenbefehle umsetzt.

Wie erwähnt, beschränkt sich der C#-Compiler nicht auf das Erzeugen von IL-Code, sondern generiert Metadaten dazu, die dem Laufzeitsystem eine Menge zusätzlicher Informationen liefern, wie etwa die Definition eines Datentyps und die Signatur seiner Elementfunktionen. Grob gesagt, enthalten Metadaten alles,

was Typbibliotheken, Registrierungseinträge und andere Informationen für COM sind – nur finden sich diese Daten hier eben in derselben Datei wie der IL-Code (und nicht auf ein halbes Dutzend anderer Speicherorte verteilt).

Das zur Speicherung von IL-Code und Metadaten verwendete Format basiert auf dem PE-Format, das Win32 seit jeher für EXE- und DLL-Dateien verwendet. Das NGWS-Laufzeitsystem ist dafür zuständig, beim Laden einer solchen Datei den IL-Code und die Metadaten auszulesen.

Bevor es nun weiter geht, sollten Sie zumindest eine ungefähre Vorstellung davon bekommen, woraus IL-Code besteht. Die folgende Liste der Befehlskategorien ist weder vollständig noch zum Auswendiglernen gedacht, sollte aber einen guten Überblick darüber geben, auf welche Arten von Anweisungen sich C#-Programme stützen:

- arithmetische und logische Operationen
- Flusskontrolle
- direkte Speicherzugriffe
- Stackmanipulation
- Argumente und lokale Variablen
- Speicherbelegung auf dem Stack
- Objektmodell
- Werte instanziierbarer Typen (Objekte)
- kritische Abschnitte
- Arrays
- typisierte Speicherorte

2.1.2 JIT-Compiler

Verwalteter Code wird von C#-Compilern – und anderen Compilern, die verwalteten Code erzeugen können – als Zwischencode (IL-Code) gespeichert. Obwohl Windows das Ergebnis der Kompilierung als gültige PE-Datei erkennt, ist ein solches Programm erst nach einer weiteren Umsetzung in Maschinenbefehle ausführbar. Für diese Umsetzung sind die *Just-in-Time*-Compiler der NGWS-Laufzeitumgebung zuständig, die oft auch kurz als *JITter* bezeichnet werden.

Wieso Just-in-Time – also stückweise bei Bedarf? Warum liest das NGWS-Laufzeitsystem nicht einfach den gesamten IL-Code einer PE-Datei auf einmal und setzt ihn in Maschinenbefehle um? Hauptsächlich, weil das bei umfangreicheren Anwendungen recht lange dauern könnte – und es eben wesentlich effizienter ist, nur die Teile einer Anwendung umzusetzen, die der Benutzer auch wirklich

braucht. Anders gesagt: Wer mit einer Textverarbeitung schnell drei Telefonnummern festhalten will, möchte wohl nur ungern darauf warten, bis das System mit dem Umsetzen der Grammatikprüfung, des eingebauten Grafikeditors und der Serienbrieffunktion zu Rande gekommen ist.

Der technische Hintergrund sieht im Wesentlichen so aus: Wenn ein Typ geladen wird, erzeugt die dafür zuständige Routine des NGWS-Laufzeitsystems für jede Methode dieses Typs einen kurzen Vorspann (mit Maschinenbefehlen) und setzt ihn ein. Wenn die Methode zum ersten Mal aufgerufen wird, sorgt dieser Vorspann für den Aufruf des JIT-Compilers, der seinerseits den IL-Code der Methode in Maschinenbefehle umsetzt und den Vorspann so ändert, dass er zukünftig auf die neu erzeugten Maschinenbefehle zeigt. Bei weiteren Aufrufen der Methode werden die Maschinenbefehle also ohne Umweg ausgeführt (was auch bedeutet, dass der JIT-Compiler im Verlauf eines Programms irgendwann weitgehend arbeitslos wird).

Wie der Terminus »JITter« bereits vermuten lässt, gibt es nicht einen einzigen JIT-Compiler, sondern mehrere davon. Für Windows wird das NGWS-Laufzeitsystem mit drei verschiedenen JIT-Compilern ausgeliefert:

- *JIT* ist der Standard-JIT-Compiler des NGWS-Laufzeitsystems: Ein optimierender Codegenerator mit integrierter Datenflussanalyse, der dicht gepackten, hoch optimierten, verwalteten Maschinencode erzeugt. JIT kommt mit dem gesamten IL-Befehlssatz zurecht, hat aber einen entsprechend hohen Ressourcenbedarf – speziell, was den Speicherplatzverbrauch durch den Compiler selbst, das sich ergebende Working Set der Anwendung und die für die Optimierungen notwendige Zeit betrifft.
- *EconoJIT* ist im Gegensatz zum JIT auf die schnellstmögliche Umsetzung von IL-Code in Maschinenbefehle ausgelegt und hält das Ergebnis bei Bedarf in einem (großen) Cache vor. Der erzeugte Maschinencode ist zwangsläufig nicht so hoch optimiert wie beim JIT, dafür geschieht die Umsetzung aber so schnell, dass es sich das Laufzeitsystem bei Speicherengpässen leisten kann, momentan unbenutzten Maschinencode wieder zu verwerfen. Auf diese Weise werden auch bei Systemen mit magerer Ausstattung des Hauptspeichers Anwendungen fast beliebiger Größe möglich.
- *PreJIT* basiert zwar auf JIT, lehnt sich aber wesentlich stärker an traditionelle Vorbilder an. Dieser Compiler kommt ausschließlich bei der Installation von NGWS-Komponenten zum Zuge, übersetzt den gesamten IL-Code der Komponente in verwalteten Maschinencode, und wird im Weiteren nicht mehr benötigt. Die Hauptvorteile liegen natürlich in geringeren Lade- und Startzeiten von Anwendungen.

Zwei dieser drei JITter sind zur Laufzeit mit von der Partie. Wie lässt sich festlegen, welcher JIT wann eingesetzt wird, und welche Strategie er bei der Speicher-

verwaltung benutzt? Dafür ist der JIT Compiler Manager zuständig – ein kleines Utility mit dem Dateinamen `jitman.exe`, das bei der Installation des NGWS-Subsystems im Ordner `bin` des SDKs untergebracht wird. Der Start dieses Programms fügt der Task-Leiste ein Symbol hinzu; ein Doppelklick auf dieses Symbol zeigt das Dialogfeld dieses Managers an (vgl. Abbildung 2.1).

Bild 2.1: Der JIT Compiler Manager zur Auswahl und Konfiguration des JIT-Compilers

So klein dieses Dialogfeld ist, so weit reichend sind die Folgen seiner Einstellungen. Die folgende Liste gibt einen Überblick:

- *Use EconoJIT only* – wenn Sie dieses Kästchen leer lassen, verwendet die NGWS-Laufzeitumgebung den Standardcompiler.
- *Max Code Pitch Overhead (%)* – gilt ausschließlich für EconoJIT und legt fest, wieviel Prozent der Laufzeit der Compiler maximal beanspruchen darf. Beim Überschreiten dieses Schwellwerts wird der Codecache vergrößert (was letztendlich darauf hinausläuft, dass das Laufzeitsystem seltener bereits übersetzten Code wieder verwirft und später den Compiler erneut bemühen muss).
- *Limit Size of Code Cache* – ist standardmäßig nicht gesetzt: Das NGWS-Laufzeitsystem belegt so viel Hauptspeicher für den Codecache, wie es kriegen kann. Wenn Sie ein Häkchen in dieses Kästchen setzen, können Sie dem Laufzeitsystem über *Max Size of Cache (bytes)* eine Obergrenze für den Codecache vorschreiben.
- *Max Size of Cache (bytes)* – legt eine Maximalgröße für den Codecache fest. Wer will, kann hier recht knauserig sein, sollte aber eine Grenze beachten: Der Codecache muss in jedem Fall so viel Platz bieten, wie die umfangreichste Methode einer Anwendung braucht – ansonsten schlägt die JIT-Kompilierung dieser Methode fehl.
- *Optimize For Size* – weist den JIT-Compiler an, möglichst Platz sparenden anstelle von möglichst schnellem Maschinencode zu erzeugen. Dieses Kästchen trägt standardmäßig kein Häkchen.
- *Enable Concurrent GC* – ist standardmäßig nicht gesetzt: Der Garbage Collector läuft in dieser Einstellung im selben Thread wie die jeweilige Anwendung – und kann diese kurzzeitig blockieren, wenn umfangreichere Aufräumarbei-

ten zu erledigen sind. Wenn Sie ein Häkchen in dieses Kästchen setzen, spendiert das NGWS-Laufzeitsystem dem Garbage Collector einen eigenen Thread. Beachten Sie jedoch, dass der Garbage Collector in einem separaten Thread etwas langsamer arbeitet und dieses Feature zum Zeitpunkt der Drucklegung dieses Buchs ausschließlich unter Windows 2000 verfügbar ist.

Da die Beispiele dieses Buchs allesamt recht klein gehalten sind, werden Sie beim Experimentieren mit den Einstellungen des JIT Compiler Managers kaum große Unterschiede finden. Die Einstellung des Garbage Collectors macht sich vor allem bei solchen Anwendungen deutlich bemerkbar, die intensiv von der Benutzeroberfläche Gebrauch machen.

2.2 Das virtuelle Objektsystem (VOS)

Die vorangehenden Abschnitte haben die Arbeitsweise des NGWS-Laufzeitsystems beschrieben, ohne dabei auf den technischen Hintergrund näher einzugehen oder zu erläutern, welche Philosophie dahinter steckt. Die nächsten Seiten holen diese Erklärungen nach: Sie beschäftigen sich ausschließlich mit dem virtuellen Objektsystem (VOS).

Das virtuelle Objektsystem legt unter anderem die Regeln fest, denen das NGWS-Laufzeitsystem bei der Deklaration, dem Einsatz und der Verwaltung von Typen folgt. Das VOS stellt damit einen Teil eines Rahmenwerks dar, das sowohl die Integration mehrerer Programmiersprachen als auch die Typensicherheit bei sprachübergreifender Programmierung gewährleistet, ohne dass die Performance darunter zu leiden hätte.

Das Rahmenwerk als solches ist die Grundlage der gesamten Architektur des NGWS-Subsystems. Da das Wissen um diese Architektur beim Entwickeln von C#-Anwendungen und C#-Komponenten wesentlich ist, gehen die folgenden Abschnitte auf die vier Teilbereiche dieses Rahmenwerks im Detail ein:

- Das VOS-Typensystem – stellt ein reichhaltiges Typensystem zur Unterstützung einer breiten Auswahl von Programmiersprachen zur Verfügung.
- Metadaten – beschreiben und referenzieren die vom VOS-Typensystem definierten Typen. Da das beständige Format von Metadaten unabhängig von der Programmiersprache ist, lassen sich Metadaten sowohl als Medium zum Datenaustausch zwischen Werkzeugen als auch für das VES der NGWS (siehe unten) verwenden.
- Die Common Language Specification (CLS) – definiert eine Untermenge der vom VOS verwendeten Typen zusammen mit Konventionen für ihren Einsatz. Wenn eine Klassenbibliothek sich auf diese Typen beschränkt und die Konventionen einhält, lässt sie sich zusammen mit allen Programmiersprachen verwenden, die ihrerseits der CLS folgen.

- Das Virtual Execution System (VES) – stellt die Implementation der VOS dar und ist für das Laden sowie Ausführen von Programmen zuständig, die für das NGWS-Laufzeitsystem geschrieben worden sind.

2.2.1 Das VOS-Typensystem

Das VOS-Typensystem soll die vollständige Implementation eines breiten Spektrums an Programmiersprachen unterstützen und muss deshalb nicht nur mit objektorientierten, sondern auch mit prozeduralen Programmiersprachen zurechtkommen.

Aktuelle Programmiersprachen bieten eine große Zahl weitgehend ähnlicher, aber dennoch inkompatibler Datentypen. Ein Beispiel dafür sind schlichte Ganzzahlwerte: VB speichert den Typ `Integer` mit 16 Bit, C++ den Typ `int` dagegen mit 32 Bit. Leider lassen sich solche Beispiele angefangen von Typen für Kalenderdatum und Uhrzeit bis hin zu den Typen für Datenbanken fast beliebig fortsetzen. Inkompatibilitäten dieser Art machen das Erstellen verteilter Anwendungen unnötig kompliziert – und das erst recht, wenn mehrere Programmiersprachen im Spiel sind.

Ein weiteres Problem, das sich aus kleinen Unterschieden der einzelnen Programmiersprachen ergibt: Ein mit einer Sprache definierter Typ lässt sich nicht in einer anderen Sprache verwenden. (COM löst dieses Problem mit einem binären Standard für Schnittstellen, aber auch nur zum Teil). Tatsächlich ist die Wiederverwendung von Code auch heute noch engen Grenzen unterworfen.

Die größte Hürde für verteilte Anwendungen sind allerdings die Objektmodelle der einzelnen Programmiersprachen, die sich fast in jedem Punkt unterscheiden – egal, ob es nun um Ereignisse, Eigenschaften oder die beständige Speicherung wertbehafteter Elemente geht.

Das VOS-Typensystem stellt einen neuen Ansatz dar, die verschiedenen Programmiersprachen unter einen Hut zu bringen: Es definiert Typen zur Beschreibung von Werten und legt sozusagen einen Vertrag fest, an den sich alle Werte eines bestimmten Typs halten müssen. Da das VOS-Typensystem sowohl objektorientierte als auch prozedurale Programmiersprachen unterstützt, gibt es hier zwei Kategorien: Werte und Objekte.

Bei Werten beschreibt der zugeordnete Typ die Speicherrepräsentation sowie die Operationen, die sich über dieser Repräsentation ausführen lassen. Objekte haben hier mehr Möglichkeiten, weil der Typ explizit in seiner Repräsentation gespeichert wird. Jedes Objekt hat eine Identität, die es von allen anderen Objekten unterscheidet. Details zu den von C# unterstützten VOS-Typen finden Sie in Kapitel 4, »Die Typen von C#«.

2.2.2 Metadaten

Obwohl Metadaten primär zur Beschreibung und Referenzierung der durch das VOS-Typensystem definierten Typen verwendet werden, sind sie nicht auf diesen Zweck beschränkt. Wenn Sie ein Programm schreiben, werden die von Ihnen deklarierten Typen – unabhängig davon, ob es dabei um wertbehaftete Typen oder um Referenzen geht – dem NGWS-Laufzeitsystem über Typdeklarationen bekannt gemacht, die sich ihrerseits zusammen mit dem Zwischencode in der ausführbaren Datei befinden.

Metadaten liefern die Informationen, die das NGWS-Laufzeitsystem für verschiedene Aufgaben benötigt: zum Auffinden und Laden von Klassen, zum Anlegen von Instanzen (Objekten) dieser Klassen im Hauptspeicher, zum Auflösen von Methodenaufrufen, zum Umsetzen von IL-Code in Maschinenbefehle, zur Sicherheitsprüfung und schließlich zum Errichten der kontextbezogenen Grenzen zur Laufzeit eines Programms (wie etwa Sichtbarkeit, Verfügbarkeit und Lebensdauer von Objekten).

Wie Metadaten zu Stande kommen, ist für Sie als ProgrammiererIn im Wesentlichen uninteressant, weil das der C#-Compiler beim Erzeugen des IL-Codes (also nicht etwa der JIT-Compiler) übernimmt und sie zusammen mit dem IL-Code in der ausführbaren Datei speichert. Die Art der Speicherung ist selbstverständlich standardisiert – im Gegensatz zu C++-Compilern, bei denen jede Implementation ihren eigenen Regeln folgt, wenn es bei exportierten Funktionen um die Kombination von Signaturen mit Bezeichnern geht.

Der wesentliche Vorteil der Kombination von Metadaten mit Code liegt darin, dass die Informationen über einen Typ zusammen mit dem Typ selbst gespeichert sind und sich nicht, wie bisher gewohnt, über mehrere Speicherorte (Typbibliotheken und die Registrierung) im System verteilen – wovon jeder COM-Programmierer ein Lied zu singen weiß. Außerdem können Sie im NGWS-Laufzeitsystem unterschiedliche Versionen einer Bibliothek in ein und demselben Kontext verwenden, weil diese Bibliotheken eben nicht zentral über die Registrierung referenziert werden, sondern über die Metadaten in den jeweiligen Dateien.

2.2.3 Die Common Language Specification (CLS)

Diese Spezifikation ist eigentlich kein eigenständiger Teil des virtuellen Objektsystems, sondern vielmehr eine Spezialisierung. Sie definiert eine Untermenge der VOS-Datentypen zusammen mit einer Reihe von Konventionen.

Wozu das gut sein soll? Nun: Eine Klassenbibliothek, die sich auf diese Untermenge beschränkt und an sämtliche von der CLS aufgestellten Regeln hält, lässt sich von allen Clients verwenden, die ihrerseits der CLS folgen – unabhängig davon, mit welcher Sprache die einzelnen Klassen definiert wurden. Dabei geht

es erfreulicherweise nicht um den kleinsten gemeinsamen Nenner: Die CLS schränkt ausschließlich die nach außen hin sichtbaren Elemente von Klassen ein (wie Methoden, Eigenschaften und Ereignisse), nicht aber deren Implementation.

Das Ergebnis: Niemand hält Sie davon ab, eine Komponente in C# zu schreiben, von dieser Klasse mit VB eine weitere Klasse abzuleiten – und weil Ihnen die mit VB hinzugefügte Funktionalität so gut gefällt, können Sie diese Klasse dann erneut als Basis einer weiteren Ableitung in C# verwenden. Das funktioniert tatsächlich – solange sich die nach außen hin sichtbaren Elemente dieser Klasse an die von der CLS aufgestellten Regeln halten.

Die in diesem Buch vorgestellten Beispiele schenken diesen Regeln wenig Beachtung - nicht zuletzt, weil es dabei in den allermeisten Fällen um eigenständige Programme und nicht um Komponenten geht. Wenn Sie eine Klassenbibliothek aufbauen wollen, sieht die Sache etwas anders aus. Tabelle 2.1 gibt einen Überblick über die Typen und Konventionen der CLS (die, wie gesagt, ausschließlich für nach außen hin sichtbare Elemente von Klassen gelten).

Vollständig ist diese Liste nicht – sie beschränkt sich auf die wesentlichen Punkte. Da in den weiteren Kapiteln auch nicht bei jedem einzelnen Typ extra auf die CLS-Verträglichkeit eingegangen wird, sollten Sie sie aber wenigstens überfliegen (und sich bitte nicht wundern, weil einige der verwendeten Termini erst in den hinteren Kapiteln dieses Buchs erklärt werden).

Einfache Typen
`bool`
`char`
`byte`
`short`
`int`
`long`
`float`
`double`
`string`
`object` (die Basisklasse aller Objekte)
Arrays
Die Dimension muss bekannt sein (>=1) und die Zählung der Elemente grundsätzlich mit 0 beginnen. Die Elemente müssen einen CLS-Typ haben.

Tab. 2.1: Typen und Konventionen der Common Language Specification

Typen

- können abstrakt oder versiegelt sein.
- können eine beliebige Zahl von Schnittstellen implementieren. Methoden mit gleichen Namen und gleicher Signatur als Teil verschiedener Schnittstellen sind möglich.
- Klassen müssen von (exakt) einer anderen Klasse abgeleitet sein. Das Verbergen und Überschreiben von Elementen ist erlaubt.
- können eine beliebige Zahl von Elementen (wertbehaftet, Methoden, Ereignisse, Objekte) haben.
- können eine beliebige Zahl von Konstruktoren definieren.
- können wahlweise öffentlich oder nur innerhalb der jeweiligen NGWS-Komponente verfügbar sein. Nur die öffentlich verfügbaren Elemente werden als Teil der Schnittstelle gewertet und müssen sich an die CLS halten.
- Alle wertbehafteten Typen müssen von `System.ValueType` abgeleitet sein. Die Ausnahme von dieser Regel sind Aufzählungen, bei denen als Basisklasse `System.Enum` vorgeschrieben ist.

Elemente

können Elemente einer Basisklasse wahlweise überschreiben und verbergen.

Argument- und Ergebnistypen von Methoden sind auf die von der CLS definierten Typen beschränkt.

Konstruktoren, Methoden und Eigenschaften können überschrieben werden.

Typen können abstrakte Elemente enthalten, dürfen dann aber nicht versiegelt sein.

Methoden

Können wahlweise statisch, virtuell oder auf die jeweilige Instanz bezogen sein.

Virtuelle und auf die jeweilige Instanz einer Klasse bezogene Methoden können wahlweise abstrakt sein oder eine Implementation haben. Für statische Methoden ist dagegen eine Implementation gefordert.

Virtuelle Methoden können (müssen aber nicht) endgültig sein.

Wertbehaftete Elemente

können statisch oder auf die jeweilige Instanz bezogen sein.

Statische Elemente können entweder Literale sein oder müssen im Konstruktor gesetzt werden.

Eigenschaften

lassen sich wahlweise über `get`- und `set`-Methoden oder über die für Eigenschaften vorgesehene Syntax zur Verfügung stellen.

Der Ergebnistyp der `get`-Methode und der erste Parameter der `set`-Methode müssen denselben CLS-Typ haben – nämlich den Typ der Eigenschaft.

Tab. 2.1: Typen und Konventionen der Common Language Specification (Forts.)

Eigenschaften einer Klasse müssen sich in ihren Namen unterscheiden. Unterschiedliche Typen sind hier nicht ausreichend.

Da der Zugriff auf Eigenschaften über Methoden stattfindet, gilt: Wenn eine Klasse eine Eigenschaft mit dem Namen *Eigenschaft* definiert, sind die Bezeichner `get_`*`Eigenschaft`* und `set_`*`Eigenschaft`* innerhalb dieser Klasse für die Zugriffsfunktionen auf diese Eigenschaft reserviert.

Eigenschaften lassen sich indizieren.

Zugriffsfunktionen für Eigenschaften müssen dem Namensschema `get_`*`Eigenschaft`*, `set_`*`Eigenschaft`* folgen.

Aufzählungen

Als zugrunde liegende Typen sind ausschließlich `byte`, `short`, `int` und `long` verwendbar.

Die Elemente eines Aufzählungstyps sind Konstanten, die den der Aufzählung zugrunde liegenden Typ tragen.

Aufzählungen können keine Schnittstellen implementieren.

Mehrere Elemente einer Aufzählung können identische Werte haben.

Aufzählungen müssen von `System.Enum` abgeleitet sein (wofür der C#-Compiler implizit sorgt).

Ausnahmen

lassen sich signalisieren und abfangen.

Eigene Klassen für Ausnahmen müssen von `System.Exception` abgeleitet sein.

Schnittstellen

können die Implementation anderer Schnittstellen voraussetzen.

können Eigenschaften, Ereignisse und virtuelle Methoden definieren. Die Implementation dieser Elemente ist Sache der abgeleiteten Klasse.

Ereignisse

müssen entweder keine oder beide Methoden zum An- und Abmelden von Clients definieren. Beide Methoden erwarten einen Parameter, nämlich eine Referenz auf eine von `System.Delegate` abgeleitete Klasse.

Für die Methoden ist das folgende Namensschema obligatorisch: `add_`*`Ereignisname`*, `remove_`*`Ereignisname`*, `raise_`*`Ereignisname`*.

Selbstdefinierte Attribute

Hier sind ausschließlich die folgenden Typen verwendbar: `string`, `char`, `bool`, `byte`, `short`, `int`, `long`, `float`, `double`, `enum` (eines CLS-verträglichen Typs) und `object`.

Delegationen

lassen sich anlegen und aufrufen.

Tab. 2.1: Typen und Konventionen der Common Language Specification (Forts.)

Bezeichner
Für das erste Zeichen eines Bezeichners gibt es einige Einschränkungen.
Zwischen Groß- und Kleinschreibung wird nicht unterschieden. Zwei Bezeichner innerhalb eines Geltungsbereichs, die sich lediglich durch Groß- und Kleinschreibung voneinander unterscheiden, sind deshalb nicht zulässig.

Tab. 2.1: Typen und Konventionen der Common Language Specification (Forts.)

2.2.4 Das Virtual Execution System (VES)

Das Virtual Execution System implementiert das virtuelle Objektsystem. Es besteht im Wesentlichen aus einer Execution Engine (EE), die ihrerseits für das NGWS-Laufzeitsystem verantwortlich ist. Die EE führt die Anwendungen aus, die in C# geschrieben und kompiliert wurden.

Das VES umfasst:

- die Intermediate Language (IL) - eine Zwischensprache, die vom Design her ein breites Spektrum von Compilern abdeckt. Im Lieferumfang der NGWS-Implementation für Windows sind Compiler für C++, Visual Basic und C# enthalten, die IL generieren können.
- das Laden von verwaltetem Code - dazu gehört das Auflösen von Namen, das Anlegen von Objektvariablen im Hauptspeicher sowie das Einsetzen eines Vorspanns (aus Maschinenbefehlen) in die Methoden zum Aufruf des JIT-Compilers. Die für Klassen zuständige Laderoutine kümmert sich des Weiteren um die Sicherheit: Sie übernimmt Konsistenzprüfungen und sorgt für die Einhaltung der im Rahmen der Typdefinition aufgestellten Verfügbarkeitsregeln.
- die Umwandlung von IL-Code in Maschinenbefehle per JIT - IL-Code ist weder mit dem traditionellen Bytecode für Interpreter noch mit Tree-Code vergleichbar. Die Umwandlung in Maschinencode stellt eine echte Kompilierung dar.
- das Laden von Metadaten, Prüfung der Typen und Prüfung der Integrität von Methoden.
- Garbage Collection und Ausnahmebehandlung - beide Dienste basieren auf der Interpretation des Stacks, dessen Aufbau sich bei verwaltetem Code schrittweise verfolgen lässt. Damit das Laufzeitsystem die einzelnen Stackbereiche (»Frames«) auseinander halten kann, muss ein Code-Manager vorhanden sein, der entweder vom Compiler oder vom JIT gestellt wird.
- Profiling und Debugging – diese beiden Dienste sind von den Informationen abhängig, die der C#-Compiler zur Verfügung stellt. Hier geht es um zwei Tabellen: eine für die Zuordnung von Quelltextzeilen zu Adressen im IL-Code

und eine für die Zuordnung von Code zu Adressen auf dem Stack. Beide Tabellen müssen während der Umwandlung von IL-Code in Maschinenbefehle neu berechnet werden.

- die Verwaltung von Threads und Kontexten sowie Fernzugriffen - diese Dienste stellt das VES verwaltetem Code ebenfalls zur Verfügung.

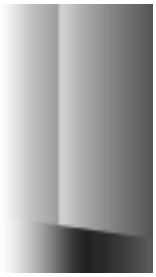
Auch wenn diese Liste nicht vollständig ist, sollte sie doch einen guten Eindruck davon vermitteln, wie das NGWS-Laufzeitsystem durch die Infrastruktur des VES unterstützt wird. Es steht mit Sicherheit zu erwarten, dass über das NGWS noch komplette Bücher geschrieben werden, die sich dann auch mit den einzelnen Teilen intensiver beschäftigen.

2.3 Zusammenfassung

Dieses Kapitel hat Ihnen das NGWS-Laufzeitsystem in einer Art Rundgang präsentiert und seine Arbeitsweise beim Erstellen, Kompilieren und der Verbreitung von C#-Programmen vorgestellt. C#-Compiler erzeugen keine Maschinenbefehle, sondern IL-Code, der gemeinsam mit Metadaten zur Beschreibung der verwendeten Typen in einer ausführbaren Datei gespeichert wird. Ein JIT-Compiler extrahiert den IL-Code und setzt ihn mit Hilfe der Metadaten in Maschinenbefehle um. Mit dem JIT Compiler Manager können Sie festlegen, welcher der drei JIT-Compiler des NGWS-Laufzeitsystems zum Einsatz kommen soll.

Im zweiten Teil dieses Kapitels ging es um die hinter dem Laufzeitsystem stehende Theorie: das Virtuelle Objektsystem (VOS) und seine Teile. Für das Design von Klassenbibliotheken besonders interessant ist die Common Language Specification (CLS), die Regeln zur sprachübergreifenden Nutzung von Klassen festlegt. Schließlich folgten noch einige Details des Virtual Execution Systems (VES), das eine Implementation des VOS durch das NGWS-Laufzeitsystem darstellt.

Kapitel 3



Die erste C#-Anwendung

Wie für Programmierbücher üblich, wird auch dieses Buch die ersten Gehversuche mit der neuen Programmiersprache C# in Form eines »Hello World«-Programms unternehmen. Das Kapitel selbst ist kurz, es dient eigentlich nur dem Zweck, Ihnen die grundlegenden Bestandteile einer C#-Anwendung vorzustellen und einen ersten Eindruck davon zu vermitteln, wie man in C# eine Anwendung schreibt, kompiliert und einfache Ein- und Ausgaben bewerkstelligt.

3.1 Wahl eines Editors

Nicht einmal ein bekennender Notepad-Enthusiast, wie der Autor dieses Buchs, wird Ihnen diesen Editor für die Bearbeitung von C#-Quellcode empfehlen. Das liegt schlicht und einfach daran, dass es sich bei C# um eine waschechte Compilersprache handelt, deren Compiler mit Fehlermeldungen nicht gerade geizig umgeht – wer von C/C++ kommt, weiß, was das heißt.

Als gute Wahl kommen eine ganze Reihe von Editoren in Frage. So können Sie Ihr gutes altes Visual C++ 6.0 so umkonfigurieren, dass es mit C#-Quelldateien zurechtkommt, oder Sie arbeiten gleich mit dem neuen Visual Studio 7. Weiterhin können Sie natürlich auch jeden anderen Editor benutzen, der Zeilennummern kennt, Code farbig unterlegen, Werkzeuge (den Compiler) aufrufen kann und eine halbwegs brauchbare Suchfunktion zu bieten hat. Ein gutes Beispiel für einen solchen Editor ist CodeWright, dessen Arbeitsbereich in Abbildung 3.1 zu sehen ist.

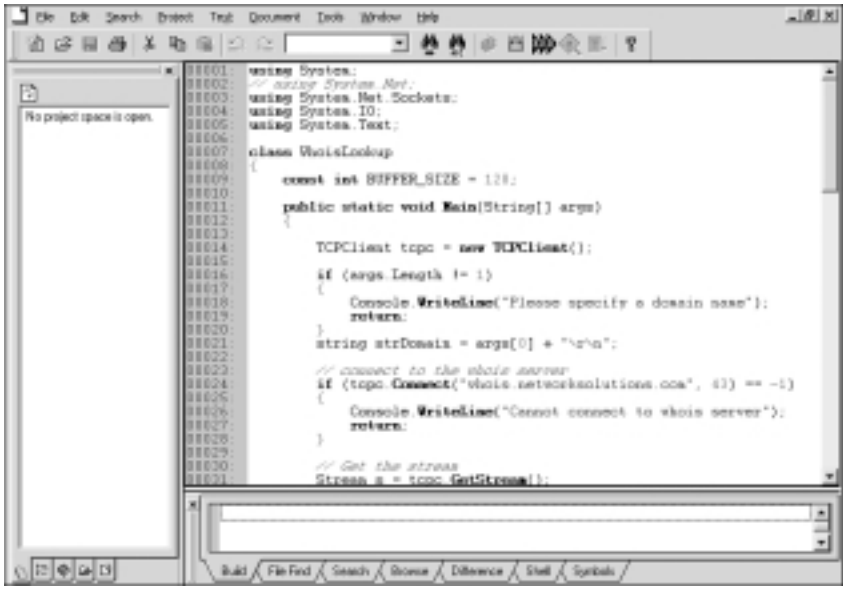


Bild 3.1: CodeWright, einer der vielen Editoren, die eine gute Wahl für die Bearbeitung von C#-Quellcode darstellen

Es versteht sich, dass keiner der erwähnten Editoren wirklich notwendig ist, um ein C#-Programm zu schreiben – im Prinzip kommt man auch mit Notepad aus. Falls Sie aber planen, ernsthafter mit C# zu programmieren, sollten Sie sich für eine komfortablere Alternative entscheiden.

3.2 Hello World

Nach diesem kurzen Ausflug in die Welt der Editoren zurück zu »Hello World«, der berühmtesten aller kleinen Anwendungen. Listing 3.1 zeigt die kürzeste C#-Formulierung für dieses Programm. Speichern Sie den Code in einer Datei namens `helloworld.cs`, damit Sie ihn später kompilieren können.

Listing 3.1: Das Programm »Hello World« in seiner einfachsten Gestalt

```
1: class HelloWorld
2: {
3:   public static void Main()
4:   {
5:     System.Console.WriteLine("Hello World");
6:   }
7: }
```

Der C#-Compiler erwartet, dass Sie Codeblöcke (Anweisungen) in geschweifte Klammern (`{` und `}`) setzen. Selbst wer noch nie ein C#- oder C++-Programm gesehen hat, müsste somit erkennen, dass die Methode `Main` der Anweisung `class` untergeordnet ist, weil sie in deren Codeblock definiert ist.

Der Startpunkt einer jeden (als eigenständiges Programm ausführbaren) C#-Anwendung ist die statische Methode `Main`, die in einer der Klassen des Programms definiert sein muss. Umgekehrt darf diese Methode immer nur von einer Klasse gestellt werden, es sei denn, Sie teilen dem Compiler explizit mit, welche `Main`-Methode er als Startpunkt verwenden soll (im Fehlerfall macht sich der Compiler natürlich durch eine entsprechende Meldung bemerkbar).

Im Gegensatz zu C++ beginnt `Main` in C# mit einem großgeschriebenen »M«. Mit dem Codeblock dieser Methode startet und endet Ihr Programm. Natürlich können Sie von diesem Codeblock aus weitere Methoden aufrufen, wie im Beispiel die statische Methode `WriteLine`, oder Objekte anlegen und deren Methoden ausführen.

Als Ergebnistyp für `Main` ist der Datentyp `void` vereinbart:

```
public static void Main()
```

Obwohl C++-ProgrammiererInnen die äußere Gestalt solcher Anweisungen bestens vertraut sein dürfte, benötigen diejenigen, die von einer anderen Programmiersprache kommen, wahrscheinlich noch eine Erläuterung der beiden Modifi-

zierer. `public` erklärt die Methode als »öffentlich«, so dass sie von jeder anderen Methode aus aufrufbar ist (und damit auch von der Laderoutine), gleich ob diese derselben Klasse angehört oder nicht. Ergänzend besagt `static`, dass die Methode zur Ausstattung der Klasse selbst gehört und für ihren Aufruf nicht eigens eine Instanz der Klasse angelegt werden muss. Anstelle eines Objektbezeichners kann somit auch der Klassenbezeichner stehen:

```
HelloWorld.Main();
```

Es ist natürlich alles andere als klug, diese Anweisung in die Methode `Main` zu setzen: Die unvermeidliche Folge wäre eine endlose Rekursion – und somit ein Stacküberlauf.

Ein weiterer wichtiger Aspekt ist der Ergebnistyp. Für die Methode `Main` haben Sie die Wahl zwischen `void` und `int`. Bei Deklaration mit `void` gibt die Methode keinen Wert zurück, bei Deklaration mit `int` eine Ganzzahl, die im Allgemeinen den Fehlerstatus beim Beenden der Anwendung ausdrückt. Damit lässt sich `Main` insgesamt auf zwei verschiedene Arten deklarieren:

```
public static void Main()
public static int Main()
```

Als C++-ProgrammiererIn werden Sie wahrscheinlich bereits ahnen, was als Nächstes kommt: Ja, man kann einer Anwendung Kommandozeilenparameter in Form eines Arrays übergeben. Die Deklaration sieht dann so aus:

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

Hier ist nicht die richtige Stelle, um den Zugriff auf diese Werte im Einzelnen zu diskutieren. Soviel sei aber gesagt: Als C++-ProgrammiererIn wird es Sie wahrscheinlich überraschen, dass C# den Aufrufpfad der Anwendung im Gegensatz zu C++ *nicht* in diesem Array übergibt, sondern sich wirklich nur auf die Kommandozeilenparameter beschränkt.

Nach dieser nun doch nicht so kurz gewordenen Einführung in die formalen Besonderheiten der Methode `Main`, ein Blick auf die einzige wirklich substanzielle Codezeile, die letztlich die Meldung »Hello World« auf den Bildschirm schreibt:

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

Wäre da nicht der Qualifizierer `System`, würde man sofort darauf kommen, dass `WriteLine` eine statische Methode der Klasse `Console` verkörpert. Wofür steht denn `System`? Nun, `System` ist der Namensraum, in dem die Klasse `Console` definiert ist. Da es unpraktisch ist, diesen Bezeichner für jeden Zugriff auf `Console` zu notieren, sieht C# die Möglichkeit vor, einen Namensraum auch als Ganzes zu importieren. Listing 3.2 zeigt, wie der Import vor sich geht:

Listing 3.2: Einen Namensraum in die Anwendung importieren

```
1: using System;
2:
3: class HelloWorld
4: {
5:   public static void Main()
6:   {
7:     Console.WriteLine("Hello World");
8:   }
9: }
```

Wie Sie sehen, umfasst die Importdeklaration nichts weiter als die Direktive `using` und den Bezeichner des zu importierenden Namensraums. Von nun an lassen sich die Elemente aus diesem Namensraum auch ohne explizite Qualifikation notieren. Aus dem riesigen Pool an Namensräumen, den das NGWS-Subsystem bereitstellt, werden Sie im Verlauf dieses Buchs nur einige wenige Elemente konkret kennenlernen. Kapitel 8, »Komponenten mit C# schreiben«, wird Sie allerdings mit dem nötigen Handwerkzeug zur Definition eigener Namensräume ausrüsten.

3.3 Kompilieren der Anwendung

Da zum Installationsumfang des NGWS-Laufzeitsystems auch alle notwendigen Compiler (VB, C++ und C#) gehören, sind Sie vom Prinzip her nicht auf den Kauf eines weiteren Entwicklungswerkzeugs angewiesen, um Ihre C#-Programme in die Sprachebene der IL (Intermediate Language) zu übersetzen. Sollten Sie noch nie eine Anwendung mit einem kommandozeilenorientierten Compiler übersetzt haben (und Make-Dateien zwar zu Ihrem Vokabular, nicht jedoch zu Ihren sinnlichen Erfahrungen zählen), werden Sie nun Ihr Debüt feiern.

Öffnen Sie ein Fenster mit der MS-DOS-Eingabeaufforderung und wechseln Sie in das Verzeichnis, das die zuvor gespeicherte Datei `helloworld.cs` enthält. Dann tippen Sie das folgende Kommando:

```
csc helloworld.cs
```

Es bewirkt, dass `helloworld.cs` kompiliert und zu der ausführbaren Datei `helloworld.exe` gebunden wird. Nachdem der Code fehlerfrei ist (sofern Sie ihn richtig abgetippt haben), bewältigt der C#-Compiler den Übersetzungsvorgang, wie in Abbildung 3.2 gezeigt, ohne das geringste Murren.

Nun ist es soweit. Sie können Ihre erste C#-Anwendung starten: Sobald Sie den Befehl `helloworld` in die Kommandozeile eingegeben haben, müsste die Ausgabe `Hello World` am Bildschirm erscheinen.

Bild 3.2: Die Anwendung mit dem Kommandozeilen-Compiler *csc.exe* übersetzen

Bevor es weitergeht: Probieren Sie doch noch den folgenden Compilerschalter aus:

```
csc /out:hello.exe helloworld.cs
```

Der Schalter ermöglicht es Ihnen, den Namen der ausführbaren Datei frei zu wählen. Dieser wahrlich unscheinbare Zusatz wird sich für den weiteren Einsatz des Compilers für die Codebeispiele in diesem Buch noch als recht wertvoll erweisen.

3.4 Ein- und Ausgabe

Was die bisherige Diskussion betrifft, so haben Sie an Code nichts weiter als die Ausgabe einer literalen Zeichenfolge über die Konsole zu sehen bekommen. Obwohl dieses Buch die Konzepte der C#-Programmierung darzustellen versucht, und nicht die Programmierung mit Benutzerschnittstellen, werden Sie nun erst einmal eine Handvoll einfacher Methoden kennenlernen, die Ihnen die Abwicklung von Ein- und Ausgaben über die Konsole gestatten – nämlich die Äquivalente der C-Funktionen `scanf` und `printf` bzw. der C++-Funktionen `cin` und `cout`. (VB ist vom Kern her nicht auf die konsolenorientierte Programmierung ausgelegt, weshalb sich hier auch keine Äquivalente angeben lassen.)

Um Ihr Programm mit der Außenwelt in Kontakt treten zu lassen, genügt es vom Prinzip her, wenn es Ausgaben produzieren und Eingaben des Benutzers entgegennehmen kann. Listing 3.3 zeigt, wie man den Benutzer zur Eingabe seines Namens veranlasst, den Namen einliest und als Teil einer persönlichen Begrüßungsformel wieder ausgibt.

Listing 3.3: Eine Eingabe von der Konsole lesen

```
 1: using System;
 2:
 3: class InputOutput
 4: {
 5:   public static void Main()
 6:   {
 7:     Console.Write("Bitte geben Sie Ihren Namen ein: ");
 8:     string strName = Console.ReadLine();
 9:     Console.WriteLine("Hallo " + strName);
10:   }
11: }
```

Zeile 7 enthält den Aufruf von `Write`, einer weiteren statischen Methode von `Console`, die für Textausgaben über die Konsole zuständig ist. Im Unterschied zu `WriteLine` verzichtet `Write` auf einen Zeilenvorschub nach Ausgabe des letzten Zeichens. Die Ausgabe mit `Write` bewirkt hier also, dass der Benutzer seinen Namen im Anschluss an die Eingabeaufforderung, d.h. in die gleiche Zeile, eingibt.

Sobald der Benutzer seinen Namen eingegeben und die Eingabetaste betätigt hat, liest das Programm die Eingabe mit der Methode `ReadLine` in die String-Variable `strName` und gibt deren Wert verkettet mit dem Literal `"Hallo "` mit der bereits bekannten Methode `WriteLine` wieder über die Konsole aus (vgl. Abbildung 3.3).

Bild 3.3: Die veränderte Version der Anwendung »Hello« übersetzen und ausführen

Das wäre es fast schon gewesen. Es fehlt eigentlich nur noch, dass Sie dem Benutzer mehrere Werte als formatierte Ausgabe präsentieren können. Listing 3.4 zeigt ein Beispiel:

Listing 3.4: Formatierte Ausgaben

```
 1: using System;
 2:
 3: class InputOutput
 4: {
 5:   public static void Main()
 6:   {
 7:     Console.Write("Bitte geben Sie Ihren Namen ein: ");
 8:     string strName = Console.ReadLine();
 9:     Console.WriteLine("Hallo {0}",strName);
10:   }
11: }
```

Zeile 9 enthält eine `WriteLine`-Anweisung, deren erster Parameter eine Formatbeschreibung darstellt und deren zweiter Parameter den in dieser Formatierung auszugebenden Wert liefert. Die Formatbeschreibung lautet:

```
"Hallo {0}"
```

Die Zeichenfolge `{0}` fungiert in dieser Beschreibung als Platzhalter für den Wert des zweiten Parameters. `WriteLine` lässt in Erweiterung dieses Schemas bis zu drei Parameter zu, die sich in der Formatbeschreibung über Platzhalter repräsentieren lassen:

```
Console.WriteLine("Hallo {0} {1} aus {2}", sVorname, sNachname, sStadt);
```

Es dürfte Sie nicht überraschen, dass man hier nicht nur Zeichenfolgen, sondern auch Werte anderer Typen einsetzen kann – Typen werden im Kapitel 4, »Die Typen von C#«, noch ausführlicher vorgestellt.

3.5 Kommentare

Ein wichtiger Bestandteil von Quelltexten sind Kommentare. Sie informieren gewissermaßen »vor Ort« über spezielle Details der Implementation, durchgeführte Änderungen und so weiter. Grundsätzlich bleibt es natürlich Ihnen selbst überlassen, ob Sie Kommentare benutzen und was Sie in einen Kommentar hineinschreiben, Sie müssen sich nur an die von C# vorgeschriebene Syntax halten, wenn Sie Kommentare setzen. Listing 3.5 zeigt die beiden Möglichkeiten, in C# Kommentare zu platzieren:

Listing 3.5: Kommentare in einen Quelltext einfügen

```
 1: using System;
 2:
 3: class HelloWorld
 4: {
 5:   public static void Main()
 6:   {
 7:     // das ist ein Kommentar in Zeilensyntax
 8:     /* dieser Kommentar ist in Blocksyntax gehalten
 9:        und kann auch mehrere Zeilen umfassen */
10:     Console.WriteLine(/*"Hello World"*/);
11:   }
12: }
```

In der Zeilensyntax sorgt der Kommentaroperator `//` dafür, dass der Compiler den Rest der jeweiligen Zeile als Kommentar betrachtet:

```
int nMyVar = 10;  // blah blah
```

Beachten Sie, dass Sie mit dieser Syntax Zeilen entweder ganz oder ab einer gewissen Spalte bis zum Zeilenende als Kommentar ausblenden können. Dasselbe Konzept findet sich auch in C++ und in Visual Basic (hier wird der Kommentar allerdings durch ein Hochkomma eingeleitet). Falls Sie einen mehrzeiligen Kommentar einfügen oder einen beliebigen Block auskommentieren wollen, verwenden Sie besser die Blocksyntax mit den beiden Kommentaroperatoren `/*` und `*/`. Der Compiler ignoriert in diesem Fall alles, was zwischen diesen beiden Operatoren steht. Exakt dieselbe Blocksyntax für Kommentare ist auch in C und C++ (nicht jedoch in VB) zulässig, so dass Sie in allen drei Programmiersprachen (C, C++ und C#) in dieselbe Falle tappen können. Angenommen, Sie haben in einer beliebigen Codesequenz bereits einen Block auskommentiert:

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

und wollen eben mal schnell zu Testzwecken einen weiteren größeren Block auskommentieren, in dem jedoch der erste Block enthalten ist:

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

Wenn Sie den größeren Block nun einfach durch die beiden Kommentaroperatoren kennzeichnen, entsteht das Problem, dass der Compiler nach Eintritt in den Kommentarblock den nächsten schließenden Kommentaroperator – also den des eingeschlossen Kommentarblocks – als Blockende betrachtet. Damit wird der restliche Code bis zum eigentlichen schließenden Kommentaroperator für den

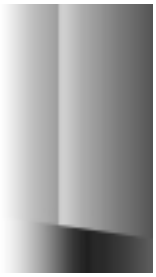
Compiler sichtbar und kann ihm eine Reihe recht interessanter Fehlermeldungen abtrotzen – zumindest jedoch eine Meldung über den vermeintlich ungepaarten Operator `*/`. Seien Sie also auf der Hut.

3.6 Zusammenfassung

In diesem Kapitel haben Sie Ihre erste C#-Anwendung kompiliert und gestartet: die Anwendung »Hello World«. Anhand dieser zu einigem Ruhm gekommenen kleinen Anwendung haben Sie die Methode `Main` kennengelernt, den definierten Startpunkt – und Endpunkt – eines jeden C#-Programms. `Main` kann so deklariert sein, dass diese Methode entweder keinen Ergebniswert zurückliefert oder aber eine ganzzahlige Fehlernummer. Zudem lässt sich bei Bedarf ein Array des Typs `string` als Aufrufparameter deklarieren, so dass sich mit dieser Methode auch Kommandozeilenparameter auswerten lassen.

Ausgehend von dieser Anwendung haben Sie noch einige Ein- und Ausgabemethoden der Klasse `Console` kennengelernt. Sie ermöglichen Ihnen, halbwegs vernünftige Beispielprogramme für die Konsole zu schreiben, wenn Sie die Sprache C# erlernen. Später werden Sie als Benutzerschnittstelle natürlich WFC, WinForms oder ASP+ verwenden.

Kapitel 4



Die Typen von C#

Nachdem Sie nun wissen, wie man ein einfaches Programm in C# schreibt, stelle ich Ihnen nun das Typensystem von C# vor. Dieses Kapitel zeigt den Einsatz der verschiedenen Wert- und Referenztypen und erläutert, was der als Boxing und Unboxing bezeichnete Mechanismus für Sie tun kann. Allzu viele Praxisbeispiele werden Sie auf den folgenden Seiten nicht finden – dafür aber eine Menge allgemeiner Informationen darüber, wie man in Programmen aus diesen Typen das Beste herausholt.

4.1 Wertbehaftete Typen

Eine Variable, die einen wertbehafteten Typ trägt, speichert einen konkreten Wert. Weil der Compiler jeden Einsatz uninitialisierter Variablen in Berechnungen als fehlerhaft moniert – Sie also zwingt, Variablen vor ihrer Verwendung zu initialisieren –, kann das von anderen Programmiersprachen her gewohnte Problem nicht initialisierter Variablen in C# per se nicht auftreten.

Die Wertzuweisung an eine Variable mit wertbehaftetem Typ ist daher ein Kopiervorgang, bei dem der zugewiesene Wert an sich kopiert wird. Bei der Zuweisung eines Referenztyps wird dagegen nur eine weitere Referenz auf ein und denselben Wert generiert. Der betroffene Wert verbleibt an Ort und Stelle und kann nun sowohl über die ursprüngliche als auch über die neue Referenz angesprochen werden – es gibt ja zwei Variablen, die auf ihn referieren.

Die wertbehafteten Typen von C# lassen sich in drei Kategorien unterteilen:

- einfache Typen
- strukturierte Typen (`struct`)
- Aufzählungen

4.1.1 Einfache Typen

Die von C# zur Verfügung gestellten einfachen Typen haben einige gemeinsame Charakteristika. Zum einen handelt es sich bei ihnen ausnahmslos um Aliase der vom NGWS-Subsystem unterstützten Typen, zum zweiten werden konstante Ausdrücke, die aus Werten mit einfachen Typen zusammengesetzt sind, bereits während der Compilierung ausgewertet (und nicht erst zur Laufzeit). Des Weiteren lassen sich alle einfachen Typen mit Literalen initialisieren.

Die einfachen Typen von C# lassen sich folgendermaßen gruppieren:

- Ganzzahltypen
- logischer Typ (`bool`)
- Zeichentyp (`char`), als Spezialfall eines ganzzahligen Werts

- Gleitkommatypen
- den Typ `decimal`

Ganzzahltypen

C# stellt insgesamt neun Ganzzahltypen zur Verfügung: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` und den in einem eigenen Abschnitt beschriebenen Typ `char`. Die Charakteristika und Wertebereiche dieser Typen sind:

- `sbyte` repräsentiert vorzeichenbehaftete Integer mit 8 Bit und einem Wertebereich von -128 bis 127.
- `byte` repräsentiert vorzeichenlose Integer mit 8 Bit und einem Wertebereich von 0 bis 255.
- `short` steht für vorzeichenbehaftete Integer mit 16 Bit und einem Wertebereich von -32.768 bis 32.767.
- `ushort` ist das vorzeichenlose Gegenstück zu `short` und hat einen Wertebereich von 0 bis 65.535.
- `int` repräsentiert vorzeichenbehaftete Integer mit 32 Bit und einem Wertebereich von -2.147.483.648 bis 2.147.483.647.
- `uint` ist die vorzeichenlose Variante von `int` mit einem Wertebereich von 0 bis 4.294.967.295.
- `long` steht für vorzeichenbehaftete Integer mit 64 Bit und einem Wertebereich von -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807.
- `ulong` bezeichnet vorzeichenlose Integer mit 64 Bit und einem Wertebereich von 0 bis 18.446.744.073.709.551.615.

C- und VB-ProgrammiererInnen werden sich vermutlich über die Wertebereiche von `int` und `long` wundern: Im Gegensatz zu anderen Programmiersprachen ist C# *nicht* von der Größe eines Maschinenwortes abhängig, weshalb `long` hier unabhängig von der Zielplattform 64 Bit hat.

Logischer Typbool

Der Datentyp `bool` repräsentiert die Booleschen Werte *True* und *False*. Variablen dieses Typs können Sie sowohl die Literale *True* und *False* als auch Boolesche Ausdrücke als Wert zuweisen:

```
bool bTest = (80 > 90);
```

Im Gegensatz zu C und C++ betrachtet C# nicht einfach jeden Integerwert ungleich 0 als *True* – und eine implizite Konvertierung zwischen `bool` und anderen Integertypen, die diese von den beiden anderen Programmiersprachen her gewohnte Konvention hinterrücks wieder einführen würde, gibt es nicht.

char

Der Datentyp `char` repräsentiert ein einzelnes Unicode-Zeichen. Solche Zeichen haben eine Breite von 16 Bit, was zur Darstellung praktisch jeden Schriftzeichens jeder auf dieser Welt existierenden Sprache ausreichend ist. Die Zuweisung eines Wertes an eine Variable vom Typ `char` kann so aussehen:

```
char chSomeChar = 'A';
```

Alternativ ist die Definition von Zeichenkonstanten über die von C her bekannten Escape-Sequenzen möglich, wobei sich wahlweise das Präfix `\x` für die hexadezimale Notation und das Präfix `\u` für die Unicode-Notation einsetzen lassen:

```
char chSomeChar = '\x65';    // 'e'
char chSomeChar = '\u0065';  // 'e'
```

Beim Präfix `\x` nimmt der Compiler gegebenenfalls selbst die Erweiterung auf 16 Bit vor, weshalb `\x65`, `\x065` und `\x0065` identische Werte darstellen. Das Präfix `\u` erfordert ebenfalls eine hexadezimale Notation, nur sind dort in jedem Fall vier Stellen gefordert: `\u65` bemängelt der Compiler demnach als fehlerhaft.

C# definiert keine impliziten Konvertierungen zwischen `char` und den anderen Integertypen. `char` lässt sich hier also nicht einfach als weiterer Integertyp behandeln (und stellt deshalb einen weiteren Bereich in C# dar, in dem C-Programmierer von ihren liebgewordenen Gewohnheiten abgehen müssen). Explizite Typumwandlungen sind dagegen sehr wohl möglich:

```
char chSomeChar = (char)65;
int nSomeInt = (int)'A';
```

Die von C her bekannten Escape-Sequenzen für Zeichenkonstanten übernimmt C# komplett. Wenn Sie Ihr Gedächtnis in dieser Hinsicht ein wenig auffrischen wollen oder müssen, hilft Tabelle 4.1 weiter.

Escape-Sequenz	Zeichenkonstante
`\'`	Einfaches Hochkomma
`\”`	Anführungszeichen
`\\`	Umgekehrter Schrägstrich
`\0`	Null (Unicode-Wert 0)
`\a`	Alert (Piepton auf dem Terminal)
`\b`	Rückschritt (Backspace)
`\f`	Seitenvorschub (Formfeed)
`\n`	Zeilenvorschub (Newline)

Tab. 4.1: Escape-Sequenzen für Zeichenkonstanten

Escape-Sequenz	Zeichenkonstante
`\r`	Wagenrücklauf (Carriage Return)
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator

Tab. 4.1: Escape-Sequenzen für Zeichenkonstanten (Forts.)

Gleitkommatypen

Zur Darstellung von Gleitkommawerten definiert C# zwei Typen namens `float` und `double`, die sich sowohl in ihrer Genauigkeit als auch hinsichtlich ihrer Wertebereiche voneinander unterscheiden:

- `float` hat einen Wertebereich von $1.5 \cdot 10^{-45}$ bis $3.4 \cdot 10^{38}$ mit einer Genauigkeit von sieben Ziffern
- `double` hat einen Wertebereich von $5.0 \cdot 10^{-324}$ bis $1.7 \cdot 10^{308}$ mit einer Genauigkeit von 15 bis 16 Ziffern

Mögliche Ergebnisse von Berechnungen mit diesen Datentypen sind:

- positive und negative 0
- positive und negative Unendlichkeit
- numerisch nicht darstellbarer Wert (NaN)
- die endliche Menge der regulären Gleitkommawerte in der gegebenen Genauigkeit

Falls ein numerischer Ausdruck auch nur einen Gleitkommawert erhält, setzt der Compiler vor der Berechnung alle anderen Werte des Ausdrucks implizit in Gleitkommawerte um.

decimal

Der Datentyp `decimal` ist für Finanzberechnungen und andere Operationen vorgesehen, bei denen es auf maximale Präzision ankommt. Variablen dieses Typs haben eine Breite von 128 Bit, einen Wertebereich von rund $1.0 \cdot 10^{-28}$ bis $7.9 \cdot 10^{28}$ und eine Genauigkeit von 28 bis 29 Ziffern. Bitte beachten Sie, dass Ziffern nicht in jedem Fall mit Dezimalstellen gleichzusetzen sind: 28 Dezimalstellen Genauigkeit stellen hier die absolute Obergrenze dar.

Absolut gesehen ist der Wertebereich von `decimal` offensichtlich erheblich kleiner als der von `double`, bei der Präzision ist das Gegenteil der Fall. Aus diesem Grund nimmt C# keine implizite Typumwandlung zwischen den beiden Typen vor: In der einen Richtung könnte sonst ein Überlauf entstehen, in der anderen ein Genauigkeitsverlust. Explizite Typumwandlungen sind dagegen möglich.

Da der Compiler Gleitkomma-Konstanten normalerweise als `double` behandelt, sollten Sie für Initialisierungen von Variablen des Typs `decimal` das Suffix `m` verwenden:

```
decimal decMyValue = 1.0m;
```

Dieses Suffix weist den Compiler an, das Literal von vornherein als `decimal` zu interpretieren.

4.1.2 Strukturierte Typen

Ein strukturierter Typ kann in C# Konstruktoren, Konstanten, Felder, Methoden, Eigenschaften, Indizierer, Operatoren und verschachtelte Typen deklarieren. Auch wenn sich das zunächst nach einer kompletten Klassendefinition anhört: In C# ist `struct` ein wertbehafteter Typ, `class` dagegen ein Referenztyp. (Achtung: In C++ lassen sich auch Klassen mit dem Schlüsselwort `struct` definieren, obwohl das in der Praxis selten geschieht.)

Was bei C# als Grundgedanke hinter `struct` steht, sind leichtgewichtige Objekte wie *Point*, *FileInfo* und so weiter, die man des Öfteren in größerer Zahl braucht. Die Definition über `struct` spart Speicher, weil hier im Gegensatz zu `class`-Objekten keine weiteren Referenzen erzeugt werden müssen, und das kann in einem Array mit einigen tausend Objekten sehr wohl einen gehörigen Unterschied ausmachen.

Das folgende Listing zeigt die Definition eines einfachen strukturierten Typs, der IP-Adressen mit vier Feldern des Typs `byte` repräsentiert. Einen Konstruktor habe ich hier genauso wie Methoden außen vor gelassen, weil dafür exakt dieselben Regeln wie für Klassen gelten – mehr dazu im nächsten Kapitel.

Listing 4.1: Definition eines einfachen strukturierten Typs

```
 1: using System;
 2:
 3: struct IP
 4: {
 5:   public byte b1,b2,b3,b4;
 6: }
 7:
 8: class Test
 9: {
10:   public static void Main()
11:   {
12:     IP myIP;
13:     myIP.b1 = 192;
14:     myIP.b2 = 168;
15:     myIP.b3 = 1;
16:     myIP.b4 = 101;
```

```
17:     Console.Write("{0}.{1}.",myIP.b1,myIP.b2);
18:     Console.Write("{0}.{1}",myIP.b3,myIP.b4);
19:   }
20: }
```

4.1.3 Aufzählungstypen

Wenn Sie einen eigenen Datentyp deklarieren wollen, dessen Wertebereich aus einer festen Menge benannter Konstanten bestehen soll, dann ist `enum` das Mittel der Wahl. In ihrer einfachsten Form sieht eine solche Deklaration so aus:

```
enum MonthNames { January, February, March, April };
```

Da diese Deklaration die Standardvorgaben benutzt, basieren *MonthNames*-Werte auf dem Typ `int`, wobei das Element *January* den Wert 0 hat. Die folgenden Bezeichner assoziiert der Compiler mit einem jeweils um eins höheren Wert (*February* = 1, *March* = 2, *April* = 3). Um dem ersten Element der Aufzählung explizit einen bestimmten Wert zuzuweisen, schreiben Sie:

```
enum MonthNames { January = 1, February, March, April };
```

Der Compiler setzt dann *February* mit 2 gleich, *March* mit 3 und so weiter. Die explizite Wahl bestimmter Werte ist in einer solchen Deklaration für jede der Konstanten möglich – sogar dann, wenn ein einzelner Wert öfter als einmal erscheint:

```
enum MonthDays { January = 31, February = 28, March = 31, April = 30 };
```

Bei Bedarf können Sie außerdem noch einen anderen Datentyp als `int` festlegen:

```
enum MonthDays : byte { January = 31, February = 28, March = 31, April =
30 };
```

Neben `int` und `byte` sind hier auch `short` und `long` verwendbar.

4.2 Referenztypen

Im Gegensatz zum wertbehafteten Typ speichert ein Referenztyp die von ihm repräsentierten Daten nicht direkt, sondern in Form eines Verweises auf einen Speicherbereich, der die Daten enthält. C# definiert die folgenden Referenztypen:

- `object`
- `class`
- Schnittstellen
- Delegationen

- `string`
- Arrays

4.2.1 object

Der Typ `object` stellt die grundlegende Basisklasse und damit die Mutter aller anderen Klassentypen dar. Und weil dem so ist, können Sie einer Variablen des Typs `object` Werte *jedes* anderen Typs zuweisen – auch einen numerischen Wert:

```
object theObj = 123;
```

In diesem Zusammenhang eine Warnung an alle C++-ProgrammiererInnen: `object` hat nichts, aber auch gar nichts mit *void*-Zeigern zu tun, nach denen Sie wahrscheinlich bereits gesucht haben. Zeiger im klassischen Sinne gibt es bei C# definitiv nicht.

Der Typ `object` wird unter anderem beim *boxing* eingesetzt – dem Verkapseln eines Wertes als Objekt. Eine Beschreibung dieses Konzepts finden Sie gegen Ende dieses Kapitels.

4.2.2 class

Mit `class` deklarierte Typen können Mitgliedsdaten, Mitgliedsfunktionen und eingeschachtelte Typen enthalten. Der Terminus Mitgliedsdaten (»data member« dient bei C# als Oberbegriff für Datenfelder, Konstanten und Ereignisse; Mitgliedsfunktionen (»member functions«) sind Methoden, Eigenschaften, Indizierer, Operatoren, Konstruktoren und Destruktoren. Mit `class` und `struct` deklarierte Typen sind in puncto Funktionalität weitgehend gleich; allerdings geht es, wie im vorangehenden Abschnitt erläutert, bei `struct` um Werte, bei `class` dagegen um Referenzen.

C# beschränkt sich im Gegensatz zu C++ auf einfache Vererbung: Die Deklaration einer von mehreren Basisklassen abgeleiteten Klasse ist hier also nicht möglich. Was C# dagegen ohne Weiteres erlaubt, ist die Deklaration einer Klasse als gemeinsame Ableitung mehrerer Interfaces (siehe nächster Abschnitt).

Da Klassen und ihrem Einsatz bei der Programmierung das Kapitel 5, »Klassen«, gewidmet ist, beschränkt sich dieser Abschnitt auf einen allgemeinen Überblick: Er will lediglich zeigen, wie sich die Klassen von C# in das Typenschema der Sprache einpassen.

4.2.3 Schnittstellen

Eine Schnittstelle (Interface) deklariert einen Referenztyp, der ausschließlich aus abstrakten Mitgliedern aufgebaut ist. Ähnliche Konzepte in C++ sind die Mitglieder eines `struct`-Typs und auf 0 gesetzten Methode. Falls Ihnen das nichts sagt:

In C# besitzt eine Schnittstelle lediglich eine Signatur, aber keinerlei Implementation oder irgendeinen Codeanteil. Woraus unter anderem folgt, dass man in C# von einer Schnittstelle keine Instanzen anlegen kann, sondern ausschließlich Instanzen von Objekten, die von dieser Schnittstelle abgeleitet sind.

Da Sie eine Schnittstelle mit Methoden, Indizierern und Eigenschaften ausstatten können, stellt sich die Frage, wo dann der Unterschied zu einer Klasse liegt? Eine Schnittstelle ist eine reine Deklaration, eine Klasse dagegen eine Definition. Der Unterschied kommt hauptsächlich bei Ableitungen zum Tragen: Wie zuvor erwähnt, lässt C# bei Ableitungen nur eine Klasse als Basis zu, Schnittstellen können es dagegen beliebig viele sein.

Nachdem man bei der Ableitung von Schnittstellen offensichtlich die gesamte Implementation selbst beisteuern muss, wäre noch zu klären, welche Vorteile dieser Ansatz haben soll. Das geht am einfachsten mit Hilfe eines Beispiels aus dem NGWS-Programmgerüst: Viele der dort verwendeten Klassen implementieren beispielsweise die Schnittstelle *IDictionary*, an die man dann über eine einfache explizite Typumwandlung herankommt:

```
IDictionary myDict = (IDictionary)EinObjektMitIDictionaryImplementation;
```

Nun können Sie mit der *IDictionary*-Implementation des Objekts arbeiten. Die Crux daran: Obwohl *IDictionary* von vielen Klassen (und sicher nicht immer auf die gleiche Weise) implementiert wird, ist die Schnittstelle und damit der Umgang mit den entsprechenden Objekten grundsätzlich gleich. Wenn Sie also den Zugriff auf *IDictionary* einmal implementiert haben, können Sie den Code jederzeit auch an anderer Stelle wiederverwenden.

Wer eigene Schnittstellen in seinen Klassen einsetzen will, sollte natürlich einiges über objektorientiertes Design wissen – und damit über ein Thema, das den Rahmen dieses Buches definitiv sprengt. Wie die Syntax zur Definition von Interfaces aussieht, ist dagegen schnell erklärt. Der folgende Codeauszug definiert eine Schnittstelle namens *IFace* mit nur einer Methode:

```
interface IFace
{
  void ShowMyFace();
}
```

Wie bereits erwähnt, können Sie mit dieser Definition allein noch kein Objekt anlegen – sie dient ausschließlich als Basis zur Ableitung einer Klasse. Die Klasse muss ihrerseits die (von der Schnittstelle definierte abstrakte) Methode *ShowMyFace* implementieren:

```
class CFace:IFace
{
  public void ShowMyFace()
  {
    Console.WriteLine("implementation");
  }

}
```

Und noch einmal: Der einzige Unterschied zwischen Klassen und Schnittstellen besteht darin, dass Schnittstellen für ihre Mitglieder keine Implementation bereitstellen. Alle weiteren Details finden Sie in Kapitel 5, »Klassen«.

4.2.4 Delegationen

Eine Delegation (Delegate – direkt übersetzt: »Abgesandter«) verkapselt eine Methode mit einer bestimmten Signatur. Im Wesentlichen handelt es sich dabei um die typensichere und überwachte Variante der aus C und C++ bekannten Funktionszeiger. Die Instanz einer Delegation kann sowohl eine statische als auch eine Instanz-gebundene Methode verkapseln.

Obwohl man Delegationen ganz normal für Methoden verwenden kann, liegt der Schwerpunkt ihres Einsatzes in C#-Programmen auf Ereignissen – also auf Rückruffunktionen von Klassen. Details dazu finden Sie gleichfalls in Kapitel 5, »Klassen«.

4.2.5 string

Auch wenn das C-Programmierer vielleicht überrascht: Der Typ `string` zur Speicherung und Manipulation von Zeichenfolgen gehört bei C# zu den elementaren Datentypen. Er ist eine direkte Ableitung von `object`, die als »versiegelt« gekennzeichnet ist – d.h. nicht als Basisklasse für eigene Ableitungen eingesetzt werden kann. Wie alle anderen Typen von C# stellt auch `string` einen Alias für eine vordefinierte Klasse dar, nämlich:

```
System.String;
```

Der Einsatz gestaltet sich ausgesprochen einfach:

```
string myString = "irgendein Text";
```

Das Aneinanderhängen zweier Zeichenfolgen ist auch nicht wesentlich komplizierter:

```
string myString = "irgendein Text" + " und noch etwas";
```

Für den Zugriff auf einzelne Zeichen einer Zeichenfolge lässt sich der Indizierer dieser Klasse verwenden:

```
char chFirst = myString[0];
```

Und der Vergleich zweier Zeichenfolgen geschieht mit dem von dieser Klasse als eigene Variante bereitgestellten Operator ==:

```
if (myString == yourString) ...
```

Es sei angemerkt, dass sich dieser Operator auf die Werte, also die Zeichenfolgen, bezieht, obwohl `string` als Klasse implementiert ist und somit einen Referenztyp darstellt. (Der Standardoperator == von C# würde bei einem Referenztyp prüfen, ob die Speicheradressen, auf die *myString* und *yourString* verweisen, identisch sind.)

Details zu den einzelnen Methoden der Klasse `string` finden Sie in den Beispielen der folgenden Kapitel. (Tatsächlich kommen nur wenige Beispiele dieses Buchs ohne `string` und die dazugehörigen Methoden aus.)

4.2.6 Arrays

Arrays stellen eine Sammlung von gleichartigen Variablen dar, die hier als *Elemente* bezeichnet werden, sämtlich den gleichen Typ haben müssen und über einen Index angesprochen werden. Was den Typ der Elemente (auch als »Typ des Arrays« bezeichnet) betrifft, gibt es keine Beschränkungen: Ein Array kann Integer-Werte, Zeichenfolgen oder Objekte beliebigen Typs speichern.

Die Zahl der Dimensionen eines Arrays – also die Zahl der Indizes, die für den Zugriff auf ein einzelnes Element angegeben werden müssen –, wird bei C# als *Rang* bezeichnet. Mit Abstand am häufigsten kommen eindimensionale Arrays zum Einsatz, die folglich den Rang eins haben; mehrdimensionale Arrays haben dagegen einen Rang größer eins. Die Zählung der Indizes beginnt mit 0 und läuft bis zur der Größe der jeweiligen Dimension minus eins.

Damit erst einmal genug der grauen Theorie. Die Deklaration eines eindimensionalen Arrays mit gleichzeitiger Initialisierung der Elemente kann so aussehen:

```
string[] arrLanguages = { "C", "C++", "C#" };
```

Tatsächlich stellt diese Notation die Abkürzung dar für:

```
arrLanguages[0]="C"; arrLanguages[1]="C++"; arrLanguages[2]="C#";
```

Mehrdimensionale Arrays lassen sich ebenfalls in einem Schritt deklarieren und initialisieren:

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

Der Compiler betrachtet dies als Abkürzung für:

```
arr[0,0] = 0; arr[0,1] = 1;
arr[1,0] = 2; arr[1,1] = 3;
arr[2,0] = 4; arr[2,1] = 5;
```

Wenn die Größe eines Arrays feststeht, im Rahmen der Deklaration aber keine explizite Initialisierung gewünscht ist, kann auch der `new`-Operator zum Einsatz kommen:

```
int [,] myArr = new int[5,3];
```

Und schließlich: Wenn sich die Größe der einzelnen Dimensionen erst zur Laufzeit bestimmen lässt, dann wird `new` mit Variablen anstelle von Konstanten aufgerufen:

```
int nVar = 5;
// ...
int[] arrToo = new int[nVar];
```

Wie am Anfang dieses Abschnitts erwähnt, lassen sich Arrays als Sammelbecken für Alles und Jedes verwenden – vorausgesetzt, sämtliche Elemente haben denselben Typ. Um Werte mit beliebigen Typen in ein Array hineinstecken zu können, müssen Sie lediglich dafür sorgen, dass es den Typ `object` trägt – für den Rest sorgt das Boxing von C#.

4.3 Boxing und Unboxing

Nachdem dieses Kapitel eine ganze Reihe von Typen vorgestellt und die Unterschiede zwischen Variablen- und Referenztypen erläutert hat, sollte klar sein: Solange es um Geschwindigkeit geht, wird man wertbehaftete Typen gegenüber Referenztypen bevorzugen, weil sich hinter Werten auch in C# letztlich nichts weiter als einfache Speicherblocks verbergen. Mitunter kann es aber auch von Vorteil sein, die von Objekten gebotenen Bequemlichkeiten für einen wertbehafteten Typ zu nutzen.

Kein Problem. C# definiert einen Mechanismus, der sozusagen das Bindeglied zwischen wertbehafteten Typen und Referenztypen liefert, indem er die Konvertierung wertbehafteter Typen in den Typ `object` und wieder zurück ermöglicht. Anders gesagt: In C# lässt sich alles und jedes letztlich als Objekt verpacken, sofern ein solches gefordert ist – und es gibt auch eine Rückfahrkarte.

4.3.1 Umwandeln von Werten in Objekte

Der Terminus *Boxing* steht bei C# für die implizite Umwandlung eines Werts in den Typ `object` und damit bildhaft eben für das Einpacken in einen Karton: Dabei wird eine neue Instanz des Typs `object` angelegt, das ein Element des entsprechenden wertbehafteten Typs besitzt, und diesem Element eine Kopie des Werts zugewiesen. Ein einfaches Beispiel dazu:

```
int nFunny = 2000;
object oFunny = nFunny;
```

Die Zuweisung in der zweiten Zeile stellt eine solche implizite Typumwandlung dar: Sie generiert das Objekt und weist ihm *nFunny* als Wert zu. Danach liegen sowohl die Integer- als auch die Objektvariable auf dem Stack, und außerdem existiert eine Kopie von *nFunny* auf dem Heap des Programms.

Was impliziert das? Nun, die Werte der beiden Variablen sind unabhängig voneinander – *oFunny* enthält keinen Querverweis auf *nFunny*, sondern einen Querverweis auf eine Kopie dieses Werts. Der folgende Codeauszug illustriert die Konsequenzen:

```
int nFunny = 2000;
object oFunny = nFunny;
oFunny = 2001;
Console.WriteLine("{0} {1}", nFunny, oFunny);
```

Wie nach den vorangehenden Erläuterungen wohl auch nicht anders zu erwarten, bleibt *nFunny* von der Wertänderung von *oFunny* unberührt, und das Programm gibt »2000 2001« aus. Solange Sie diesen Punkt im Kopf behalten, steht einer Nutzung der Objektfunktionalität für wertbehaftete Typen nichts weiter im Wege.

4.3.2 Umwandeln von Objekten in Werte

Das Unboxing (Auspacken) eines Wertes ist im Gegensatz zu seiner Verpackung als Objekt eine Operation, die Sie explizit anfordern müssen – nicht zuletzt, weil der Compiler von Ihnen wissen will, welchen Datentyp Sie in dem Objekt erwarten. C# prüft bei einer solchen Anforderung, ob das Objekt auch tatsächlich einen Wert des gewünschten Datentyps enthält: Wenn ja, wird dieser Wert ausgepackt und an der Variable als Kopie zugewiesen. Ein Beispiel dazu:

```
int nFunny = 2000;
object oFunny = nFunny;
int nFunny2 = (int)oFunny;
```

Auf die Anforderung eines nicht im Objekt enthaltenen Datentyps – beispielsweise mit

```
double nNotSoFunny = (double)oFunny;
```

reagiert das NGWS-Laufzeitsystem mit einer Ausnahme des Typs `InvalidCastException`. (Details dazu finden Sie in Kapitel 7, »Ausnahmen behandeln«.)

4.4 Zusammenfassung

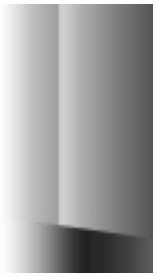
Dieses Kapitel hat die von C# vorgegebenen Typen vorgestellt. Zu den einfachen Typen gehören die Integertypen, `bool`, `char`, die Gleitkommatypen und `decimal`, mit denen man die meisten mathematischen und finanzmathematischen Berechnungen und logische Bedingungen formulieren wird.

Vor der Einführung der Referenztypen ging es kurz um `struct`, einen engen Verwandten von `class`. Typen dieser Art verhalten sich im Prinzip wie Klassen, zählen aber zu den wertbehafteten Typen. Ihr erheblich geringerer Überbau macht sie vor allem dann zum Mittel der Wahl, wenn es um größere Mengen kleinerer Objekte geht.

Im Abschnitt über Referenztypen ging es zuerst um `object`, die Mutter aller Klassen in C#, die außerdem beim Boxing und Unboxing von Werten zum Einsatz kommt. Des Weiteren wurden Delegationen, Strings und Arrays besprochen.

Die Art von Datentyp, die angehende C#-ProgrammiererInnen wohl auch noch in ihren Träumen verfolgen wird, ist die Klasse. Sie bildet sozusagen der Kern der objektorientierten Programmierung mit dieser Sprache. Das nächste Kapitel geht denn auch in Länge und Breite auf dieses Konzept mit all seinen Möglichkeiten ein.

Kapitel 5



Klassen

Nachdem sich das vorangehende Kapitel ausführlich mit den Datentypen von C# und ihrem Einsatz beschäftigt hat, geht es nun um das wichtigste Konstrukt dieser Sprache: Die Klasse. Ohne eine Klasse ließe sich ein C#-Programm nicht einmal kompilieren. Das folgende Material geht davon aus, dass Ihnen die grundlegenden Bausteine von Klassen geläufig sind: Methoden, Eigenschaften, Konstruktoren und Destruktoren. C# bereichert diese Palette sowohl um Indizierer als auch um Ereignisse.

Im Einzelnen geht es in diesem Kapitel um:

- den praktischen Einsatz von Konstruktoren und Destruktoren
- das Schreiben von Methoden für Klassen
- Zugriffsfunktionen für Eigenschaften
- die Implementation von Indizierern
- die Definition von Ereignissen und die Benachrichtigung von Clients über Delegationen
- Modifizierer für Klassen, ihre Methoden und Zugriffsfunktionen

5.1 Konstruktoren und Destruktoren

Als Erstes kommen bei einer Klasse beziehungsweise ihren Objekten die Anweisungen aus dem Konstruktor der Klasse zur Ausführung – und zwar noch bevor Sie die erste Methode oder Eigenschaft des Objekts aufrufen können. Das gilt auch, wenn Sie für eine Klasse überhaupt keinen eigenen Konstruktor definieren, weil der Compiler dann einen Standardkonstruktor einsetzt:

```
class TestClass
{
  public TestClass(): base() {}  // vom Compiler eingesetzt
}
```

Konstruktoren haben grundsätzlich denselben Namen wie ihre Klasse und liefern keinen Funktionswert zurück. Im Normalfall sind sie öffentlich deklariert und werden zur Initialisierung von Elementvariablen verwendet:

```
public TestClass()
{
  // Initialisierungscode für
  // Elementvariablen usw.
}
```

Für Klassen, die ausschließlich statische Elemente definieren (also Methoden, Eigenschaften, Elementvariablen usw., die für den Typ an sich definiert sind – nicht für einzelne Objekte des jeweiligen Typs), kann man einen `private`-Konstruktor definieren:

```
private TestClass() {}
```

Auch wenn das einen Vorgriff auf den Abschnitt dieses Kapitels darstellt, in dem es um Modifizierer geht: `private` teilt dem Compiler mit, dass dieser Konstruktor ausschließlich innerhalb der Klasse zur Verfügung steht und vom restlichen Programm aus nicht aufgerufen werden kann. Deshalb lassen sich auch keine Objekte dieser Klasse anlegen.

Bei Konstruktoren sind Sie nicht auf die parameterlose Variante beschränkt. Sie können ohne weiteres Konstruktoren definieren, die eine beliebige Zahl von Argumenten zur Initialisierung von Datenfeldern usw. erwarten:

```
public TestClass(string strName, int nAge) { ... }
```

In C und C++ stattet man Klassen recht oft mit einer zusätzlichen Initialisierungsmethode aus, weil Konstruktoren dort ebenfalls keine direkten Ergebnisse liefern (über die sich signalisieren ließe, ob bestimmte Voraussetzungen – wie etwa das Vorhandensein anderer Objekte – für die Konstruktion erfüllt sind). In C# sind Konstruktoren zwar ebenfalls »ergebnislos«, separate Initialisierungsmethoden aber trotzdem selten notwendig, weil man hier im Falle eines Falles innerhalb von Konstruktoren eine selbstdefinierte Ausnahme auslösen kann. (Details dazu finden Sie in Kapitel 7, »Ausnahmen behandeln«.)

Der Destruktor einer Klasse stellt das Gegenteil ihres Konstruktors dar, ist also für die Deinitialisierung zuständig. Der Name dieser Routine ist ebenfalls vorgegeben: Er besteht aus dem Klassennamen mit einer vorangestellten Tilde:

```
public ~TestClass()
{
  // Aufräumen
}
```

Hier kann sich sehr wohl die Notwendigkeit zum Schreiben einer separaten Aufräumfunktion ergeben – nämlich dann, wenn Objekte der Klasse vergleichsweise kostbare Ressourcen (wie größere Bereiche auf dem Heap) belegen. Wozu dieses Extra, wenn die C#-Laufzeitumgebung doch einen Garbage Collector enthält, der automatisch den Destruktor von Objekten aufruft, die den Geltungsbereich verlassen haben? Weil diese Automatik nur in bestimmten Zeitabständen in Gang gesetzt wird und außerdem noch von anderen Umständen wie der aktuellen Speicherbelegung abhängt – mithin durchaus die Möglichkeit besteht, dass ein Objekt Ressourcen wesentlich länger belegt als vom Programmierer beabsichtigt.

Für das Aufräumen zu exakt definierten Zeitpunkten empfiehlt es sich deshalb, eine separate Methode zu schreiben, die dann ihrerseits auch vom Destruktor aufgerufen wird:

```
public void Release()
{
  // alle vom Objekt belegten Ressourcen freigeben
}

public ~TestClass()
{
  Release();
  // weitere Aufräumarbeiten, falls notwendig
}
```

Rein technisch gesehen ist der Aufruf von *Release* im Destruktor nicht unbedingt notwendig, weil sich die Garbage Collection von sich aus um die Freigabe des Objekts kümmert. Nur zu welchem Zeitpunkt, ist die Frage – und außerdem ist es bewährte Programmierpraxis, das Aufräumen nicht zu vergessen.

5.2 Methoden

Mit einem Konstruktor und einem Destruktor ausgerüstete Objekte lassen sich wunderbar anlegen und wieder freigeben, bieten aber außer Speicherplatzverbrauch definitiv keine Funktionalität. In den meisten Fällen wird man den größten Teil der Funktionalität in Form von Methoden implementieren. Einige Beispiele für statische Methoden haben Sie in diesem Buch bereits zu sehen bekommen. Solche Methoden sind allerdings Ausstattung der Klasse (also des Datentyps an sich) und nicht der einzelnen Instanz (Objekt).

Damit die unvermeidlichen Fragen zu Methoden so schnell und übersichtlich wie möglich beantwortet werden, folgen hier drei Abschnitte:

- Parameter von Methoden
- Überschreiben von Methoden
- Verbergen von Methoden

5.2.1 Parameter von Methoden

Eine Methode, die wechselnde Werte bearbeiten können soll, muss einerseits eine Möglichkeit zur Übergabe von Werten definieren, andererseits ein Ergebnis (in irgend einer Form) zurückliefern. Die folgenden drei Abschnitte beschäftigen sich mit den diversen Formen der Datenübergabe, die C# im Zusammenhang mit Methoden zu bieten hat:

- Eingangsparameter
- Referenzparameter
- Ausgangsparameter

Eingangsparameter

Diese Art von Parametern haben Sie bereits in den Beispielen der vorangehenden Kapitel gesehen. Hier geht es um die schlichte Wertübergabe. Der Compiler versorgt die Methode mit einer Kopie des übergebenen Werts. Das folgende Listing zeigt diese Technik:

Listing 5.1: Wertübergabe von Parametern

```
 1: using System;
 2:
 3: public class SquareSample
 4: {
 5:   public int CalcSquare(int nSideLength)
 6:   {
 7:     return nSideLength*nSideLength;
 8:   }
 9: }
10:
11: class SquareApp
12: {
13:   public static void Main()
14:   {
15:     SquareSample sq = new SquareSample();
16:     Console.WriteLine(sq.CalcSquare(25).ToString());
17:   }
18: }
```

Da *CalcSquare* einen echten Wert (exakter: die Kopie eines Werts) und keine Referenz auf eine Variable erwartet, ist zum Aufruf dieser Methode in Zeile 16 eine Konstante verwendbar. *CalcSquare* hat ein direktes Funktionsergebnis (Typ `int`), das erst in eine Zeichenfolge umgesetzt und dann an *Console.WriteLine* übergeben wird. Die explizite Deklaration einer temporären Variablen zum Zwischenspeichern des Ergebnisses von *CalcSquare* werden Sie in diesem Listing nicht finden, weil sie unnötig ist bzw. sich der Compiler um dieses Detail gegebenenfalls schon selbst kümmert.

Für C/C++-Programmierer stellen Eingangsparameter und die Wertübergabe absolut nichts Neues dar. Wenn Sie Ihre Erfahrungen mit Visual Basic gesammelt haben, müssen Sie vorsichtig sein: Solange Sie nicht explizit einen Modifizierer angeben, findet bei C# standardmäßig eine Wertübergabe statt – der C#-Compiler baut hier also ein implizites *ByVal* ein, und nicht ein *ByRef*.

Moment mal. Was geschieht bei der Übergabe eines *string*-Wertes oder eines beliebigen anderen Objekts? Geht es da nicht per se um Referenzen? Ganz recht. Hier sollte ich also meine zuvor gemachte Aussage relativieren: Für manche Datentypen heißt Wertübergabe: Übergabe einer Kopie der Referenz auf den Wert. Um etwas weiter auszuholen: Bei COM ist alles und jedes eine Schnittstelle, und jede Klasse kann eine oder mehrere Schnittstellen haben. Eine Schnittstelle ist nicht mehr und nicht weniger als eine Sammlung von Funktionszeigern; Daten enthält sie nicht. Das Kopieren einer solchen Zeigersammlung wäre schlichte Platzverschwendung, weshalb eine Methode bei der Wertübergabe lediglich die Startadresse dieser Sammlung zu sehen bekommt – also die Information, die auch der Aufrufer hat.

Referenzparameter

Mit Eingangsparametern und direkten Funktionsergebnissen lässt sich zwar bereits eine Menge anfangen, wenn es allerdings darum geht, den Wert einer Variable (also den Speicherbereich, der mit dieser Variable assoziiert ist) des Aufrufers zu verändern, sind die Grenzen erreicht. Für solche Aufgaben sind Referenzparameter (kurz: `ref`-Parameter) zuständig:

```
void MyMethod(ref int nInOut)
```

Da diese Methode eine echte Variable übergeben bekommt, deren Wert sie jederzeit lesen (und ändern) kann, muss der Aufrufer für die Initialisierung der Variablen sorgen. Ansonsten beschwert sich der Compiler. Das folgende Listing zeigt eine Methode, die mit einem `ref`-Parameter arbeitet.

Listing 5.2: Übergabe von Parametern als Referenz

```
 1: // class SquareSample
 2: using System;
 3:
 4: public class SquareSample
 5: {
 6:   public void CalcSquare(ref int nOne4All)
 7:   {
 8:     nOne4All *= nOne4All;
 9:   }
10: }
11:
12: class SquareApp
13: {
14:   public static void Main()
15:   {
16:     SquareSample sq = new SquareSample();
17:
18:     int nSquaredRef = 20;  // muss initialisiert werden
19:     sq.CalcSquare(ref nSquaredRef);
```

```
20:     Console.WriteLine(nSquaredRef.ToString());
21:   }
22: }
```

Wie in diesem Listing zu sehen, beschränken sich die Änderungen im Wesentlichen auf den Modifizierer `ref`, der sowohl bei der Definition als auch beim Aufruf anzugeben ist. Da die Variable *nSquaredRef* per Referenz übergeben wird, kann sie die Methode sowohl als Grundlage der Berechnung als auch zum Zurückliefern des Ergebnisses verwenden. In ernsthaften Anwendungen empfiehlt es sich allerdings *immer*, bei solchen Berechnungen voneinander getrennte Eingangs- und Ausgangsparameter zu verwenden.

Ausgangsparameter

Wie der Name bereits impliziert, sind mit `out` gekennzeichnete Parameter ausschließlich für die Rückgabe eines Ergebnisses gedacht. Ausgangsparameter (kurz: `out`-Parameter) stellen Referenzen auf Variablen des Aufrufers dar, in denen Methoden ihre Ergebnisse zurückgeben können. Werte *an* eine Methode übergeben kann man mit ihnen nicht, weshalb auf der Seite des Aufrufers auch keine Initialisierung der jeweiligen Variablen notwendig ist. Das folgende Listing zeigt eine Methode, die mit einem `out`-Parameter arbeitet.

Listing 5.3: Einsatz eines Ausgangsparameters

```
 1: using System;
 2:
 3: public class SquareSample
 4: {
 5:   public void CalcSquare(int nSideLength, out int nSquared)
 6:   {
 7:     nSquared = nSideLength * nSideLength;
 8:   }
 9: }
10:
11: class SquareApp
12: {
13:   public static void Main()
14:   {
15:     SquareSample sq = new SquareSample();
16:
17:     int nSquared;  // keine Initialisierung notwendig
18:     sq.CalcSquare(15, out nSquared);
19:     Console.WriteLine(nSquared.ToString());
20:   }
21: }
```

5.2.2 Überschreiben von Methoden

Die *Polymorphie* stellt eines der wichtigeren Prinzipien objektorientierter Programmierung dar. Wenn wir den theoretisch-philosophischen Überbau einmal außen vor lassen, bedeutet das, dass Sie in einer abgeleiteten Klasse Methoden der Basisklasse erneut definieren (also ersetzen) können – vorausgesetzt, der Schöpfer der Basisklasse hat die jeweilige Methode als überschreibbar gekennzeichnet. Diese Kennzeichnung geschieht in C# mit dem Schlüsselwort `virtual`:

```
virtual void CanBOverridden()
```

Wenn Sie diese Methode in einer abgeleiteten Klasse durch eine eigene Variante ersetzen wollen, stellen Sie der Deklaration das Schlüsselwort `override` voran:

```
override void CanBeOverridden()
```

Die Sichtbarkeit einer Methode lässt sich durch Überschreiben in abgeleiteten Klassen übrigens nicht ändern. Mehr dazu und zu den entsprechenden Modifizierern weiter hinten in diesem Kapitel.

Bemerkenswerter noch als die Möglichkeit, von einer Basisklasse vererbte Methoden in einer abgeleiteten Klasse umdefinieren zu können, ist der Effekt, dass die neuen Varianten dieser Methoden auch nach einer expliziten Typumwandlung in die Basisklasse noch greifen:

```
((BaseClass)DerivedClassInstance).CanBeOverridden();
```

Hier wird also die Methode *CanBeOverriden* der abgeleiteten Klasse aufgerufen – und nicht die Basisvariante. Das folgende Listing demonstriert dieses Prinzip etwas ausführlicher. Es definiert eine Basisklasse *Triangle*, deren Methode zur Flächenberechnung (*ComputeArea*) in einer abgeleiteten Klasse durch einen neue Variante ersetzt wird.

Listing 5.4: Überschreiben einer Methode der Basisklasse

```
 1: using System;
 2:
 3: class Triangle
 4: {
 5:   public virtual double ComputeArea(int a, int b, int c)
 6:   {
 7:     // Heronsche Formel
 8:     double s = (a + b + c) / 2.0;
 9:     double dArea = Math.Sqrt(s*(s-a)*(s-b)*(s-c));
10:     return dArea;
11:   }
12: }
13:
14: class RightAngledTriangle:Triangle
```

```
15: {
16:   public override double ComputeArea(int a, int b, int c)
17:   {
18:     double dArea = a*b/2.0;
19:     return dArea;
20:   }
21: }
22:
23: class TriangleTestApp
24: {
25:   public static void Main()
26:   {
27:     Triangle tri = new Triangle();
28:     Console.WriteLine(tri.ComputeArea(2, 5, 6));
29:
30:     RightAngledTriangle rat = new RightAngledTriangle();
31:     Console.WriteLine(rat.ComputeArea(3, 4, 5));
32:   }
33: }
```

Die Methode *Triangle.ComputeArea* erwartet drei Parameter vom Typ `int`, liefert ein Ergebnis des Typs `double` und ist öffentlich verfügbar. *RightAngledTriangle*, die Ableitung der Klasse *Triangle*, ersetzt die Methode *ComputeArea* durch eine eigene Variante, die eine wesentlich einfachere Flächenberechung ausführt. Die Methode *Main* des Programms legt von jeder der Klassen ein Objekt an und ruft dann die (klassenspezifische) Variante von *ComputeArea* auf.

Was vor allem wohl noch fehlt, ist eine Erläuterung zur Zeile 14 dieses Listings:

```
class RightAngledTriangle : Triangle
```

Der Doppelpunkt (:) in dieser Anweisung teilt dem Compiler mit, dass *RightAngledTriangle* von *Triangle* abgeleitet ist – und stellt auch bereits das gesamte Drumherum dar, das C# bei Ableitungen sehen will.

Wenn Sie sich die Implementation von *ComputeArea* in der Klasse *RightAngledTriangle* genauer ansehen, werden Sie feststellen, dass der dritte Parameter dort gar nicht ausgewertet wird. Er ließe sich für die Prüfung benutzen, ob das Dreieck auch wirklich rechtwinklig ist – und stellt eine gute Gelegenheit dar, Aufrufe der Basisklassenvariante von Methoden aus abgeleiteten Klassen heraus zu demonstrieren:

Listing 5.5: Aufruf der Basisklassenvariante

```
1: class RightAngledTriangle:Triangle
2: {
3:   public override double ComputeArea(int a, int b, int c)
4:   {
```

```
 5:     const double dEpsilon = 0.0001;
 6:     double dArea = 0;
 7:     if (Math.Abs((a*a + b*b - c*c)) > dEpsilon)
 8:     {
 9:       dArea = base.ComputeArea(a,b,c);
10:     }
11:     else
12:     {
13:        dArea = a*b/2.0;
14:     }
15:
16:     return dArea;
17:   }
18: }
```

Für diese Prüfung wird die Formel von Pythagoras benutzt, die für rechtwinklige Dreiecke das Ergebnis 0 liefert. Auf ein von Null (und einem hier als Minimalwert eingeführten Epsilon) abweichendes Ergebnis reagiert die Methode mit dem Aufruf der Basisklassenvariante:

```
dArea = base.ComputeArea(a, b, c);
```

Der springende Punkt ist hier der Qualifizierer `base`, über den sich zu jedem Zeitpunkt Elemente der Basisklasse ansprechen lassen. Derartige Rückgriffe wird man vor allem dann einsetzen, wenn die Funktionalität der Basisklasse gefordert ist (die man in einer abgeleiteten Klasse natürlich nicht duplizieren möchte).

5.2.3 Verbergen von Methoden

Eine zweiter Weg zum Ersatz von Methoden besteht darin, die Bezeichner der Basisklasse zu verbergen. Diese Alternative ist unter anderem bei Ableitungen von Klassen nützlich, die Sie nicht selbst geschrieben haben (und deren Methoden unter Umständen das Schlüsselwort `virtual` fehlt). Sehen Sie sich einmal das folgende Listing an, stellen Sie sich dabei vor, *BaseClass* sei von jemand anderem geschrieben worden, und Sie wollten nun *DerivedClass* von dieser Klasse ableiten:

Listing 5.6: Die abgeleitete Klasse implementiert eine in der Basisklasse nicht vorhandene Methode

```
1: using System;
2:
3: class BaseClass
4: {
5: }
6:
7: class DerivedClass:BaseClass
```

```
 8: {
 9:   public void TestMethod()
10:   {
11:     Console.WriteLine("DerivedClass::TestMethod");
12:   }
13: }
14:
15: class TestApp
16: {
17:   public static void Main()
18:   {
19:     DerivedClass test = new DerivedClass();
20:     test.TestMethod();
21:   }
22: }
```

In diesem Beispiel implementiert *DerivedClass* irgend ein zusätzliches Feature über die neue Methode *TestMethod*. So weit, so gut. Was aber, wenn der Entwickler von *BaseClass* selbst schon auf die Idee gekommen ist, *TestMethod* als Bezeichner für eine Methode zu verwenden, und dieser Methode auch noch dieselbe Signatur zu geben (also gleiche Parametertypen in gleicher Reihenfolge – oder eben überhaupt keine Parameter)? Das folgende Listing simuliert diese Konstellation:

Listing 5.7: Die Basisklasse implementiert eine gleichnamige Methode

```
 1: class BaseClass
 2: {
 3:   public void TestMethod()
 4:   {
 5:     Console.WriteLine("BaseClass::TestMethod");
 6:   }
 7: }
 8:
 9: class DerivedClass:BaseClass
10: {
11:   public void TestMethod()
12:   {
13:     Console.WriteLine("DerivedClass::TestMethod");
14:   }
15: }
```

Tatsächlich hätten Sie in einer klassischen Programmiersprache nun ein hübsches Problem am Hals. Kein Wunder, dass sich der C#-Compiler recht gesprächig zeigt:

```
hiding2.cs(13,14): warning CS0114: 'DerivedClass.TestMethod()' hides in-
herited member 'BaseClass.TestMethod()'. To make the current method
override that implementation, add the override keyword. Otherwise add
the new keyword.
```

Übersetzt: 'DerivedClass.TestMethod()' verbirgt das vererbte Mitglied 'BaseClass.TestMethod()'. Wenn Sie die Implementation [der Basisklasse] durch die aktuelle Methode ersetzen wollen, fügen Sie das Schlüsselwort `override` hinzu. Ansonsten verwenden Sie das Schlüsselwort `new`.

Mit dem Modifizierer `new` teilen Sie dem Compiler mit, dass die Methode der abgeleiteten Klasse die gleichnamige Methode der Basisklasse verbergen soll, ohne dass Änderungen am Code der Basisklasse oder an anderen Programmteilen, die diese Basisklasse verwenden, notwendig wären. Der folgende Code zeigt den Einsatz von `new`:

Listing 5.8: Verbergen der Methode der Basisklasse

```
 1: class BaseClass
 2: {
 3:   public void TestMethod()
 4:   {
 5:     Console.WriteLine("BaseClass::TestMethod");
 6:   }
 7: }
 8:
 9: class DerivedClass:BaseClass
10: {
11:   new public void TestMethod()
12:   {
13:     Console.WriteLine("DerivedClass::TestMethod");
14:   }
15: }
```

Wohlgemerkt: Eine mit `new` in einer Ableitung deklarierte Methode kommt nur dann zum Einsatz, wenn Sie die Objekte auch über die abgeleitete Klasse ansprechen:

```
DerivedClass newtest = new DerivedClass();
BaseClass oldtest = new DerivedClass();
newtest.TestMethod();// = DerivedClass.TestMethod
oldtest.TestMethod();// = BaseClass.TestMethod
((BaseClass)newtest).TestMethod();// = BaseClass.TestMethod
```

Dieses Verhalten unterscheidet sich grundlegend von Methoden, die in der Basisklasse als `virtual` deklariert und in der abgeleiteten Klasse mit `override` überschrieben werden. Dort ist garantiert, dass die jeweils »neueste« Variante einer Methode aufgerufen wird.

5.3 Eigenschaften

Es gibt in C# zwei Möglichkeiten, benannte Attribute von Klassen öffentlich verfügbar zu machen: als Datenelemente oder als Eigenschaften. Bei Datenelementen geht es schlicht um wertbehaftete Variablen, die über den Modifizierer `public` als öffentlich ausgewiesen werden. Eigenschaften haben dagegen nur indirekt etwas mit Speicherbereichen zu tun, weil sich hinter ihnen *Zugriffsfunktionen* verbergen.

Zugriffsfunktionen (Accessors) legen die Anweisungen fest, die eine Klasse beim lesenden bzw. schreibenden Zugriff auf eine Eigenschaft ausführt. Lesende Zugriffsfunktionen werden durch das Schlüsselwort `get` gekennzeichnet, schreibende Zugriffsfunktionen durch das Schlüsselwort `set`.

Bevor die Theorie wieder zu grau wird, ein praktisches Beispiel in Form einer Klasse *House* mit einer Eigenschaft *SquareFeet*, die sich sowohl lesen als auch (von außerhalb dieser Klasse) neu setzen lässt:

Listing 5.9: Implementation von Zugriffsfunktionen

```
 1: using System;
 2:
 3: public class House
 4: {
 5:   private int m_nSqFeet;
 6:
 7:   public int SquareFeet
 8:   {
 9:     get { return m_nSqFeet; }
10:     set { m_nSqFeet = value; }
11:   }
12: }
13:
14: class TestApp
15: {
16:   public static void Main()
17:   {
18:     House myHouse = new House();
19:     myHouse.SquareFeet = 250;
20:     Console.WriteLine(myHouse.SquareFeet);
21:   }
22: }
```

Der aktuelle Wert der Eigenschaft *SquareFeet* wird hier in einem privaten Feld (*m_nSqFeet*) gespeichert. Wenn Sie aus *SquareFeet* eine gewöhnliche öffentliche Elementvariable machen wollen würden, dann müssten Sie lediglich die Zugriffsfunktionen streichen und nur die Variablendeklaration übrig lassen:

```
public int SquareFeet;
```

Für einfache Variablen ist das auch durchaus in Ordnung. Wenn Sie allerdings Details über die innere Speicherstruktur Ihrer Klasse verbergen wollen, dann sollten Sie Zugriffsfunktionen verwenden. Die `set`-Funktion bekommt den neuen Wert als Parameter *value* übergeben (wobei dieser Name fix vorgegeben ist – siehe Zeile 10 des Listings).

Der Hauptzweck von Zugriffsfunktionen liegt natürlich nicht im Verbergen innerer Speicherstrukturen, sondern zum einen in der Möglichkeit, Lese- oder Schreibzugriffe explizit zu sperren:

- `get` implementiert, `set` nicht: Die Eigenschaft kann außerhalb der Klassendefinition nur gelesen, nicht aber verändert werden.
- `set` implementiert, `get` nicht: Die Eigenschaft kann außerhalb der Klassendefinition nur geschrieben, nicht aber wieder gelesen werden.
- `get` und `set` implementiert: Der Wert der Eigenschaft lässt sich außerhalb der Klassendefinition sowohl lesen als auch verändern.

Der andere und mindestens genauso wichtige Vorteil: `set` kann selbstverständlich Prüfungen enthalten und beispielsweise dafür sorgen, dass das Haus nicht versehentlich eine negative Quadratmeterzahl erhält; hinter `get` kann sich ebenso gut eine dynamische Eigenschaft verstecken – mithin ein Wert, der erst auf Anforderung berechnet und dann eventuell auch noch verifiziert wird. Eine Klasse *House* könnte bei Leseanforderungen beispielsweise die Quadratmeterzahlen sämtlicher in einer eigenen Liste gehaltenen Räume zusammenrechnen – und eine Klasse *Picture* ein Bild erst dann laden, wenn das Programm die Daten tatsächlich benötigt.

5.4 Indizierer

Wollten Sie Ihre Klassen nicht schon immer einmal auf einfachste Weise mit indizierten Zugriffen ausstatten – wie bei den Elementen eines Arrays? Mit C# geht das praktisch ohne Drumherum. Die grundlegende Syntax sieht so aus:

```
Attribute Modifizierer Deklarator { Deklarationen }
```

Ein einfaches Beispiel für eine Implementation:

```
public string this[int nIndex]
{
  get { ... }
  set { ... }
}
```

Dieser Indizierer liest bzw. schreibt einen Zeichenkette mit einem bestimmten Index. Attribute hat er keine, verwendet aber den Modifizierer `public`. Der Deklaratorteil besteht aus `string` als Typangabe und `this` für den Indizierer der Klasse.

Die Implementationsregeln für `get` und `set` sind dieselben wie bei Eigenschaften (was bedeutet, dass Sie Zugriffe wahlweise auf Lesen oder Schreiben einschränken können). Einen Unterschied gibt es allerdings: Bei der in eckigen Klammern anzugebenden Parameterliste haben Sie praktisch freie Wahl. Die einzigen Grenzen sind hier, dass diese Liste mindestens einen Parameter enthalten muss und sich die Modifizierer `ref` sowie `out` nicht verwenden lassen.

Indizierer haben keine eigenen Namen. Das Schlüsselwort `this` steht für den Indizierer der Schnittstelle, die die Klasse implementiert. Für Klassen, die mehrere Schnittstellen implementieren (also von mehreren Schnittstellen abgeleitet sind), kann hier der Name der Schnittstelle (in der Form *Schnittstellenname.this*) angegeben werden, für die der Indizierer zuständig ist.

Das folgende Listing demonstriert den Einsatz eines Indizierers anhand einer einfachen Klasse, die einen Hostnamen zu einer IP-Adresse auflöst – oder eben bei Hostnamen wie *www.microsoft.com* zu einer ganzen Liste von IP-Adressen, die dann über den Indizierer zugänglich sind.

Listing 5.10: Abfrage von IP-Adressen über einen Indizierer

```
 1: using System;
 2: using System.Net;
 3:
 4: class ResolveDNS
 5: {
 6:   IPAddress[] m_arrIPs;
 7:
 8:   public void Resolve(string strHost)
 9:   {
10:     IPHostEntry iphe = DNS.GetHostByName(strHost);
11:     m_arrIPs = iphe.AddressList;
12:   }
13:
14:   public IPAddress this[int nIndex]
15:   {
16:     get
17:     {
18:       return m_arrIPs[nIndex];
19:     }
20:   }
21:
22:   public int Count
23:   {
24:     get { return m_arrIPs.Length; }
```

```
25:   }
26: }
27:
28: class DNSResolverApp
29: {
30:   public static void Main()
31:   {
32:     ResolveDNS myDNSResolver = new ResolveDNS();
33:     myDNSResolver.Resolve("www.microsoft.com");
34:
35:     int nCount = myDNSResolver.Count;
36:     Console.WriteLine("{0} IP-Adressen für {1} gefunden:", nCount,
       "www.microsoft.com");
37:     for (int i=0; i < nCount; i++)
38:       Console.WriteLine(myDNSResolver[i]);
39:   }
40: }
```

Die zur Auflösung des Hostnamens verwendete Klasse *DNS* ist Teil des Namensraums *System.Net*. Da dieser Namensraum nicht in der NGWS-Kernbibliothek enthalten ist, muss die entsprechende Bibliothek explizit beim Aufruf des Compilers angegeben werden:

```
csc /r:System.Net.dll /out:resolver.exe resolver.cs
```

Die Logik des Programms ist recht geradlinig: In der Methode *Resolve* wird die statische Methode *GetHostByName* der Klasse *DNS* aufgerufen, die ein *IPHostEntry*-Objekt zurückgibt. Dieses Objekt enthält das Array *AddressList*, dessen Elemente *IPAddress*-Objekte sind, und um das es in diesem Beispiel geht: *AddressList* wird von *Resolve* in der objekteigenen Elementvariablen *m_arrIPs* gespeichert.

Nachdem das Array nun besetzt ist, kann die Anwendung die IP-Adressen über den Indizierer der Klasse *ResolveDNS* abzählen (Zeilen 37 und 38 im Listing). Dieser Indizierer implementiert nur eine `get`-Methode, weil das Verändern der IP-Adressen nicht sonderlich sinnvoll wäre. Details zu der verwendeten `for`-Schleife finden Sie übrigens in Kapitel 6, »Kontrollstrukturen«.

5.5 Ereignisse

Bei Klassen ergibt sich des Öfteren die Notwendigkeit, dass ihre Objekte sich bei anderen Teilen einer Anwendung (oder anderen Objekten, was in C# dasselbe ist) sozusagen aus eigenem Antrieb melden müssen. Wenn Sie Erfahrungen mit anderen Programmiersprachen gesammelt haben, wird Ihnen die Vielzahl der Wege geläufig sein, über die man solche Rückmeldungen zu Stande bringen kann – bei-

spielsweise mit Zeigern auf Rückruffunktionen oder den Event Sinks von ActiveX-Steuerelementen. C# stellt mit Ereignissen einen eigenen Mechanismus für solche Benachrichtigungen zur Verfügung.

Ereignisse lassen sich wahlweise als Elementvariablen oder als Eigenschaften einer Klasse definieren. Beiden Verfahren ist gemeinsam, dass der Typ eines Ereignisses mit `delegate` gekennzeichnet sein muss – dem Schlüsselwort, das in C# das Äquivalent zu Prototypen von Funktionszeigern darstellt.

Ereignisse können von einer beliebigen Zahl von Clients konsumiert werden (sind im Gegensatz zu den klassischen Funktionszeigern also nicht auf einen einzigen Abnehmer beschränkt); Clients können sich zu jedem Zeitpunkt mit einem Ereignis verbinden bzw. davon wieder lösen. Delegationen lassen sich sowohl als statische als auch als auf die jeweilige Objektinstanz bezogene Methoden implementieren, wobei die zweite Möglichkeit vor allem C++-Programmierern entgegenkommt.

Nachdem nun nicht nur alle wesentlichen Features von Ereignissen, sondern auch die der dazugehörigen Delegationen erwähnt worden sind, folgt anstelle weiterer Theorie ein praktisches Beispiel.

Listing 5.11: Implementation einer Ereignisbehandlung

```
 1: using System;
 2:
 3: // forward-Deklaration
 4: public delegate void EventHandler(string strText);
 5:
 6: class EventSource
 7: {
 8:   public event EventHandler TextOut;
 9:
10:   public void TriggerEvent()
11:   {
12:     if (null != TextOut) TextOut("Ereignis ausgelöst");
13:   }
14: }
15:
16: class TestApp
17: {
18:   public static void Main()
19:   {
20:     EventSource evsrc = new EventSource();
21:
22:     evsrc.TextOut += new EventHandler(CatchEvent);
23:     evsrc.TriggerEvent();
24:
25:     evsrc.TextOut -= new EventHandler(CatchEvent);
```

```
26:     evsrc.TriggerEvent();
27:
28:     TestApp theApp = new TestApp();
29:     evsrc.TextOut += new EventHandler(theApp.InstanceCatch);
30:     evsrc.TriggerEvent();
31:   }
32:
33:   public static void CatchEvent(string strText)
34:   {
35:     Console.WriteLine(strText);
36:   }
37:
38:   public void InstanceCatch(string strText)
39:   {
40:     Console.WriteLine("Instance " + strText);
41:   }
42: }
```

In Zeile 4 dieses Programms wird die Delegation (also der Prototyp der Ereignisbehandlungsroutine) deklariert. Diesen Typ verwendet die Klasse *EventSource* für das Element *TextOut* in Zeile 8. (Tatsächlich kann man die Deklaration von Delegationen als Deklaration eines speziellen Typs sehen, der dann seinerseits für die Deklaration von Ereignissen verwendbar ist.)

Die Klasse *EventSource* hat lediglich eine Methode, die zum Auslösen des Ereignisses dient. Bitte beachten Sie, dass *TriggerEvent* das Ereignis *TextOut* erst einmal prüfen muss: Wenn dort der Wert `null` steht, bedeutet das, dass sich zu diesem Zeitpunkt niemand für dieses Ereignis interessiert.

Die Klasse *TestApp* beherbergt in diesem Beispiel neben *Main* zwei Methoden mit einer zu dem Ereignis passenden Signatur. Die eine (*CatchEvent*) ist statisch, bezieht sich also auf die Klasse, die andere ist speizifisch für das jeweilige *TestApp*-Objekt.

Nach dem Anlegen eines *EventSource*-Objekts meldet *Main* erst einmal die statische Methode bei dem *TextOut*-Ereignis an:

```
evsrc.TextOut += new EventHandler(CatchEvent);
```

Im Weiteren wird also die *TestApp*-Methode *CatchEvent* als Reaktion auf das Ereignis aufgerufen. Wenn das zu irgend einem Zeitpunkt nicht weiter gewünscht sein sollte, dann ist eine Abmeldung fällig, die so aussieht:

```
evsrc.TextOut -= new EventHandler(CatchEvent);
```

Das An- und Abmelden von Methoden für die Ereignisbehandlung ist ausschließlich innerhalb der Klasse möglich, die die jeweiligen Methoden selbst definiert.

(Die Ereignisbehandlung einer anderen Klasse können Sie in *TestApp* also nicht verändern.)

Der zweite Teil von *Main* demonstriert schließlich, dass sich Ereignisse auch von Methoden behandeln lassen, die auf Objektvariablen einer Klasse bezogen sind: Er legt ein Objekt der Klasse *TestApp* an und registriert die Methode *InstanceCatch* dieses Objekts.

Was man mit Ereignissen und Delegationen in der Praxis anfangen kann, wird nicht nur von ASP+, sondern auch von den Windows Foundation Classes in aller Ausführlichkeit demonstriert.

5.6 Modifizierer

In den vorangehenden Abschnitten dieses Kapitels haben Sie bereits einige Modifizierer wie `public` und `virtual` zu Gesicht bekommen. Auf den folgenden Seiten geht es nun um eine vollständige Betrachtung, die der Übersichtlichkeit halber in drei Abschnitte unterteilt ist:

- Modifizierer für Klassen
- Modifizierer für Elemente
- Modifizierer für die Verfügbarkeit

5.6.1 Modifizierer für Klassen

Bei Klassen haben sich die bis dato vorgestellten Beispiele auf Modifizierer für die Verfügbarkeit beschränkt. C# definiert allerdings zwei Modifizierer, die ausschließlich auf Klassen anwendbar sind:

- `abstract` – für Klassen, deren wesentliche Eigenschaft daraus besteht, dass man sie ausschließlich als Basis für weitere Klassen, nicht aber zum Anlegen von Objekten verwenden kann. Ableitungen einer abstrakten Klasse, von denen sich Objekte anlegen lassen sollen, müssen alle in der Basisklasse als abstrakt gekennzeichneten Elemente implementieren.
- `sealed` – für Klassen, von denen sich keine weiteren Klassen ableiten lassen. Dieser Modifizierer wird von einigen Klassen des NGWS-Programmiergerüsts verwendet und ist nicht mit `abstract` kombinierbar.

Das folgende Beispiel demonstriert den Einsatz dieser beiden Modifizierer: Es leitet eine versiegelte Klasse von einer abstrakten Klasse ab (was man in der Praxis wohl nur selten auf derart direktem Wege machen wird).

Listing 5.12: Abstrakte und versiegelte Klassen

```
 1: using System;
 2:
 3: abstract class AbstractClass
 4: {
 5:   abstract public void MyMethod();
 6: }
 7:
 8: sealed class DerivedClass:AbstractClass
 9: {
10:   public override void MyMethod()
11:   {
12:     Console.WriteLine("versiegelte Klasse ");
13:   }
14: }
15:
16: public class TestApp
17: {
18:   public static void Main()
19:   {
20:     DerivedClass dc = new DerivedClass();
21:     dc.MyMethod();
22:   }
23: }
```

5.6.2 Modifizierer für Elemente

Bei Elementen bietet C# eine erheblich größere Auswahl von Modifizierern als bei Klassen. Einige dieser Modifizierer wurden bereits vorgestellt, der Rest wird im weiteren Verlauf dieses Buchs anhand praktischer Beispiele erläutert:

- `abstract` – für Methoden und Zugriffsfunktionen, die in der jeweiligen Klassendeklaration keine Implementation haben. Auf diese Weise deklarierte Elemente sind implizit `virtual` und müssen bei der Implementation in abgeleiteten Klassen mit `override` gekennzeichnet werden.
- `const` – für wertbehaftete Elemente und lokale Variablen. Auf diese Weise gekennzeichnete Ausdrücke werden zum Zeitpunkt der Kompilierung ausgewertet, weshalb sie keine Referenzen auf Variablen enthalten können.
- `event` – für die Deklaration von Variablen und Eigenschaften als Ereignis. Über Elemente dieses Typs werden Clients mit Ereignissen von Klassen und Objektvariablen verbunden.
- `extern` – teilt dem Compiler mit, dass eine Methode an anderer Stelle als im aktuellen Quelltext implementiert ist. Details dazu finden Sie in Kapitel 10, »Integration von nicht verwaltetem Code«.

- `override` – zum Überschreiben von Methoden und Zugriffsfunktionen, die in einer zugrunde liegenden Klasse als `virtual` deklariert sind. Die Signaturen (also Parameterzahl, -typen und -reihenfolge) dieser Methode bzw. Zugriffsfunktion und ihres Ersatzes müssen identisch sein.
- `readonly` – der Wert eines mit diesem Modifizierer deklarierten Elements lässt sich nur im Rahmen der Deklaration sowie im Konstruktor der Klasse ändern.
- `static` – für Elemente, die zur Klasse selbst (und nicht zu Objekten der Klasse) gehören. Dieser Modifizierer ist für wertbehaftete Variablen, Methoden, Eigenschaften, Operatoren und nicht zuletzt für Konstruktoren verwendbar.
- `virtual`-für Methoden und Zugriffsfunktionen, die sich in Ableitungen der Klasse ersetzen lassen.

5.6.3 Modifizierer für die Verfügbarkeit

Diese Modifizierer legen fest, auf welcher Ebene Elemente einer Klasse (wie wertbehaftete Felder, Methoden und Eigenschaften) verfügbar sind. Elemente, die nicht explizit mit einem solchen Modifizierer gekennzeichnet sind, stehen öffentlich zur Verfügung.

Die folgenden Modifizierer legen die Verfügbarkeit fest:

- `public` – das Element ist sowohl für Code innerhalb als auch außerhalb der Klasse verfügbar. Dieser Modifizierer stellt die niedrigste Stufe der Einschränkung dar.
- `protected` – das Element ist für Code innerhalb der Klasse und für Code innerhalb aller Ableitungen dieser Klasse verfügbar. Zugriffe von außerhalb der Klasse und ihrer Ableitungen sind nicht möglich.
- `private` – das Element ist ausschließlich für Code innerhalb der Klasse verfügbar, also weder für Code außerhalb der Klasse noch für Code innerhalb von Ableitungen.
- `internal` - das Element ist für Code innerhalb derselben NGWS-Komponente (Anwendung oder Bibliothek) verfügbar, nicht aber für Code außerhalb. Diese Einschränkung ließe sich als `public` auf der NGWS-Komponentenebene und `private` für den Rest der Welt bezeichnen.

Das folgende Listing demonstriert diese Modifizierer. Es ist gegenüber dem vorangehenden Beispiel lediglich um einige Elementvariablen und eine abgeleitete Klasse erweitert.

Listing 5.13: Modifizierer für die Verfügbarkeit

```
 1: using System;
 2:
 3: internal class Triangle
 4: {
 5:   protected int m_a, m_b, m_c;
 6:   public Triangle(int a, int b, int c)
 7:   {
 8:     m_a = a;
 9:     m_b = b;
10:     m_c = c;
11:   }
12:
13:   public virtual double Area()
14:   {
15:     // Heronsche Formel
16:     double s = (m_a + m_b + m_c) / 2.0;
17:     double dArea = Math.Sqrt(s*(s-m_a)*(s-m_b)*(s-m_c));
18:     return dArea;
19:   }
20: }
21:
22: internal class Prism:Triangle
23: {
24:   private int m_h;
25:   public Prism(int a, int b, int c, int h):base(a,b,c)
26:   {
27:     m_h = h;
28:   }
29:
30:   public override double Area()
31:   {
32:     double dArea = base.Area() * 2.0;
33:     dArea += m_a*m_h + m_b*m_h + m_c*m_h;
34:     return dArea;
35:   }
36: }
37:
38: class PrismApp
39: {
40:   public static void Main()
41:   {
42:     Prism prism = new Prism(2,5,6,1);
43:     Console.WriteLine(prism.Area());
44:    }
45: }
```

Die Klassen *Triangle* und *Prism* sind hier beide als `internal` deklariert, also ausschließlich innerhalb der aktuellen NGWS-Komponente verfügbar. Bitte beachten Sie, dass der Begriff NGWS-Komponente für die Zusammenfassung von Klassen zu Modulen steht, also nicht im Sinne von COM+-Komponenten gemeint ist. Die Klasse *Triangle* hat drei als `protected` deklarierte Elemente, die im Konstruktor initialisiert und in der Methode *Area* verwendet werden – sowohl in der Basisklassenvariante als auch in der durch die abgeleitete Klasse überschriebenen Version dieser Methode. Die Klasse *Prism* deklariert ein Feld *m_h* als `private`, das ausschließlich in dieser Klasse (also auch nicht in eventuellen weiteren Ableitungen davon) verfügbar ist.

Je mehr gründliche Überlegung Sie beim Design von Klassen dem Thema Verfügbarkeit widmen, desto einfacher können später eventuell notwendige Modifikationen ausfallen. Ein Element, das ohne viel Federlesens von vornherein als `public` deklariert worden ist, dürfte sich nachträglich nur noch schwerlich an veränderte Bedürfnisse anpassen lassen – ganz im Gegensatz zu Teilen einer Klasse, die per se ausschließlich im Verborgenen arbeiten. Das gilt nicht nur für Elemente von Klassen, sondern auch für die Klassen selbst.

5.7 Zusammenfassung

In diesem Kapitel ging es um die verschiedenen Bestandteile von Klassen, die ihrerseits die Vorlage für Objekte darstellen. Die im Konstruktor einer Klasse festgelegten Anweisungen werden beim Anlegen von Objekten einer Klasse als Erstes ausgeführt und übernehmen üblicherweise die Initialisierung wertbehafteter Elemente, die sich ihrerseits später von Methoden der Klasse einsetzen lassen.

Methoden können Werte als Parameter übernehmen, Referenzen an Variablen weiterreichen oder sich auch nur auf das Zurückgeben von Ausgangsparametern beschränken. Sie lassen sich zur Implementation neuer oder veränderter Funktionalität überschreiben; die Alternative ist das Verbergen einer gleichnamigen Methode einer Basisklasse.

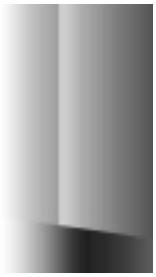
Benannte Attribute einer Klasse lassen sich sowohl über wertbehaftete Elementvariablen als auch über Eigenschaften mit entsprechenden Zugriffsfunktionen implementieren. Zugriffsfunktionen werden über die Schlüsselwörter `get` und `set` gekennzeichnet und ermöglichen unter anderem die Prüfung neu zu setzender bzw. zurückzugebender Werte. Eigenschaften, die nur eine der beiden Zugriffsfunktionen implementieren, lassen sich außerhalb der Klasse ausschließlich lesen bzw. schreiben.

Ein weiteres Feature von C#-Klassen stellen Indizierer dar: Sie erlauben den Zugriff auf Werte über dieselbe Syntax, wie sie bei Arrays zum Einsatz kommt.

Des Weiteren definiert C# einen Mechanismus zur Signalisierung von und Reaktion auf Ereignisse. Klassen und ihre Objekte können bei Bedarf eine beliebige Zahl von Clients über Zustandsänderungen benachrichtigen.

Die Lebensspanne eines Objekts endet mit dem Aufruf seines Destruktors durch den Garbage Collector. Da sich nicht exakt bestimmen lässt, zu welchem Zeitpunkt der Garbage Collector zum Zuge kommt, sollten Sie gegebenenfalls eine Methode zur Freigabe von Ressourcen definieren, die ein Programm dann direkt aufruft, wenn es ein Objekt nicht länger braucht.

Kapitel 6



Kontrollstrukturen

Eine spezielle Kategorie von Anweisungen werden Sie in jeder Programmiersprache finden: Kontrollstrukturen. Dieses Kapitel stellt Ihnen die Kontrollstrukturen der Sprache C# vor, die sich wie üblich in zwei Gruppen unterteilen lassen:

- Anweisungen für die bedingte Ausführung von Code
- Anweisungen für Schleifen

Wenn Ihnen C oder C++ geläufig ist, wird Ihnen vieles in diesem Kapitel recht bekannt vorkommen. Seien Sie in diesem Fall aber dennoch wachsam – es gibt einige Unterschiede.

6.1 Bedingte Ausführung

Kontrollstrukturen für die bedingte Ausführung sind Anweisungen, die anhand eines Wertes entscheiden, welcher Codezweig zur Ausführung kommen soll. C# kennt zwei Anweisungen dieser Art: `if` und `switch`.

6.1.1 if

Die allgemeinere der beiden Anweisungen ist die `if`-Anweisung. Sie führt die Anweisung *`bedingteAnweisung`* nur dann aus, wenn sich als Wahrheitswert für die logische Bedingung *`booleschерAusdruck`* der Wert `true` ergibt:

```
if (booleschерAusdruck) bedingteAnweisung
```

Natürlich lässt sich in C# auch ein `else`-Zweig angeben, der dann zur Ausführung kommt, wenn die Bedingung den Wahrheitswert `false` liefert:

```
if (booleschерAusdruck) bedingteAnweisung else bedingteAnweisung
```

Um beispielsweise sicherzustellen, dass die Zeichenfolge `strTest` nicht leer ist, schreiben Sie:

```
if (0 != strTest.Length)
{
}
```

Der für die Bedingung verwendete Vergleichsoperator `!=` hat die Bedeutung: »ist ungleich«. Als C/C++-ProgrammiererIn werden Sie wahrscheinlich Code wie diesen gewohnt sein:

```
if (strTest.Length)
{
}
```

In C# bedenkt der Compiler solche Formulierungen mit der Fehlermeldung:

```
error CS0029: Cannot implicitly convert type 'int' to 'bool'
```

Das liegt daran, dass die `if`-Anweisung für die Bedingung den Datentyp `bool` erwartet und die `Length`-Eigenschaft des `string`-Objekts den Typ `int` trägt – eine implizite Typumwandlung von `int` nach `bool` ist in C#, wie bereits erwähnt, nicht definiert.

Vielleicht werden Sie es als Nachteil empfinden, eine liebgewordene Programmiergewohnheit aufgeben zu müssen. Der Vorteil ist aber, dass Sie so nie wieder der berüchtigten »Zuweisung in der `if`-Bedingung« auf den Leim gehen werden:

```
if (nMyValue = 5) ...
```

Damit eine Vergleichsoperation daraus wird, sollte der Code eigentlich so lauten:

```
if (nMyValue == 5) ...
```

Und nur diese Formulierung lässt der C#-Compiler auch durchgehen. Die weiteren, den Sprachen C und C++ entlehnten, Vergleichsoperatoren von C# sind:

- `==` – liefert `true`, wenn beide Operanden übereinstimmen
- `!=` – liefert `true`, wenn beide Operanden nicht übereinstimmen
- `<`, `<=`, `>`, `>=` – liefert `true`, wenn die Operanden die entsprechende Relation erfüllen (»kleiner als«, »kleiner gleich«, »größer als« und »größer gleich«).

Erwartungsgemäß können Sie mit diesen Operatoren nicht jeden Datentyp mit jedem Datentyp vergleichen. Da die einzelnen Datentypen jeweils ihre eigenen überladenen Varianten für diese Operatoren definieren (oder auch nicht), ist man für typübergreifende Vergleiche – etwa eines `int`-Werts mit einem `double`-Wert – auf die implizite Typumwandlung durch den Compiler angewiesen. Wenn das nicht greift, bleibt noch das Mittel der expliziten Typumwandlung – doch dann sollte man sich schon genau darüber im Klaren sein, was passiert, wenn man »Äpfel mit als Äpfel getarnten Birnen« vergleicht.

Das folgende Listing zeigt verschiedene Szenarien für den Einsatz der `if`-Anweisung und demonstriert auch recht schön den Umgang mit dem Datentyp `string`. Der Code wertet den ersten Kommandozeilenparameter der Anwendung daraufhin aus, ob er mit einem großen oder kleinen Buchstaben oder einer Ziffer beginnt.

Listing 6.1: Ein einzelnes Zeichen einer Zeichenfolge untersuchen

```
1: using System;
2:
3: class NestedIfApp
4: {
```

```
 5:   public static int Main(string[] args)
 6:   {
 7:     if (args.Length != 1)
 8:     {
 9:       Console.WriteLine("Aufruf: Ein Kommandozeilenargument
          angeben");
10:       return 1;  // Fehlernummer
11:     }
12:
13:     char chLetter = args[0][0];
14:
15:     if (chLetter >= 'A')
16:       if (chLetter <= 'Z')
17:       {
18:         Console.WriteLine("{0} ist Großbuchstabe", chLetter);
19:         return 0;
20:       }
21:
22:     chLetter = Char.FromString(args[0]);
23:     if (chLetter >= 'a' && chLetter <= 'z')
24:       Console.WriteLine("{0} ist Kleinbuchstabe", chLetter);
25:
26:     if (Char.IsDigit((chLetter = args[0][0])))
27:       Console.WriteLine("{0} ist Ziffer", chLetter);
28:
29:     return 0;
30:   }
31: }
```

Der erste, in Zeile 7 beginnende `if`-Block testet, ob das Programm mit weniger oder mehr als einem Kommandozeilenparameter aufgerufen wurde und beendet es gegebenenfalls nach Ausgabe einer Meldung mit dem Fehlercode 1. Andernfalls analysiert das Programm das erste Zeichen des ersten (und einzigen) Kommandozeilenparameters. Wenn es darum geht, das erste Zeichen aus einer Zeichenfolge herauszulösen, haben Sie mehrere Möglichkeiten: den Einsatz des `char`-Indizierers unter Angabe des Indexwertes 0 (Zeile 13) oder den Aufruf der statischen Methode `FromString` aus der Klasse `Char`, die grundsätzlich das erste Zeichen einer übergebenen Zeichenfolge zurückliefert (Zeile 22).

Die Zeilen 15 bis 20 enthalten zwei verschachtelte `if`-Blöcke, die das Zeichen auf Großschreibung untersuchen und gegebenenfalls eine entsprechende Meldung ausgeben. Der Test auf Kleinschreibung (Zeilen 23, 24) erfolgt unter Verwendung des logischen UND-Operators `&&`, und den Test, ob das Zeichen eine Ziffer ist (Zeilen 26, 27), erledigt die statische Methode `IsDigit` der Klasse `Char`.

Das Gegenstück zu `&&` ist der logische ODER-Operator `||`. Beachten Sie, dass der C#-Compiler für diese beiden Operatoren den zweiten Operanden gar nicht erst

auswertet, wenn sich der Wert des Ausdrucks bereits aus dem ersten Operanden ergibt. Falls also der erste Operand in einer UND-Operation den Wert `false` hat, ist das Ergebnis des Ausdrucks in jedem Fall `false`, und falls er in einer ODER-Operation den Wert `true` hat, ist das Ergebnis des Ausdrucks in jedem Fall `true`.

Vom Prinzip her können Sie also Laufzeit einsparen, indem Sie den Operanden zuerst notieren, der die Operation voraussichtlich bereits allein entscheiden kann. Leider hat diese Form der Optimierung auch ihren Preis: Sie können sich nicht darauf verlassen, dass der zweite Operand ausgewertet wird – was in der Praxis zu schwer auffindbaren Fehlern führen kann, wenn diese Auswertung den Wert einer Variablen ändert. Das folgende Beispiel stellt die Situation übertrieben augenfällig dar:

```
if (1 == 1 || (5 == (strLength=str.Length)))
{
  Console.WriteLine(strLength);
}
```

Da der erste Operand in jedem Fall `true` ist, kommt die im zweiten Operanden versteckte Zuweisung nie zur Ausführung. Die Praxis hält subtilere Fallen bereit – etwa, wenn als zweiter Operand eine Methode zum Aufruf kommt, die den Wert eines Referenzparameters oder globalen Variable verändert.

```
if (Test(lNumber) || TestAndAdd1(lNumber))
{
}
```

6.1.2 switch

Im Gegensatz zur `if`-Anweisung entscheidet bei der `switch`-Anweisung ein Steuerausdruck darüber, welcher von mehreren möglichen Anweisungszweigen zur Ausführung kommt. Dazu sind den einzelnen Anweisungszweigen (je unterschiedliche) Fallbeschreibungen in Form konstanter Werte zugeordnet. Die allgemeine Syntax der `switch`-Anweisung lautet:

```
switch (Steuerausdruck)
{
  case konstanterAusdruck:
    Anweisungszweig
  default:
    Anweisungszweig
}
```

Als Datentypen für *Steuerausdruck* sind `sbyte`, `byte`, `short`, `ushort`, `uint`, `long`, `ulong`, `char`, `string` sowie alle Aufzählungstypen erlaubt. Weiterhin kommen auch Typen in Frage, für die der Compiler eine implizite Typumwandlung in einen dieser Typen vornehmen kann.

Für die Abarbeitung der `switch`-Anweisung gilt folgende Reihenfolge:

1. Als Erstes wird der Ausdruck *`Steuerausdruck`* ausgewertet.
2. Stimmt der Wert von *`Steuerausdruck`* mit *`konstanterAusdruck`* eines Zweigs überein, kommt es zur Ausführung der in dem Zweig enthaltenen Anweisungen.
3. Stimmt der Wert des Steuerausdrucks mit keiner Fallbeschreibung überein, kommt es zur Ausführung der im `default`-Zweig enthaltenen Anweisungen – sofern ein solcher definiert ist; andernfalls wird der `switch`-Block übersprungen, ohne dass es zur Ausführung irgendeines Zweiges kommt.

Vor der Diskussion weiterer Details der `switch`-Kontrollstruktur sollten Sie sich das Programm in Listing 6.2 ansehen. Es arbeitet mit einer `switch`-Anweisung, die sechs Fälle unterscheidet, um für einen gegebenen Monat die Anzahl der Tage (ohne Berücksichtigung von Schaltjahren) zu ermitteln.

Listing 6.2: Eine `switch`-Anweisung unterscheidet, welcher Monat wie viele Tage hat

```
 1: using System;
 2:
 3: class FallThrough
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     if (args.Length != 1) return;
 8:
 9:     int nMonth = Int32.Parse(args[0]);
10:     if (nMonth < 1 || nMonth > 12) return;
11:     int nDays = 0;
12:
13:     switch (nMonth)
14:     {
15:       case 2: nDays = 28; break;
16:       case 4:
17:       case 6:
18:       case 9:
19:       case 11: nDays = 30; break;
20:       default: nDays = 31;
21:     }
22:     Console.WriteLine("Der Monat hat {0} Tage.",nDays);
23:   }
24: }
```

Der `switch`-Block umfasst die Zeilen 13 bis 21. C/C++-ProgrammiererInnen wird der Code recht vertraut vorkommen, nicht zuletzt deshalb, weil er `break`-Anweisungen enthält. Dennoch, es gibt einen Unterschied, der einem in C# Fehler ver-

meiden hilft. Wann immer ein Zweig Anweisungen enthält, *muss* er durch eine `break`-Anweisung (oder eine `goto`-Anweisung zu einem anderen Zweig) abgeschlossen sein, ansonsten meldet der Compiler einen Fehler. Das Durchlaufen mehrerer Zweige aufgrund einer fehlenden `break`-Anweisung wie in C/C++ ist also nicht möglich. Der folgende Code würde von einem C/C++-Compiler akzeptiert – bei C# kommt dagegen eine Fehlermeldung heraus:

```
nVar = 1
switch (nVar)
{
  case 1:
    DoSomething();
  case 2:
    DoMore();
}
```

Falls `nVar` bei Eintritt in die `switch`-Anweisung den Wert 1 hat, werden in C/C++ also beide Zweige ausgeführt. Das sprachliche Potenzial dieser Konzeption ist zwar mächtig, seine Implizitheit lädt aber auch dazu ein, schwer aufzufindende Fehler zu produzieren. Bei C# hat man daher darauf verzichtet. Was aber, wenn man für den einen oder anderen Fall die gleichen Zweige hintereinander ausführen möchte? Listing 6.3 zeigt, wie das in C# gelöst ist:

Listing 6.3: Einsatz von `goto` *`label`* und `goto` *`default`* in einer `switch`-Anweisung

```
 1: using System;
 2:
 3: class SwitchApp
 4: {
 5:   public static void Main()
 6:   {
 7:     Random objRandom = new Random();
 8:     double dRndNumber = objRandom.NextDouble();
 9:     int nRndNumber = (int)(dRndNumber * 10.0);
10:
11:     switch (nRndNumber)
12:     {
13:       case 1:
14:         // nichts tun
15:         break;
16:       case 2:
17:         goto case 3;
18:       case 3:
19:         Console.WriteLine("Fall 2 oder 3");
20:         break;
21:       case 4:
22:         goto default;
23:         // Weiterer Code in diesem Zweig ist nicht erreichbar
```

```
24:         // und wird vom Compiler mit einer Warnung bedacht.
25:       default:
26:         Console.WriteLine("Zufallszahl lautet: {0}", nRndNumber);
27:     }
28:   }
29: }
```

In diesem Beispiel generiert der in der Klasse `Random` verkapselte Zufallsgenerator den Wert des Steuerausdrucks (Zeilen 7 bis 9). Der `switch`-Block enthält zwei Sprunganweisungen, die Zweige miteinander verketten:

- `goto case` *`AndererFall`* – Sprung in den Zweig für einen anderen Fall
- `goto default` – Sprung in den Zweig `default`

Mit diesen beiden Sprunganweisungen ist das Ausdruckspotenzial der `switch`-Kontrollstruktur in C# sogar noch reichhaltiger als in C/C++ und zugleich sinkt das Fehlerpotenzial, da jeder Sprung explizit formuliert werden muss. Warum »reichhaltiger»? Nun, erstens spielt die Anordnung der Zweige keine Rolle mehr, so dass man beispielsweise den `default`-Zweig an den Anfang setzen kann, um die Lesbarkeit des Codes zu verbessern; zweitens lassen sich mit C# auch Verkettungen formulieren, die in C/C++ allein durch die Reihenfolge nicht mehr ausgedrückt werden können – so etwa, wenn Fall 1 und Fall 2 jeweils mit dem Fall `default` verkettet sein sollen, wie das folgende Beispiel zeigt:

```
switch (nSomething)
{
  default:
     DoDefault()
  case 1:
     DoCase1();
     goto default;
  case 2:
     DoCase2();
     goto default;
}
```

Vielleicht ist es Ihnen bereits aufgefallen: In C# kann der `switch`-Steuerausdruck auch den Typ `string` haben. Auch wenn das Visual-Basic-Programmierern nicht mehr als ein Achselzucken entlocken wird, stellt es gegenüber C/C++ eine wirklich sensationelle Ausweitung der Ausdruckskraft dar.

Die folgende `switch`-Anweisung führt eine Fallunterscheidung auf der Basis von Zeichenfolgen durch:

```
string strTest = "Chris";
switch (strTest)
{
```

```
  case "Chris":
    Console.WriteLine("Hallo Chris!");
    break;
}
```

6.2 Schleifen

Wenn Sie in C# eine einzelne Anweisung oder einen Anweisungsblock mehrfach ausführen wollen, stehen Ihnen vier unterschiedliche Schleifenarten zur Auswahl. Die entsprechenden Anweisungen sind:

- `for`
- `foreach`
- `while`
- `do`

6.2.1 for

Die `for`-Anweisung ist speziell dann nützlich, wenn bereits vor Eintritt in die Schleife bekannt ist, wie viele Durchläufe es geben soll. Ihre Syntax ist wieder einmal C/C++ entlehnt, so dass sie von der Ausdruckskraft her den anderen Schleifen um nichts nachsteht – im Gegenteil, sie bietet sogar die meiste Flexibilität. Die Schleife wird so lange durchlaufen, wie die (optionale) Abbruchbedingung erfüllt ist oder ein Abbruch mit `break` erfolgt. Zudem besteht natürlich die (gleichfalls optionale) Möglichkeit, Laufvariablen zu initialisieren und nach jedem Schleifendurchlauf zu aktualisieren. Die allgemeine Form lautet:

```
for (Initialisierung; Abbruchbedingung; Aktualisierung) Anweisung
```

Beachten Sie, dass die Bestandteile *`Initialisierung`*, *`Abbruchbedingung`* und *`Aktualisierung`* jeweils optional sind. Vom Prinzip her können Sie mit der `for`-Anweisung auch eine Endlosschleife formulieren, die durch eine bedingte `break`- oder `goto`-Anweisung abgebrochen wird:

```
for (;;)
{
  if(bCondition) break;  // bedingter Abbruch der Endlosschleife
}
```

Ein weiterer wichtiger Punkt ist, dass jeder der drei Bestandteile mehrere durch Komma getrennte Anweisungen umfassen kann. Sie könnten also beispielsweise im Initialisierungsteil einer `for`-Anweisung zwei Variablen initialisieren und im Aktualisierungsteil vier Variablen aktualisieren.

Als C/C++-Programmierer müssen Sie nur auf eines achten: Der Ergebnistyp von *Abbruchbedingung* muss `bool` sein – wie bei der `if`-Anweisung also.

Listing 6.4 zeigt ein Programm, das die Fakultät eines als Kommandozeilenargument übergebenen Zahlenwertes über eine typische `for`-Schleife iterativ berechnet (was bekanntlich schneller geht als rekursiv):

Listing 6.4: Fakultät in einer *for*-Schleife berechnen

```
 1: using System;
 2:
 3: class Factorial
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     long nFactorial = 1;
 8:     long nComputeTo = Int64.Parse(args[0]);
 9:
10:     long nCurDig = 1;
11:     for (nCurDig=1; nCurDig <= nComputeTo; nCurDig++)
12:       nFactorial *= nCurDig;
13:
14:     Console.WriteLine("{0}! ist {1}",nComputeTo, nFactorial);
15:   }
16: }
```

Der Code in diesem Beispiel sieht recht schwerfällig aus, das ist er auch. Dafür bietet er aber einen guten Ausgangspunkt für Verbesserungen. So könnte man die Deklaration und Initialisierung der Laufvariablen `nCurDig` in den Initialisierungsteil der `for`-Anweisung verlegen:

```
for (long nCurDig=1; nCurDig <= nComputeTo; nCurDig++) nFactorial *=
nCurDig;
```

Oder man könnte den Initialisierungsteil der `for`-Anweisung gänzlich weglassen, da die Laufvariable im Zuge der Deklaration in Zeile 10 bereits richtig initialisiert wird. (Sie erinnern sich: C# fordert die Initialisierung aller Variablen.)

```
for (; nCurDig <= nComputeTo; nCurDig++) nFactorial *= nCurDig;
```

Außerdem muss die Aktualisierung der Laufvariablen nicht unbedingt im Aktualisierungsteil der `for`-Anweisung geschehen, ein Postinkrement ++ bei der (letzten) Nennung im Schleifenkörper tut es auch:

```
for ( ; nCurDig <= nComputeTo; ) nFactorial *= nCurDig++;
```

Wenn Sie nun auch noch den Bedingungsteil der `for`-Anweisung loswerden wollen, müssen Sie nur eine bedingte `break`-Anweisung in den Schleifenkörper setzen:

```
for ( ; ; )
{
  if (nCurDig > nComputeTo) break;
  nFactorial *= nCurDig++;
}
```

Neben der `break`-Anweisung, die den Abbruch der `for`-Schleife bewirkt, gibt es noch die `continue`-Anweisung. Sie überspringt den Rest des Schleifenkörpers, verhindert aber nicht den Wiedereintritt.

```
for (; nCurDig <= nComputeTo;)
{
  if (5 == nCurDig) continue; // überspringt den restlichen Code
  nFactorial *= nCurDig++;
}
```

6.2.2 foreach

In Visual Basic gibt es sie schon lange: die Anweisung `For Each` zum Durchlaufen von Auflistungen. Mit der Anweisung `foreach` steht C# demnach also in der Tradition von Visual Basic. Die allgemeine Form lautet:

```
foreach (ElemTyp Iterator in AuflistungsAusdruck) Anweisung
```

Für die Laufvariable *`Iterator`* wird der Elementtyp *`ElemTyp`* der Auflistung *`AuflistungsAusdruck`* deklariert. Sie ermöglicht bei jedem Schleifendurchlauf den Zugriff auf das jeweils nächste Element der Auflistung. Beachten Sie jedoch, dass sich die Laufvariable im Schleifenkörper nicht als Linkswert in einer Wertzuweisung und damit auch nicht als Referenzparameter (`ref`) oder Ausgabeparameter (`out`) einsetzen lässt.

Es versteht sich, dass `foreach` nur für Klassen eingesetzt werden kann, die als Auflistung definiert sind. Wie aber sieht man einer Klasse die Auflistung an? Nun, entscheidend ist, dass die Klasse eine Methode namens `GetEnumerator()` definiert und für den von ihr gelieferten Ergebnistyp (`struct`-Typ, Klasse oder Schnittstelle) die Methode `MoveNext()` sowie die Eigenschaft `Current` öffentlich verfügbar ist. Weitere Details finden Sie in der Sprachreferenz, in der dieses Thema ausführlich behandelt wird.

Das folgende Listing demonstriert den Einsatz der `foreach`-Anweisung anhand einer Klasse, die all diese Anforderungen erfüllt:

Listing 6.5: Ausgabe aller Umgebungsvariablen des Programms

```
1: using System;
2: using System.Collections;
3:
4: class EnvironmentDumpApp
```

```
 5: {
 6:   public static void Main()
 7:   {
 8:     IDictionary envvars = Environment.GetEnvironmentVariables();
 9:     Console.WriteLine("Es gibt {0} Umgebungsvariablen",
           envvars.Keys.Count);
10:     foreach (string strKey in envvars.Keys)
11:     {
12:       Console.WriteLine("{0} = {1}", strKey, envvars[strKey].
          ToString());
13:     }
14:   }
15: }
```

Der Aufruf von `GetEnvironmentVariables` in Zeile 8 liefert eine Referenz auf die Schnittstelle `IDictionary`, für die viele Klassen des NGWS-Subsystems eine Implementation bereitstellen. `IDictionary` sieht die Definition zweier Auflistungen vor: `Keys` und `Values`. Eine davon, `Keys`, durchläuft der Code im Rahmen einer `foreach`-Anweisung, um die Namen und die Werte der Umgebungsvariablen der Reihe nach auszugeben (Zeile 12).

Beim Einsatz von `foreach` müssen Sie auf eines besonders achten: dass Sie den richtigen Typ für die Laufvariable deklarieren. Ein falscher Typ wird nämlich nicht in jedem Fall vom Compiler entdeckt, wohl aber zur Laufzeit, wo er einen Ausnahmefehler verursacht.

6.2.3 while

Wenn es darum geht, eine Anweisung schlicht in Abhängigkeit von einer Bedingung wiederholt auszuführen, ist die `while`-Anweisung das Mittel der Wahl. Die allgemeine Form dieser Anweisung lautet:

```
while (Abbruchbedingung) Anweisung
```

Die Abbruchbedingung, ein Ausdruck, für den C# den Ergebnistyp `bool` fordert, wird vor jedem (Wieder-)Eintritt in den Schleifenkörper ausgewertet. Sie bestimmt damit implizit, wie oft der Schleifenkörper *`Anweisung`* durchlaufen wird. Wie `for` definiert die `while`-Anweisung eine *abweisende* Schleife, deren Schleifenkörper nicht unbedingt durchlaufen werden muss. Und auch `break` sowie `continue` sind (mit gleicher Wirkung) zulässig.

Um den Gebrauch der `while`-Anweisung zu demonstrieren, zeigt Listing 6.6 die Ausgabe einer C#-Quelldatei über die Konsole:

Listing 6.6: Ausgabe einer Datei über die Konsole

```
1: using System;
2: using System.IO;
```

```
 3:
 4: class WhileDemoApp
 5: {
 6:   public static void Main()
 7:   {
 8:     StreamReader sr = File.OpenText ("whilesample.cs");
 9:     String strLine = null;
10:
11:     while (null != (strLine = sr.ReadLine()))
12:     {
13:         Console.WriteLine(strLine);
14:     }
15:
16:     sr.Close();
17:   }
18: }
```

Der Code öffnet die Datei *whilesample.cs* im Textmodus. Eine `while`-Anweisung liest den Inhalt der Datei Zeile für Zeile und gibt ihn an der Konsole aus, bis die Methode `ReadLine` den Wert `null` anstelle einer Zeichenfolge zurückliefert. Beachten Sie, dass der Bedingungsteil der Schleife eine Wertzuweisung enthält, deren Ausführung in einem logischen Ausdruck mit `||` oder `&&` nur dann gewährleistet wäre, wenn sie den ersten (linken) Operanden bildet.

6.2.4 do

Die vierte und letzte Schleifenart von C# liegt in Form der `do`-Anweisung vor. Als nicht abweisendes Gegenstück der `while`-Anweisung wertet die `do`-Anweisung die Abbruchbedingung erst *nach* Durchlaufen des Schleifenkörpers aus. Die allgemeine Form lautet:

```
do
{
  Anweisung
}
while (Abbruchbedingung);
```

Der Schleifenkörper *`Anweisung`* einer `do`-Anweisung wird in jedem Fall einmal durchlaufen. Weitere Durchläufe steuert dann der Wahrheitswert des Booleschen Ausdrucks *`Abbruchbedingung`*. Wie bei den anderen Schleifenarten sind auch bei der `do`-Schleife die Anweisungen `break` und `continue` (mit gleicher Wirkung) zulässig.

Listing 6.7 zeigt ein Beispiel für eine `do`-Anweisung. Das Programm fordert den Benutzer in einer `do`-Schleife wiederholt auf, eine Zahl einzugeben. Sobald dieser die Eingabe einer weiteren Zahl ablehnt, bricht die Schleife ab, und das Programm gibt das arithmetische Mittel aller eingegebenen Zahlen aus:

Listing 6.7: Arithmetisches Mittel in einer do-Schleife berechnen

```
 1: using System;
 2:
 3: class ComputeAverageApp
 4: {
 5:   public static void Main()
 6:   {
 7:     ComputeAverageApp theApp = new ComputeAverageApp();
 8:     theApp.Run();
 9:   }
10:
11:   public void Run()
12:   {
13:     double dValue = 0;
14:     double dSum = 0;
15:     int nNoOfValues = 0;
16:     char chContinue = 'j';
17:     string strInput;
18:
19:     do
20:     {
21:       Console.Write("Bitte eine Zahl eingeben: ");
22:       strInput = Console.ReadLine();
23:       dValue = Double.Parse(strInput);
24:       dSum += dValue;
25:       nNoOfValues++;
26:       Console.Write("Noch eine Zahl (j/n)? ");
27:
28:       strInput = Console.ReadLine();
29:       chContinue = Char.FromString(strInput);
30:     }
31:     while ('j' == chContinue);
32:
33:     Console.WriteLine("Arithmetisches Mittel: {0}", dSum /
       nNoOfValues);
34:   }
35: }
```

In Abweichung zu den bisherigen Beispielen legt die Methode `Main` hier ein Objekt der Klasse `ComputeAverageApp` an und ruft dann dessen `Run`-Methode auf, in der die eigentliche Logik für die Berechnung des arithmetischen Mittels enthalten ist.

Die `do`-Anweisung zieht sich über die Zeilen 19 bis 31. Die Abbruchbedingung prüft, ob der Benutzer die Frage nach Eingabe einer weitere Zahl mit »j« beantwortet hat. Jedes andere Zeichen führt zum Abbruch der Schleife und zur Ausgabe des arithmetischen Mittels.

Das Beispiel macht den einzigen Unterschied zwischen `do` und `while` deutlich: Die `while`-Anweisung prüft die Abbruchbedingung bereits vor dem ersten Durchlaufen des Schleifenkörpers, die `do`-Anweisung erst danach.

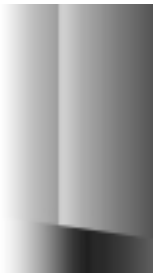
6.3 Zusammenfassung

Dieses Kapitel hat Ihnen die in C# verfügbaren Kontrollstrukturen für die bedingte und die wiederholte Ausführung von Anweisungen vorgestellt. Am häufigsten werden Sie in Ihren Programmen wahrscheinlich die `if`-Anweisung einsetzen. Da Sie der Compiler dazu zwingt, Bedingungen im Datentyp `bool` zu formulieren, ist eine Verwechslung von `=` und `==` ausgeschlossen. Sie müssen nur darauf aufpassen, dass die Operanden Boolescher Ausdrücke keine wertverändernden Berechnungen anstellen, da in C# die Auswertung des zweiten Operanden in Booleschen Ausdrücken mit `&&` und `||` nicht garantiert ist.

Die aus der C-Welt stammende `switch`-Anweisung hat sich in C# ebenfalls zu ihrem Vorteil verändert. Sie müssen es nun explizit formulieren, wenn für einen Fall mehrere Zweige durchlaufen werden sollen. Die Reihenfolge der Zweige spielt keine Rolle mehr, und die Ausdruckskraft hat sich verbessert. Darüber hinaus lässt sich die Fallunterscheidung in C# auch mit dem Datentyp `string` formulieren – was in C/C++ nicht möglich ist.

In der zweiten Hälfte des Kapitels ging es um die vier Schleifenanweisungen `for`, `foreach`, `while` und `do`. Jede der Anweisungen dient einem anderen Zweck: `for` ermöglicht eine feste Anzahl von Schleifendurchläufen; `foreach` zählt die Elemente von Auflistungen auf; `while` implementiert eine abweisende Schleife mit implizitem Kriterium; `do` kontrolliert eine nicht abweisende Schleife mit implizitem Kriterium.

Kapitel 7



Ausnahmen behandeln

Ein großer Vorteil des NGWS-Laufzeitsystems ist, dass es die Ausnahmebehandlung für verschiedene Sprachen standardisiert. Auf diese Weise kann eine von C# signalisierte Ausnahme jederzeit in einem Visual-Basic-Client behandelt werden oder umgekehrt. `HRESULT`-Werte und Schnittstellen à la `ISupportErrorInfo` werden in absehbarer Zeit wohl ausgedient haben.

Obwohl die sprachübergreifende Ausnahmebehandlung eine großartige Sache ist, konzentriert sich dieses Kapitel ausschließlich auf die Ausnahmebehandlung innerhalb von C#. Man dreht ein wenig am Verhalten des Compilers, was die Behandlung von Überläufen angeht, und schon geht der Spaß los: Überläufe führen zu Ausnahmen, und Ausnahmen lassen sich im Code auffangen und behandeln. Das ist aber nur die eine Seite der Medaille; die andere sieht so aus, dass man Ausnahmen selbst signalisiert und sogar implementiert.

7.1 Die Anweisungen *checked und unchecked*

Im Verlauf einer Berechnung kann es schnell einmal passieren, dass das errechnete Ergebnis nicht mehr im Wertebereich des zugrunde gelegten Datentyps liegt. Diese Situation wird gemeinhin als *Überlauf* bezeichnet, und je nach verwendeter Programmiersprache erhält der Benutzer davon in irgendeiner Form Kenntnis – oder eben auch nicht. (Kommt Ihnen dies von C++ her etwa bekannt vor?)

Wie also verfährt C# mit Überläufen? Um das diesbezügliche Standardverhalten der Sprache zu erkunden, betrachten Sie noch einmal das bereits in Kapitel 6 vorgestellte Programm, das die Fakultät einer Zahl berechnet. Zu Ihrer Bequemlichkeit ist es hier noch einmal abgedruckt:

Listing 7.1: Programm zur Berechnung der Fakultät einer Zahl

```
 1: using System;
 2:
 3: class Factorial
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     long nFactorial = 1;
 8:     long nComputeTo = Int64.Parse(args[0]);
 9:
10:     long nCurDig = 1;
11:     for (nCurDig=1; nCurDig <= nComputeTo; nCurDig++)
12:       nFactorial *= nCurDig;
13:
14:     Console.WriteLine("{0}! ist {1}",nComputeTo, nFactorial);
15:   }
16: }
```

Wenn Sie dem Programm beim Aufruf nun ein etwas zu »großzügig« bemessenes Argument mitgeben, etwa:

```
factorial 2000
```

gibt es als Ergebnis den Wert 0 aus – ohne weiteren Kommentar. Mit anderen Worten, Sie können davon ausgehen, dass C# Überlaufsituationen mit Diskretion behandelt und keine expliziten Warnungen von sich gibt.

Dieses Verhalten lässt sich ändern, indem Sie einen Compilerschalter setzen, der die Überlaufkontrolle für das gesamte Programm aktiviert, oder indem Sie die Überlaufkontrolle auf der Ebene einzelner Anweisungen aktivieren. Die zwei folgenden Abschnitte diskutieren diese beiden Ansätze.

7.1.1 Überlaufkontrolle per Compiler-Schalter aktivieren

Sofern Sie für die gesamte Anwendung eine Überlaufkontrolle wünschen, reicht es, den Compilerschalter `/checked+` zu setzen, der standardmäßig deaktiviert ist. Der Aufruf für die Übersetzung des Beispielprogramms lautet dann:

```
csc factorial.cs /checked+
```

Wenn Sie das auf diese Weise kompilierte Programm nun mit dem Kommandozeilenargument 2000 aufrufen, meldet Ihnen das NGWS-Laufzeitsystem einen unbehandelten Laufzeitfehler (vgl. Abbildung 7.1).

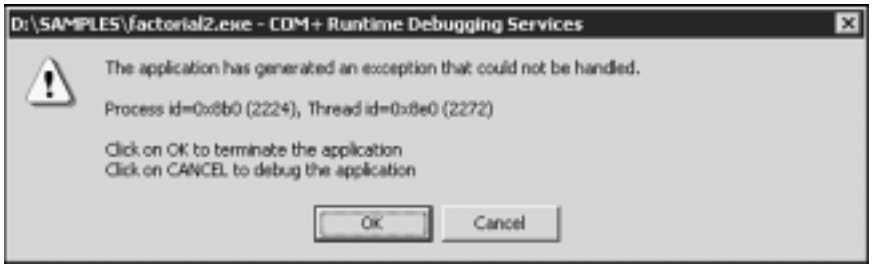


Bild 7.1: Bei aktivierter Überlaufkontrolle scheitert die Berechnung der Fakultät für die Zahl 2000

Nach Bestätigung des Dialogfelds mit OK erhalten Sie über die Konsole zusätzlich die Fehlermeldung:

```
Exception occurred: System.OverflowException
   at Factorial.Main(System.String[])
```

Dem lässt sich entnehmen, dass ein Überlauf die Ausnahme `System.OverflowException` auslöst. Über die Behandlung solcher Ausnahmen gleich mehr im Anschluss an den folgenden Abschnitt.

7.1.2 Überlaufkontrolle vom Code aus steuern

Falls Ihnen eine Überlaufkontrolle für den gesamten Code zuviel des Guten ist, werden Sie es wahrscheinlich begrüßen, dass C# in Form der Anweisung `checked` die Aktivierung der Überlaufkontrolle auch auf der Ebene einzelner Anweisungen bzw. Codeblocks unterstützt. Listing 7.2 zeigt, wie das geht:

Listing 7.2: Berechnung der Fakultät mit Überlaufkontrolle für die gefährdete Anweisung

```
 1: using System;
 2:
 3: class Factorial
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     long nFactorial = 1;
 8:     long nComputeTo = Int64.Parse(args[0]);
 9:
10:     long nCurDig = 1;
11:     for (nCurDig=1; nCurDig <= nComputeTo; nCurDig++)
12:       checked { nFactorial *= nCurDig; }
13:
14:     Console.WriteLine("{0}! ist {1}",nComputeTo, nFactorial);
15:   }
16: }
```

Selbst wenn Sie das Programm nun unter Angabe von `/checked-` kompilieren, ist die Überlaufkontrolle für Zeile 12 aktiviert und generiert die bekannte Fehlermeldung.

Umgekehrt können Sie bei aktivem Compilerschalter `checked+` bestimmte Codeblocks von der Überlaufkontrolle ausnehmen, indem Sie die `unchecked`-Anweisung benutzen. Der Gebrauch ist analog:

```
unchecked
{
  nFactorial *= nCurDig;
}
```

7.2 Anweisungen für die Ausnahmebehandlung

Nachdem Sie nun wissen, wie man eine Ausnahme auslöst – und Sie werden ungewollt sicher noch mehr Wege finden, dies zu tun – stellt sich die Frage: Wie geht man mit ihr um? Falls Sie schon einmal ernsthaft eine Win32-Anwendung

mit C++ programmiert haben, wird Ihnen die strukturierte Ausnahmebehandlung (SEH = structured exception handling) sicher geläufig sein. In C# sieht die Sache vom Prinzip her ähnlich aus.

Die folgenden drei Abschnitte stellen Ihnen die C#-Anweisungen für die strukturierte Ausnahmebehandlung vor:

- Ausnahmen mit `try...catch` behandeln
- Aufräumcode mit `try...finally` ausführen
- Ausnahmen mit `try...catch...finally` voll im Griff

7.2.1 Ausnahmen mit *try...catch* behandeln

Wenn Ausnahmen ins Spiel kommen, sollten Sie sich darum kümmern, dem Benutzer entsprechende Meldungen des Laufzeitsystems zu ersparen und den Abbruch des Programms respektive die Aktivierung des Debuggers zu vermeiden. Mit anderen Worten: Es ist erforderlich, die Ausnahmen abzufangen und in irgendeiner Form sinnvoll zu behandeln, damit die Ausführung fortgesetzt werden kann.

Die Anweisungen, die Ihnen dies ermöglichen, sind `try` und `catch`. Ein `try`-Block enthält die (regulären) Anweisungen des Programms, die möglicherweise zu einer Ausnahme führen; ein `catch`-Block enthält zusätzliche Anweisungen für die Fehlerbehandlung im Falle einer Ausnahme. Listing 7.3 zeigt eine Implementation des diskutierten Programms, die die Ausnahme `OverflowException` mittels `try` und `catch` behandelt:

Listing 7.3: Behandlung der Ausnahme *`OverflowException`*

```
 1: using System;
 2:
 3: class Factorial
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     long nFactorial = 1, nCurDig=1;
 8:     long nComputeTo = Int64.Parse(args[0]);
 9:
10:     try
11:     {
12:       checked
13:       {
14:         for (; nCurDig <= nComputeTo; nCurDig++)
15:           nFactorial *= nCurDig;
16:       }
17:     }
18:     catch (OverflowException oe)
19:     {
```

```
20:       Console.WriteLine("Überlauf bei der Berechnung von {0}!",
             nComputeTo);
21:       return;
22:     }
23:
24:     Console.WriteLine("{0}! ist {1}", nComputeTo, nFactorial);
25:   }
26: }
```

Der Klarheit halber wurden die einzelnen Blocks hier durch geschweifte Klammern markiert, auch wenn dies nicht in jedem Fall erforderlich war. Der Code enthält an der bewussten Stelle eine `checked`-Anweisung, so dass die Fehlerbehandlung auch dann zum Zuge kommen kann, wenn der angesprochene Compilerschalter nicht gesetzt wurde.

Wie Sie sehen, ist die Fehlerbehandlung selbst nichts Besonderes. Sie müssen eigentlich nur den fehleranfälligen Code in einen `try`-Block packen und die zu erwartende Ausnahme anschließend in einem `catch`-Block geeignet deklarieren und einer sorgfältigen Behandlung unterziehen. Die `return`-Anweisung erwirkt den Abbruch der Methode.

Falls Ihnen die Art der zu erwartenden Ausnahme nicht bekannt ist, können Sie, um auf der sicheren Seite zu sein, die Deklaration des Ausnahmeobjekts im `catch`-Block auch weglassen:

```
try
{
...
}
catch
{
...
}
```

Dieser Ansatz hat natürlich den Nachteil, dass im `catch`-Block keinerlei Informationen über die Art des aufgetretenen Fehlers verfügbar sind. Besser ist es, ein allgemeines Fehlerobjekt mit dem Typ `System.Exception`, der Basisklasse aller Ausnahmen für Laufzeitfehler, zu deklarieren. Die allgemeine Form für die Fehlerbehandlung mit `try` und `catch` sieht damit so aus:

```
try
{
...
}
catch(System.Exception e)
{
...
}
```

Beachten Sie, dass Sie das Objekt `e` weder in einem `ref`- oder `out`-Parameter übergeben noch als Linkswert in einer Zuweisung verwenden dürfen.

7.2.2 Aufräumcode mit *try…finally ausführen*

Wenn Ihnen mehr daran gelegen ist, nach einer Ausnahme aufzuräumen, als den Fehler selbst zu behandeln, sind Sie mit dem `try…finally`-Konstrukt am besten bedient. Es unterdrückt zwar die Fehlermeldung des Systems bei Auftreten einer Ausnahme nicht, gibt Ihnen aber die Möglichkeit, Code zu platzieren, der auf jeden Fall im Anschluss an den `try`-Block zur Ausführung kommt, gleich ob ein Abbruch der Routine erfolgt oder nicht.

Damit kann das Programm beispielsweise dem Benutzer noch eine Meldung ausgeben, bevor es vom Laufzeitsystem abgebrochen wird. Listing 7.4 zeigt diesen Fall:

Listing 7.4: Der Code in einer *`finally`* Anweisung wird auf jeden Fall ausgeführt

```
 1: using System;
 2:
 3: class Factorial
 4: {
 5:   public static void Main(string[] args)
 6:   {
 7:     long nFactorial = 1, nCurDig=1;
 8:     long nComputeTo = Int64.Parse(args[0]);
 9:     bool bAllFine = false;
10:
11:     try
12:     {
13:       checked
14:       {
15:         for (; nCurDig <= nComputeTo; nCurDig++)
16:           nFactorial *= nCurDig;
17:       }
18:       bAllFine = true;
19:     }
20:     finally
21:     {
22:       if (!bAllFine)
23:          Console.WriteLine("Überlauf bei der Berechnung von {0}!",
             nComputeTo);
24:       else
25:          Console.WriteLine("{0}! ist {1}", nComputeTo, nFactorial);
26:     }
27:   }
28: }
```

Nachdem der `finally`-Block sowohl im regulären als auch im Ausnahmefall zur Ausführung kommt, muss seine Logik so gestaltet sein, dass er auf beide Fälle entsprechend reagiert. Das Programm pflegt für diesen Zweck die Zustandsvariable `bAllFine`, die nur dann den Wert `true` annimmt, wenn der Code im `try`-Block keine Ausnahme produziert.

Falls Ihnen die strukturierte Ausnahmebehandlung von C++ geläufig ist, werden Sie sich vielleicht fragen, wie in C# das Gegenstück der `__leave`-Anweisung aussieht. (Wenn Sie diese Anweisung noch nicht kennen: `__leave` wird in C++ dafür eingesetzt, einen `try`-Block vorzeitig zu verlassen und unmittelbar in den `finally`-Block zu verzweigen.) Leider gibt es ein solches Gegenstück in C# nicht, so dass Sie darauf angewiesen sind, sich anders zu behelfen. Listing 7.5 zeigt wie:

Listing 7.5: Vom *try*-Block aus in den *finally*-Block verzweigen

```
 1: using System;
 2:
 3: class JumpTest
 4: {
 5:   public static void Main()
 6:   {
 7:     try
 8:     {
 9:       Console.WriteLine("try");
10:       goto __leave;
11:     }
12:     finally
13:     {
14:       Console.WriteLine("finally");
15:     }
16:
17:     __leave:
18:     Console.WriteLine("__leave");
19:   }
20: }
```

Das Programm generiert die folgende Ausgabe:

```
try
finally
__leave
```

Wie Sie sehen, muss sich auch die `goto`-Anweisung der Regel beugen, dass der `finally`-Block bei Verlassen des `try`-Blocks zur Ausführung kommt. Der Sprung zur Marke `__leave` erfolgt somit erst nach Abarbeitung des `finally`-Blocks. Und da die Sprungmarke unmittelbar auf den `finally`-Block folgt, geht die Kontrolle an die Anweisung über, die auch ohne Sprung als Nächstes zur Ausführung

gekommen wäre. Der Sprung hat demnach denselben Effekt wie die `__leave`-Anweisung des SEH: Das unmittelbare Verlassen des `try`-Blocks und die Ausführung des `finally`-Blocks.

Ein Skeptiker könnte hier natürlich den Verdacht hegen, dass der C#-Compiler die `goto`-Anweisung komplett unter den Tisch kehrt, weil sie die letzte Anweisung im `try`-Block war und die Kontrolle an dieser Stelle naturgemäß an den `finally`-Block übergeht. Um dies zu klären, setzen Sie die `goto`-Anweisung einfach vor den Aufruf der Methode `Console.WriteLine`. Sie erhalten dann zwar eine Compilerwarnung, dass der Methodenaufruf nicht mehr erreichbar sei, die Ausgabe des Programms zeigt aber, dass der Sprung stattfindet – die Zeile `"try"` fehlt nämlich.

7.2.3 Ausnahmen mit *try...catch...finally* voll im Griff

In der Praxis werden Sie wahrscheinlich beide Techniken für den Umgang mit Ausnahmen einsetzen: Sie fangen die Ausnahmen erst ab, um sie zu behandeln und räumen dann auf, damit die reguläre Ausführung des Programms weitergehen kann. Mit anderen Worten: Ihr Fehlerbehandlungscode arbeitet mit allen drei Anweisungen: `try`, `catch` und `finally`. Listing 7.6 zeigt, wie der Umgang mit Fehlern aussehen kann, die bei Divisionen durch Null ausgelöst werden:

Listing 7.6: Mehrere *catch*-Blöcke

```
 1: using System;
 2:
 3: class CatchIT
 4: {
 5:   public static void Main()
 6:   {
 7:     try
 8:     {
 9:       int nTheZero = 0;
10:       int nResult = 10 / nTheZero;
11:     }
12:     catch(DivideByZeroException divEx)
13:     {
14:       Console.WriteLine("Division durch Null!");
15:     }
16:     catch(Exception Ex)
17:     {
18:       Console.WriteLine("Andere Ausnahme");
19:     }
20:     finally
21:     {
22:     }
23:   }
24: }
```

Das Besondere an diesem Beispiel ist, dass der Code gleich zwei `catch`-Blöcke für denselben `try`-Block verwendet. Der erste `catch`-Block kümmert sich speziell um die häufiger auftretende Ausnahme `DivideByZeroException` und der zweite um alle restlichen Ausnahmen.

Es ist wichtig, dass Sie `catch`-Blöcke so anreihen, dass der Code für die Behandlung spezieller Ausnahmen vor dem für allgemeinere Ausnahmen zu stehen kommt. Das Beispiel in Listing 7.7 zeigt, was passiert, wenn Sie sich nicht an diese Reihenfolge halten:

Listing 7.7: *catch*-Blöcke in falscher Reihenfolge

```
 1: try
 2: {
 3:   int nTheZero = 0;
 4:   int nResult = 10 / nTheZero;
 5: }
 6: catch(Exception Ex)
 7: {
 8:   Console.WriteLine("Allgemeine Ausnahme" + Ex.ToString());
 9: }
10: catch(DivideByZeroException divEx)
11: {
12:   Console.WriteLine("Und sie ward nie gesehen...");
13: }
```

Der Compiler bemerkt den Verdreher und generiert eine Fehlermeldung nach folgendem Muster:

```
wrongcatch.cs(10,9): error CS0160: A previous catch clause already cat-
ches all exceptions of this or a super type ('System.Exception')
```

Zum Schluss dieser Betrachtung sollte nicht unerwähnt bleiben, das der Ausnahmemechanismus des NGWS-Laufzeitsystems gegenüber dem SEH folgenden Mangel (oder wenn Sie so wollen: Unterschied) aufweist: Es gibt kein Äquivalent für den in SEH-Ausnahmefiltern definierten Bezeichner `EXCEPTION_CONTINUE_EXECUTION`. Dieser Bezeichner ermöglicht im Wesentlichen die Wiederholung des für die Ausnahme verantwortlichen Codes, nachdem die Ursache des Fehlers behoben ist. Eine bevorzugte Speicherverwaltungstechnik des Autors war in diesem Zusammenhang beispielsweise die Speicheranforderung in Reaktion auf Ausnahmen aufgrund von Zugriffsverletzungen.

7.3 Ausnahmen signalisieren

Ausnahmen kommen nicht von irgendwoher: Sie werden konkret ausgelöst, wenn es etwas zu signalisieren gibt. Ihr eigener Code ist davon natürlich nicht ausgenommen. Sie können jederzeit selbst Ausnahmen auslösen, um dem Aufrufer einen Fehler anzuzeigen. Das Vorgehen ist einfach:

```
throw new ArgumentException("Argument darf nicht 5 sein");
```

Sie übergeben einer `throw`-Anweisung die Instanz einer Ausnahmeklasse. Die im Beispiel verwendete Ausnahmeklasse `ArgumentException` gehört zu den Standardausnahmen des NGWS-Laufzeitsystems, über die Tabelle 7.1 einen Überblick gibt.

Datentyp der Ausnahme	Beschreibung
`Exception`	Basisklasse für alle Ausnahmen
`SystemException`	Basisklasse für alle zur Laufzeit generierten Ausnahmen
`IndexOutOfRangeException`	signalisiert eine Bereichsverletzung für einen Arrayindex
`NullReferenceException`	signalisiert den Zugriff auf eine `null`-Referenz
`InvalidOperationException`	wird von verschiedenen Methoden signalisiert, wenn die Operation für den aktuellen Objektzustand nicht durchgeführt werden kann
`ArgumentException`	Basisklasse für Ausnahmen, die auf Argumentfehler zurückgehen
`ArgumentNullException`	wird von Methoden signalisiert, wenn für einen Parameter unerwartet das Argument `null` vorliegt
`ArgumentOutOfRangeException`	wird von Methoden signalisiert, wenn für das Argument eines Parameters eine Bereichsverletzung vorliegt
`InteropException`	Basisklasse für Ausnahmen, die auf Umgebungen außerhalb des NGWS-Laufzeitsystems bezogen sind oder daraus stammen
`ComException`	enthält `HRESULT`-Information im klassischen Stil des COM
`SEHException`	verkapselt Informationen im Stile der strukturierten Ausnahmebehandlung (SEH) des Win32

Tab. 7.1: Standardausnahmen des NGWS-Laufzeitsystems

Es ist nicht erforderlich, für jeden `throw`-Aufruf ein neues Ausnahmeobjekt anzulegen. Sofern in einem `catch`-Block bereits ein entsprechendes Objekt zur Hand ist, können Sie dieses jederzeit weiterverwenden. Andererseits gibt es auch Situationen, in denen keiner der in Tabelle 7.1 genannten standardmäßigen Ausnahmetypen Ihren speziellen Bedürfnissen gerecht wird. Sie werden dann nicht umhinkommen, selbst eine passende Ausnahmeklasse abzuleiten und zu implementieren. Die folgenden zwei Abschnitte vertiefen diese beiden Aspekte.

7.3.1 Ausnahmen weiterleiten

Bei der Implementation eines `catch`-Blocks stehen Sie vor der Wahl, ob Sie eine abgefangene Ausnahme vor Ort behandeln oder schlicht an den Aufrufer weiterleiten, um die Behandlung einem weiter außen gelegenen `try`…`catch`-Block zu überlassen. Listing 7.8 zeigt ein Beispiel für diesen Ansatz:

Listing 7.8: Eine Ausnahme an den Aufrufer weiterleiten

```
 1: try
 2: {
 3:   checked
 4:   {
 5:     for (; nCurDig <= nComputeTo; nCurDig++)
 6:       nFactorial *= nCurDig;
 7:   }
 8: }
 9: catch (OverflowException oe)
10: {
11:   Console.WriteLine("Überlauf bei der Berechnung von {0}!",
            nComputeTo);
12:   throw;
13: }
```

Beachten Sie, dass der `throw`-Aufruf hier ohne die Angabe eines Ausnahmeobjekts auskommt. Natürlich hätten Sie auch schreiben können:

```
throw oe;
```

Von nun an ist die Behandlung der Ausnahme Sache des Aufrufers!

7.3.2 Eigene Ausnahmeklassen definieren

Trotz der generellen Empfehlung, sich auf die standardmäßigen Ausnahmeklassen des NGWS-Laufzeitsystems zu beschränken, kann es in verschiedenen Szenarien durchaus vorteilhaft sein, mit eigenen Ausnahmeklassen zu arbeiten. Sie bieten dem Empfänger den Vorteil, dass er auf solche Ausnahmen differenzierter reagieren kann als auf standardmäßige Ausnahmen.

Listing 7.9 stellt die Implementation einer eigenen Ausnahmeklasse namens `MyImportantException` vor. Sie hält sich an zwei Regeln: Erstens trägt der Klassenbezeichner das Suffix `Exception` und zweitens definiert die Klasse alle drei für Ausnahmeklassen empfohlenen Konstruktoren. An diese Regeln sollten auch Sie sich halten:

Listing 7.9: Implementation der eigenen Ausnahmeklasse *MyImportantException*

```
 1: using System;
 2:
 3: public class MyImportantException:Exception
 4: {
 5:   public MyImportantException()
 6:     :base() {}
 7:
 8:   public MyImportantException(string message)
 9:     :base(message) {}
10:
11:   public MyImportantException(string message, Exception inner)
12:     :base(message,inner) {}
13: }
14:
15: public class ExceptionTestApp
16: {
17:   public static void TestThrow()
18:   {
19:     throw new MyImportantException("Etwas Schreckliches ist
        passiert");
20:   }
21:
22:   public static void Main()
23:   {
24:     try
25:     {
26:       ExceptionTestApp.TestThrow();
27:     }
28:     catch (Exception e)
29:     {
30:       Console.WriteLine(e);
31:     }
32:   }
33: }
```

Wie Sie sehen, enthält die Implementation von `MyImportantException` keinerlei Spezialitäten, sondern verlässt sich vollständig auf die Basisklasse `System.Exception`. Der restliche Code des Programms testet die Ausnahmeklasse mittels eines `catch`-Blocks, der auf Ausnahmen des Typs `System.Exception` zugeschnitten ist.

Nachdem sich die Implementation der Klasse eigentlich nur auf die standardmäßige Ausnahmeklasse `System.Exception` stützt – die drei Konstruktoren rufen schlicht die jeweilige Basisklassenvariante auf – stellt sich natürlich die Frage, was die Ableitung einer eigenen Ausnahmeklasse überhaupt für einen Sinn hat. Das Besondere daran ist natürlich der neue Typ. Er kann in einem `catch`-Block als Auswahlkriterium fungieren und erlaubt es, Code speziell auf diesen Ausnahmetyp zuzuschneiden.

Wenn Sie eine Klassenbibliothek mit eigenem Namensraum implementieren, sollten Sie die dazugehörigen Ausnahmeklassen gleichfalls in diesen Namensraum packen. Darüber hinaus bietet es natürlich Vorteile, wenn Ihre Ausnahmeklassen Eigenschaften für erweiterte Fehlerinformationen definieren – auch wenn das vorangegangene Beispiel darauf verzichtet hat.

7.4 Ausnahmen richtig dosiert

Abschließend noch eine Liste mit Ratschlägen für den wohldosierten Umgang mit Ausnahmen:

- Geben Sie jeder Ausnahme, die Sie signalisieren, einen aussagekräftigen Fehlertext mit.
- Signalisieren Sie nur dann eine Ausnahme, wenn es die Situation wirklich erfordert, das heißt, wenn ein Funktionswert oder Ausgabeparameter für die gleiche Aufgabe nicht in Frage kommt.
- Signalisieren Sie eine Ausnahme des Typs `ArgumentException`, wenn für einen Aufrufparameter ein fehlerhafter Wert übergeben wurde.
- Signalisieren Sie eine Ausnahme des Typs `InvalidOperationException`, wenn die gewünschte Operation aufgrund des aktuellen Objektzustands nicht durchführbar ist.
- Signalisieren Sie immer den Ausnahmetyp, der für die Situation am besten passt.
- Arbeiten Sie mit verketteten Ausnahmen – dadurch wird es möglich, den Weg einer Ausnahme zurückzuverfolgen.
- Unterlassen Sie es, Ausnahmen für gewöhnliche Situationen oder reguläre Fehler zu signalisieren.
- Unterlassen Sie es, Ausnahmen im Rahmen der regulären Programmlogik einzusetzen
- Unterlassen Sie es, Ausnahmen des Typs `NullReferenceException` oder `IndexOutOfRangeException` innerhalb von Methoden zu signalisieren.

7.5 Zusammenfassung

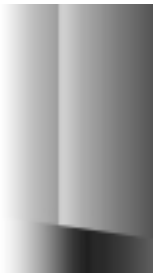
Ausgangspunkt für dieses Kapitel war der Umgang mit Überläufen. Ein Compilerschalter erlaubt es Ihnen, die in C# standardmäßig unterdrückte Signalisierung von Überläufen für die gesamte Anwendung einzuschalten. Falls Sie eine differenzierte Kontrolle wünschen, stehen Ihnen die Anweisungen `checked` und `unchecked` zur Verfügung, die die Überlaufkontrolle blockorientiert und unabhängig von den Compilereinstellungen ein- bzw. ausschalten.

Erkennt das Laufzeitsystem einen Überlauf, signalisiert es diesen als Ausnahme. Das Kapitel hat drei Ansätze diskutiert, wie Ihr Code auf eine Ausnahme reagieren kann. Der allgemeinste Ansatz beinhaltet die Implementation von `try`-, `catch`- und `finally`-Blöcken. Anhand verschiedener Beispiele haben Sie darüber hinaus erfahren, welche Unterschiede der Ausnahmemechanismus von C# gegenüber der strukturierten Ausnahmebehandlung (SEH) des Win32 aufweist.

Die Behandlung von Ausnahmen wird erforderlich, wenn Sie mit fertigen Klassen arbeiten, die Signalisierung dagegen, wenn Sie Ihre eigenen Klassen implementieren. Für die Signalisierung sind drei Situationen zu unterscheiden: Es kann erstens notwendig sein, nicht behandelbare Ausnahmen als solche weiterzuleiten, zweitens, erkannte Fehler in Form standardmäßiger Ausnahmen des NGWS-Laufzeitsystems zu signalisieren, und drittens, eigene Ausnahmeklassen zu implementieren, um Fehler in spezifischen Zusammenhängen hinreichend präzisieren zu können.

Den Ausklang des Kapitels bildeten schließlich noch verschiedene Ratschläge für den wohldosierten und bedachten Umgang mit Ausnahmen.

Kapitel 8



Komponenten mit C# schreiben

In diesem Kapitel geht es um das Schreiben von Komponenten in C#: Wie entsprechende Quelltexte aussehen müssen, wie man sie kompiliert, und wie sich das Ergebnis in einer Client-Anwendung verwenden lässt. Die im zweiten Teil des Kapitels behandelten Namensräume helfen bei der Organisation von Anwendungen.

Die folgenden Seiten sind also in zwei große Abschnitte unterteilt:

- die erste Komponente
- Namensräume

8.1 Die erste Komponente

Die bis dato vorgestellten Beispiele haben sämtlich Klassen in ein und derselben Anwendung sowohl deklariert als auch eingesetzt – was nicht unbedingt so sein muss: Wenn Sie eine Klasse als Komponente formulieren und in einer separaten Datei unterbringen, dann lässt sich diese Klasse auch von mehreren Anwendungen aus nutzen.

In C# geschriebene Komponenten werden in DLLs untergebracht, unterscheiden sich aber stark von den üblicherweise in C++ geschriebenen COM-Komponenten; unter anderem bekommen Sie es bei C# mit wesentlich weniger Details der von COM her gewohnten Infrastruktur zu tun. Die folgenden Abschnitte zeigen, wie man in C# eine Komponente erstellt, kompiliert und schließlich in einer Anwendung einsetzt:

- Erstellen einer Komponente
- Kompilieren von Komponenten
- Erstellen einer einfachen Client-Anwendung

8.1.1 Erstellen einer Komponente

Obwohl es primär um eine Demonstration geht, versteht sich das folgende Beispiel nicht als Selbstzweck: Es geht um eine Klasse, die eine Webseite von einem Server lädt und den HTML-Code als String zur weiteren Verwendung zur Verfügung stellt. Allzu kompliziert ist diese Klasse nicht, weil sie den größten Teil der Arbeit dem NWGS-Subsystem überlässt.

Die Klasse `RequestWebPage` hat zwei Konstruktoren, eine Eigenschaft und eine Methode. Die Eigenschaft trägt den Namen `URL` und speichert die Adresse der Webseite, deren Inhalt sich über die Methode `GetContent` laden lässt.

Listing 8.1: Die Klasse `RequestWebPage` für das Herunterladen von Webseiten

```
 1: using System;
 2: using System.Net;
 3: using System.IO;
 4: using System.Text;
 5:
 6: public class RequestWebPage
 7: {
 8:   private const int BUFFER_SIZE = 128;
 9:   private string m_strURL;
10:
11:   public RequestWebPage()
12:   {
13:   }
14:
15:   public RequestWebPage(string strURL)
16:   {
17:     m_strURL = strURL;
18:   }
19:
20:   public string URL
21:   {
22:     get { return m_strURL;  }
23:     set { m_strURL = value; }
24:   }
25:   public void GetContent(out string strContent)
26:   {
27:     // ist ein URL angegeben?
28:     if (m_strURL == ""
29:       throw new ArgumentException("URL muss gesetzt sein!");
30:
31:     WebRequest theRequest =
               (WebRequest) WebRequestFactory.Create(m_strURL);
32:     WebResponse theResponse = theRequest.GetResponse();
33:
34:     // Bytepuffer für die Antwort anlegen
35:     int BytesRead = 0;
36:     Byte[] Buffer = new Byte[BUFFER_SIZE];
37:
38:     Stream ResponseStream = theResponse.GetResponseStream();
39:     BytesRead = ResponseStream.Read(Buffer, 0, BUFFER_SIZE);
40:
41: // StringBuilder tut sich mit schrittweise vergrößerten Puffern
      leichter
42:     StringBuilder strResponse = new StringBuilder("");
43:     while (BytesRead != 0 )
44:     {
```

```
45:       strResponse.Append(Encoding.ASCII.GetString(Buffer, 0,
          BytesRead));
46:       BytesRead = ResponseStream.Read(Buffer, 0, BUFFER_SIZE);
47:     }
48:
49:     // Ausgangsparameter mit dem Ergebnis besetzen
50:     strContent = strResponse.ToString();
51:   }
52: }
```

Mit einem einzigen, parameterlosen Konstruktor käme die Klasse natürlich auch aus. In der Praxis wird es aber meist so sein, dass man beim Anlegen eines Objekts die zu lesende Webadresse bereits kennt und auch angeben wird. Spätere Änderungen der Webadresse – beispielsweise zum Lesen einer weiteren Webseite – sind ohne weiteres über die `get`- und `set`-Zugriffsfunktionen für die Eigenschaft `URL` möglich.

Interessant wird die Sache in der Methode `GetContent`. Sie prüft erst einmal auf recht primitive Weise den Wert von `URL` – falls dort eine leere Zeichenfolge steht, löst sie eine `ArgumentException` aus – und fordert dann von der Klasse `WebRequestFactory` das Anlegen eines `WebRequest`-Objekts für den Wert von `URL` an.

Objekte dieser Klasse bieten eine Vielzahl von Methoden für das Senden von Cookies, zusätzlichen Kopfinformationen, Abfragestrings usw., die hier sämtlich ungenutzt bleiben: `GetContent` fordert in Zeile 32 ohne weitere Umschweife direkt eine Antwort in Form eines `WebResponse`-Objekts an. (Wenn Sie die Abfrage der Webseite in irgendeiner Weise modifizieren wollen – eben beispielsweise durch Cookies, – dann müssen Sie die entsprechenden Eigenschaften und Methoden von `WebRequest` vor dem Anfordern der Antwort einsetzen.)

Die Anweisungen in den Zeilen 35 und 36 initialisieren einen Bytepuffer, der dann zum Lesen der Webseite über den mit `GetResponseStream` angelegten Stream verwendet wird. Die `while`-Schleife endet erst, wenn eine Leseoperation 0 Bytes ergibt, also das Ende des Streams erreicht ist.

Auf den ersten Blick sieht es so aus, als wenn sich die `while`-Schleife die Sache unnötig kompliziert machen würde: Anstatt den Bytepuffer einfach an eine Zeichenkette anzuhängen, verwendet sie ein `StringBuilder`-Objekt. Zur Erklärung sehen Sie sich bitte einmal das folgende Beispiel an:

```
strMyString = strMyString + "und noch etwas Text";
```

Ganz offensichtlich geht es hier um das Kopieren von Werten: Der konstante Wert `"und noch etwas Text"` wird implizit in ein `string`-Objekt konvertiert (Stichwort: Boxing), und für das Aneinanderhängen der beiden Zeichenketten ist ein weiteres `string`-Objekt fällig, das dann schließlich `strMyString` zugewiesen wird. Anders

gesagt: Sowohl die anzuhängende Zeichenfolge als auch der vorherige Wert von `strMyString` werden mehrfach kopiert.

Schön wäre es natürlich, wenn

```
strMyString += "und noch etwas Text";
```

sich mit weniger Kopieraktionen begnügen würde – was aber in C# leider nicht der Fall ist: Der (entsprechend überladene) Operator += ist für Zeichenketten exakt so implementiert wie zuvor beschrieben.

Und genau dies rechtfertigt den Einsatz der Klasse `StringBuilder`. Sie verwendet für das Anhängen, Einfügen, Herausnehmen und Ersetzen von Zeichen einen einzigen Puffer und kommt mit einem Minimum an Kopieroperationen aus.

Womit eigentlich nur noch ein einziges interessantes Detail dieser Klasse unerwähnt geblieben wäre: Die Umsetzung der Codierung in Zeile 45. `Encoding.ASCII` sorgt schlicht dafür, dass das Programm die Zeichen in dem Zeichensatz zu sehen bekommt, den es anfordert.

Nach dem Einlesen und Konvertieren des gesamten Streams fordert die Methode schließlich den Wert des `StringBuilder`-Objekts als Zeichenkette an und weist ihn der Ausgangsvariablen zu. Alternativ hätte man dieser Methode auch ein direktes Funktionsergebnis vom Typ `string` geben können – allerdings nur um den Preis einer weiteren Kopieraktion.

8.1.2 Kompilieren von Komponenten

Obwohl die Klasse `RequestWebPage` eine Komponente werden soll, unterscheidet sich der Quelltext in keiner Weise von dem für eine Klasse, die ausschließlich innerhalb einer einzigen Anwendung existiert. Tatsächlich wird der Speicherort einer Klasse erst bei der Kompilierung festgelegt. Der folgende Aufruf erzeugt eine Bibliothek anstelle einer Anwendung:

```
csc /r:System.Net.dll /t:library /out:wrq.dll requestwebpage.cs
```

Der Schalter `/t:library` weist den C#-Compiler an, eine Bibliothek zu erstellen, ohne nach einer statischen Methode `Main` zu suchen. Da hier erneut der Namensraum `System.Net` verwendet wird, muss die dazugehörige Bibliothek `System.Net.dll` explizit über den Schalter `/r:` angegeben werden.

Die bei dieser Kompilierung erzeugte Bibliothek `wrq.dll` lässt sich nun ähnlich wie `System.Net.dll` in Anwendungen einsetzen. Da es in diesem Kapitel ausschließlich um private Komponenten geht, können Sie sich das Kopieren von `wrq.dll` in einen speziellen Ordner sparen – es reicht, wenn diese Datei im selben Verzeichnis zu finden ist wie die Anwendung, die sie einsetzen.

8.1.3 Erstellen einer einfachen Client-Anwendung

Nachdem die Komponente geschrieben und fehlerfrei kompiliert worden ist, fehlt nur noch ein Anwendungsbeispiel dafür. Das folgende Listing zeigt ein einfaches, kommandozeilenorientiertes Programm, das zur Abfrage einer vom Autor unterhaltenen Webseite verwendbar ist.

Listing 8.2: Einsatz der Klasse *RequestWebPage* zum Abfragen einer einfach gehaltenen Webseite

```
 1: using System;
 2:
 3: class TestWebReq
 4: {
 5:   public static void Main()
 6:   {
 7:     RequestWebPage wrq = new RequestWebPage();
 8:     wrq.URL = "http://www.alphasierrapapa.com/iisdev/";
 9:
10:     string strResult;
11:     try
12:     {
13:       wrq.GetContent(out strResult);
14:     }
15:     catch (Exception e)
16:     {
17:       Console.WriteLine(e);
18:       return;
19:     }
20:
21:     Console.WriteLine(strResult);
22:   }
23: }
```

Beachten Sie, dass der Aufruf von `GetContent` durch `try...catch` geklammert ist: Zum einen kann die Methode selbst eine Ausnahme des Typs `ArgumentException` signalisieren, zum anderen reagieren die von `GetContent` verwendeten NGWS-Klassen auf Probleme ihrerseits mit Ausnahmen. Da die Klasse `RequestWebPage` Ausnahmen nicht selbst behandelt, muss sich das Programm darum kümmern.

Der Rest des Codes bedarf keiner ausgiebigen Erklärung mehr: `Main` ruft den Standardkonstruktor der Klasse auf, setzt die Eigenschaft `URL` des neu angelegten Objekts und benutzt dann seine Methode `GetContent`. Interessanter ist da schon der Aufruf des Compilers – ihm müssen Sie schließlich mitteilen, in welcher Bibliothek er die Klasse `RequestWebPage` suchen soll:

```
csc /r:wrq.dll testwebreq.cs
```

Mehr bleibt nicht zu tun, einmal abgesehen von der recht bescheidenen Darstellungsform: Die gelesenen Daten huschen ohne weitere Interpretation schlicht über das Konsolenfenster. Natürlich steht es Ihnen frei, einen Parser für den HTML-Code der Webseite zu schreiben, der Spezifisches extrahiert, und das Ganze schließlich als weitere Komponente in Ihrer eigenen Bibliothek zu speichern. Dem Autor schwebt eine SSL-taugliche Version der Komponente vor, die sich für die Online-Verifizierung von Kreditkartennummern in ASP+-Webseiten einsetzen lässt.

Es dürfte Ihnen aufgefallen sein, dass das Beispielprogramm ohne ein eigenes `using` für die verwendete Bibliothek auskommt. Das liegt daran, dass der Quelltext der Komponente keinen eigenen Namensraum definiert. Womit wir beim zweiten Hauptthema dieses Kapitels angekommen wären.

8.2 Namensräume

Namensräume wie `System` und `System.Net` sind Ihnen im Rahmen der vorangehenden Beispiele dieses Buches bereits des Öfteren begegnet. C# verwendet Namensräume zur Organisation von Programmen, wobei es die hierarchische Natur dieser Organisation einfach macht, Elemente eines Programms anderen Programmen zur Verfügung zu stellen. Tatsächlich muss es nicht einmal um die öffentliche Darstellung gehen: Namensräume erleichtern nämlich auch die interne Organisation von Anwendungen.

Obwohl Sie C# nicht dazu zwingt, sollten Sie die Hierarchie innerhalb von Anwendungen grundsätzlich durch Namensräume deutlich machen. Das NGWS-Subsystem ist ein gutes Beispiel dafür, wie eine solche Hierarchie aufgebaut sein kann.

Der folgende Codeauszug zeigt die Definition eines einfachen Namensraums `My.Test` (wobei der Punkt die Hierarchieebenen unterscheidet) in einem C#-Quelltext:

```
namespace My.Test
{
   // Definitionen und Deklarationen in diesem Namensraum
}
```

Zugriffe auf Elemente eines Namensraums sind sowohl über einen vollständig qualifizierten Bezeichner als auch über die Direktive `using`, die sämtliche Elemente des angegebenen Namensraums in den aktuellen Namensraum importiert, möglich. Beide Techniken finden Sie in den vorangehenden Beispielen dieses Buchs.

Bevor Sie nun selbst Namensräume definieren, noch ein paar Worte zur Verfügbarkeit der Bezeichner. Solange Sie keinen Modifizierer für die Verfügbarkeit

angeben, setzt der Compiler als Standardvorgabe `internal` ein. Soll ein Bezeichner öffentlich verfügbar sein, verwenden Sie das Schlüsselwort `public`. Andere Modifizierer für die Verfügbarkeit sind in diesem Zusammenhang nicht erlaubt.

Und damit wieder einmal genug der Theorie. In den folgenden Abschnitten geht es um praktische Beispiele für die folgenden Techniken:

- Einbetten einer Klasse in einen Namensraum
- Einsatz eigener Namensräume in einer Anwendung
- Mehrere Klassen mit gemeinsamem Namensraum

8.2.1 Einbetten einer Klasse in einen Namensraum

Zur Praxisdemonstration selbst definierter Namensräume bietet sich der englische Originaltitel dieses Buchs an; Um Sie aber nicht mit dem schlichten Verpacken der bereits diskutierten Komponente `RequestWebPage` in einen Namensraum `Presenting.CSharp` zu langweilen, stellt das folgende Beispiel eine komplett neue Klasse namens `WhoisLookup` vor, mit der sich *Whois*-Informationen abrufen lassen.

Listing 8.3: Implementation der Klasse *`WhoisLookup`* in einem eigenen Namensraum

```
 1: using System;
 2: using System.Net.Sockets;
 3: using System.IO;
 4: using System.Text;
 5:
 6: namespace Presenting.CSharp
 7: {
 8:  public class WhoisLookup
 9:  {
10:   public static bool Query(string strDomain, out string strWhoisIn-
fo)
11:   {
12:     const int BUFFER_SIZE = 128;
13:
14:     if ("" == strDomain)
15:       throw new ArgumentException("Domainname fehlt");
16:
17:     TCPClient tcpc = new TCPClient();
18:     strWhoisInfo = "N/A";
19:
20:     // Verbindungsversuch zum Whois-Server
21:     if (tcpc.Connect("whois.networksolutions.com", 43) != 0)
22:         return false;
23:
```

```
24:     // Stream der TCP-Verbindung
25:     Stream s = tcpc.GetStream();
26:
27:     // Anfrage senden
28:     strDomain += "\r\n";
29:     Byte[] bDomArr = Encoding.ASCII.GetBytes(strDomain.ToChar-
        Array());
30:     s.Write(bDomArr, 0, strDomain.Length);
31:
32:     Byte[] Buffer = new Byte[BUFFER_SIZE];
33:     StringBuilder strWhoisResponse = new StringBuilder("");
34:
35:     int BytesRead = s.Read(Buffer, 0, BUFFER_SIZE);
36:     while (BytesRead != 0 )
37:     {
38:      strWhoisResponse.Append(Encoding.ASCII.GetString(Buffer, 0,
        BytesRead));
39:      BytesRead = s.Read(Buffer, 0, BUFFER_SIZE);
40:     }
41:
42:     tcpc.Close();
43:     strWhoisInfo = strWhoisResponse.ToString();
44:     return true;
45:   }
46:  }
47: }
```

Der Namensraum `Presenting.CSharp` wird in Zeile 6 deklariert; er umschließt die Klasse `WhoisLookup` mit geschweiften Klammern in den Zeilen 7 und 47. Falls Sie nun noch auf weitere Erklärungen warten: Mehr ist zur Definition eines neuen Namensraums wirklich nicht zu tun.

Der Code von `WhoisLookup` verdient dagegen durchaus noch einige Worte – vor allem, weil er zeigt, wie einfach sich in C# Sockets einsetzen lassen. Nach der (mit Sicherheit verbesserungswürdigen) Prüfung des Domainnamens legt die statische Methode *Query* ein `TCPClient`-Objekt an, das die gesamte Kommunikation mit dem Whois-Server über den Port 43 übernimmt. Die Verbindung zum Server wird in Zeile 21 hergestellt:

```
if (tcpc.Connect("whois.networksolutions.com", 43) != 0)
```

Da das Fehlschlagen dieses Verbindungsversuchs sehr wohl zu erwarten ist, reagiert die Methode darauf nicht mit einer Ausnahme – erinnern Sie sich noch an die im vorangehenden Kapitel genannten Regeln? Das direkte Funktionsergebnis ist ein schlichter Fehlercode; 0 steht dagegen für eine zustande gekommene Verbindung.

Whois-Abfragen setzen erst einmal das Senden von Daten voraus – nämlich des Domainnamens, zu dem Informationen gewünscht werden. In Zeile 25 holt sich die Methode deshalb eine Referenz auf den (bidirektionalen) Stream der TCP-Verbindung. Der Domainname bekommt ein CR/LF-Paar angehängt, um das Ende der Anfrage zu markieren, wird in ein Byte-Array umgepackt und dann zum Whois-Server gesendet (Zeile 30).

Der Rest des Codes dürfte Ihnen von der Klasse `RequestWebPage` her recht bekannt vorkommen: Eine `while`-Schleife, in der die Antwort des Servers in einen Puffer eingelesen wird, und ein `StringBuilder`-Objekt, das die einzelnen Pufferinhalte aneinander hängt. Nach dem Ende der Leseaktion wird die Verbindung explizit über `Close` geschlossen, bevor die Methode das Ergebnis an den Aufrufer weiterreicht. Im Prinzip könnte man das auch dem Garbage Collector überlassen – nur gehören TCP-Verbindungen zu den Ressourcen, die man keine Millisekunde länger belegen sollte als unbedingt nötig.

Bevor Sie diese Klasse in einer Anwendung einsetzen können, müssen Sie sie als Komponente zu einer Bibliothek kompilieren. Obwohl der Quelltext einen Namensraum definiert, enthält die dafür notwendige Kommandozeile keine neuen Elemente:

```
csc /r:System.Net.dll /t:library /out:whois.dll whoislookup.cs
```

Wenn Sie den Quelltext unter dem Namen `whois.cs` speichern, können Sie sich den Schalter `/out:` im Prinzip sparen, weil Bibliotheken als Standardvorgabe denselben Namen wie der C#-Quelltext (nur eben mit der Erweiterung .dll) bekommen. Dass man `/out:` auch in einem solchen Fall angeben sollte, ist eher eine Frage sinnvoller Gewohnheiten. Die meisten Projekte bestehen aus mehreren Quelltextdateien, und ohne explizite Festlegung per `/out:` verwendet der Compiler für die Bibliothek dann einfach den Namen des zuerst angegebenen Quelltexts.

8.2.2 Einsatz eigener Namensräume in einer Anwendung

Da die Komponente einen eigenen Namensraum hat, muss eine Anwendung diesen Namensraum entweder mit

```
using Presenting.CSharp;
```

importieren oder für seine Elemente vollständig qualifizierte Bezeichner verwenden – beispielsweise:

```
Presenting.CSharp.WhoisLookup.Query(...);
```

So lange keine Konflikte durch gleichnamige Bezeichner verschiedener Namensräume zu erwarten sind, sollte dem Import mit `using` der Vorzug gegeben werden – nicht nur, weil das die Tipparbeit kräftig reduziert. Das folgende Listing zeigt einen einfachen Client der neuen Komponente.

Listing 8.4: Eine Beispielanwendung für die Komponente *`WhoisLookup`*

```
 1: using System;
 2: using Presenting.CSharp;
 3:
 4: class TestWhois
 5: {
 6:   public static void Main()
 7:   {
 8:     string strResult;
 9:     bool bReturnValue;
10:
11:     try
12:     {
13:       bReturnValue = WhoisLookup.Query("microsoft.com", out
          strResult);
14:     }
15:     catch (Exception e)
16:     {
17:       Console.WriteLine(e);
18:       return;
19:     }
20:     if (bReturnValue)
21:       Console.WriteLine(strResult);
22:     else
23:       Console.WriteLine("Keine Informationen vom Whois-Server
          erhalten.");
24:   }
25: }
```

In Zeile 2 wird der Namensraum `Presenting.CSharp` mit `using` importiert. Darauf folgende Anweisungen, die sich auf die Klasse `WhoisLookup` beziehen, kommen deshalb ohne explizite Angabe dieses Namensraums aus.

Das Programm fragt die Whois-Informationen für die Domain microsoft.com ab, wobei sich dieser Domainname im Quelltext auch durch einen Namen Ihrer Wahl ersetzen lässt. Tatsächlich wäre es natürlich sinnvoller, wenn man den Domainnamen beim Start des Programms über die Kommandozeile angeben könnte. Das folgende Listing zeigt eine entsprechende Variante des Programms, die der Kürze halber auf eine Ausnahmebehandlung verzichtet:

Listing 8.5: Übergabe eines Kommandozeilenparameters an die Methode *Query*

```
 1: using System;
 2: using Presenting.CSharp;
 3:
 4: class WhoisShort
 5: {
 6:   public static void Main(string[] args)
 7:   {
 8:     string strResult;
 9:     bool bReturnValue;
10:
11:     bReturnValue = WhoisLookup.Query(args[0], out strResult);
12:
13:     if (bReturnValue)
14:       Console.WriteLine(strResult);
15:     else
16:       Console.WriteLine("Whois-Anfrage gescheitert.");
17:   }
18: }
```

Sie kompilieren dieses kleine Programm wie gehabt mit:

```
csc /r:whois.dll whoisshort.cs
```

Der Name der abzufragenden Domain wird nun beim Start des Programms in der Kommandozeile angegeben. Für die Abfrage von microsoft.com beispielsweise:

```
whoisclnt microsoft.com
```

Welche Registrierungsinformationen bei einer fehlerfreien Abfrage für microsoft.com herauskommen, gibt das nächste Listing in leicht verkürzter Fassung wieder. Tatsächlich ist `whoisshort` ein recht nützliches Werkzeug, das komponentenbasiert arbeitet und insgesamt in weniger als einer Stunde geschrieben wurde. Wie lange das wohl mit C++ gedauert hätte?

Listing 8.6: Whois-Information über microsoft.com (gekürzt)

```
D:\CSharp\Samples\Namespace>whoisshort
...

Registrant:
Microsoft Corporation (MICROSOFT-DOM)
   1 microsoft way
   redmond, WA 98052
   US

   Domain Name: MICROSOFT.COM
```

```
   Administrative Contact:
      Microsoft Hostmaster  (MH37-ORG)  msnhst@MICROSOFT.COM
   Technical Contact, Zone Contact:
      MSN NOC  (MN5-ORG)  msnnoc@MICROSOFT.COM
   Billing Contact:
      Microsoft-Internic Billing Issues  (MDB-ORG)  msnbill@MICRO-
SOFT.COM

   Record last updated on 20-May-2000.
   Record expires on 03-May-2010.
   Record created on 02-May-1991.
   Database last updated on 9-Jun-2000 13:50:52 EDT.

   Domain servers in listed order:

   ATBD.MICROSOFT.COM          131.107.1.7
   DNS1.MICROSOFT.COM          131.107.1.240
   DNS4.CP.MSFT.NET            207.46.138.11
   DNS5.CP.MSFT.NET            207.46.138.12
```

8.2.3 Mehrere Klassen mit gemeinsamem Namensraum

Wenn Sie die Klassen `WhoisLookup` und `RequestWebPage` beide im Namensraum `Presenting.CSharp` unterbringen wollen, haben Sie nicht allzu viel Arbeit vor sich. Da `WhoisLookup` bereits Teil dieses Namensraums ist, beschränken sich die Änderungen auf die Datei `requestwebpage.cs` und bestehen aus wenigen Zeilen:

```
namespace Presenting.CSharp
{
  public class RequestWebPage
  {
  ...
  }
}
```

Obwohl die beiden Klassen in getrennten Dateien untergebracht sind, werden sie durch den folgenden Aufruf des Compilers in einen Namensraum zusammengefasst:

```
csc /r:System.Net.dll /t:library /out:presenting.csharp.dll whois-
lookup.cs requestwebpage.cs
```

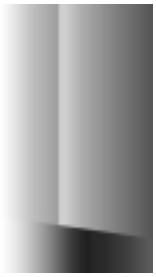
Der in diesem Beispiel mit `/out:` festgelegte Name der DLL ist nicht zwingend. Im allgemeinen verwendet man für solche Bibliotheken aber den Namensraum auch als Dateinamen, weil sich dann der Zusammenhang zwischen `using`-Direktiven und bei der Kompilierung anzugebenden Bibliotheken von selbst ergibt.

8.3 Zusammenfassung

In diesem Kapitel ging es um das Erstellen von Komponenten – also Klassen, die sich nicht nur in einer, sondern in einer beliebigen Zahl von Anwendungen (Clients) einsetzen lassen. Die Vergabe von Namensräumen für solche Komponenten ist nicht zwingend, hilft aber sowohl beim Aufbau einer Klassenhierarchie als auch bei der internen Organisation einzelner Anwendungen.

Komponenten lassen sich in C# ausgesprochen einfach erstellen. Installiert werden müssen sie nicht: Es reicht, wenn sich die Bibliothek im selben Verzeichnis befindet wie ihre Clients. Dieses Bild ändert sich allerdings, wenn es um Klassenhierarchien geht, die überall zur Verfügung stehen sollen. Mit dem Wie und Warum beschäftigt sich das nächste Kapitel.

Kapitel 9



Konfiguration und Verbreitung

Das vorige Kapitel hat Ihnen gezeigt, wie Sie eine Komponente implementieren und im Rahmen einer einfachen Testanwendung einsetzen. Obwohl die vorgestellte Komponente vom Prinzip her fertig ist, sollte man ihr noch die eine oder andere Technik angedeihen lassen.

- Bedingtes Kompilieren
- Kommentarbasierte Dokumentation in XML
- Versionspflege

9.1 Bedingtes Kompilieren

Ein Feature, auf das die wenigsten Programmierer verzichten wollen, ist die bedingte Kompilation. Diese Technik eröffnet die Möglichkeit, Teile des Codes bedingt ein- und auszublenden, etwa um Debug-, Demo- oder Release-Versionen einer Anwendung auf Grundlage ein und desselben Quelltexts zu generieren.

C# unterstützt zwei unterschiedliche Formen der bedingten Übersetzung:

- Einsatz des Präprozessors
- `conditional`-Attribut

9.1.1 Einsatz des Präprozessors

In C++ verkörpert der Präprozessor einen separaten Lauf über den Quellcode, der der eigentlichen Kompilierung vorgeschaltet ist. Anders in C#: Hier »emuliert« der Compiler den Präprozessor, und es gibt auch keinen separaten Lauf. Der Präprozessor von C# ist nicht mehr (und nicht weniger) als ein Mechanismus für die bedingte Kompilierung!

Obwohl der C#-Compiler keine Makros unterstützt, kennt er alle notwendigen Anweisungen, um das bedingte Ein- und Ausblenden von Code auf Basis von Symboldefinitionen zu ermöglichen. Die folgenden Abschnitte stellen Ihnen die entsprechenden C#-Direktiven vor, die denen von C++ recht ähnlich sind.

- Definition von Symbolen
- Code mit Hilfe von Symbolen ein- und ausblenden
- Fehlermeldungen und Warnungen generieren

Definition von Symbolen

Makros lassen mit der Präprozessorfunktionalität von C# nicht definieren, wohl aber Symbole. Darüber hinaus kann der Präprozessor bestimmte Codeanteile ein- und ausblenden, je nachdem, ob ein bestimmtes Symbol definiert ist oder nicht.

Die eine Möglichkeit, ein Symbol zu definieren, besteht darin, eine `#define`-Direktive dafür an den Anfang der C#-Quelldatei zu setzen:

```
#define DEBUG
```

Diese Zeile definiert das Symbol `DEBUG` für den gesamten Bereich der Quelldatei. Beachten Sie, dass Symboldefinitionen vor allen anderen Anweisungen zu treffen sind. So nötigt der folgende Codeauszug den Compiler beispielsweise zu einer Fehlermeldung:

```
using System;
#define DEBUG
```

Weiterhin kann man auch den Compiler für die Symboldefinition benutzen, indem man den Compilerschalter `/define` verwendet. Die so definierten Symbole sind dann in allen Quelldateien bekannt, die der jeweilige Lauf umfasst:

```
csc /define:DEBUG mysymbols.cs
```

Um mehrere Symbole mit dem Compiler zu definieren, setzen Sie die Symbole hintereinander, durch ein Semikolon getrennt:

```
csc /define:RELEASE;DEMOVERSION mysymbols.cs
```

Die gleiche Definition lässt sich natürlich auch auf Ebene einer einzelnen C#-Quelldatei treffen. Dazu ergänzen Sie je Symbol eine `#define`-Direktive in einer neuen Zeile.

Falls ein über den Compiler definiertes Symbol in einer bestimmten Quelldatei unbekannt bleiben soll, ergänzen Sie für das Symbol eine `#undef`-Direktive:

```
#undef DEBUG
```

Für `#undef`-Direktiven gelten dieselben Regeln wie für `#define`-Direktiven: Ihre Wirkung bleibt auf die jeweilige Quelldatei beschränkt, und sie müssen *vor* der ersten Anweisung in einer eigenen Zeile stehen.

Viel mehr gibt es über die Symboldefinition mit dem C#-Präprozessor nicht zu sagen. Die folgenden Abschnitte zeigen, wie man Symbole für die bedingte Übersetzung von Code einsetzt.

Code mit Hilfe von Symbolen ein- und ausblenden

Die Definition von Symbolen dient in erster Linie dem Zweck, Teile des Codes bedingt ein- und ausblenden zu können. Das folgende Listing zeigt die Variante eines bereits in Kapitel 5 vorgestellten Programms, das in Abhängigkeit von dem Symbol `CALC_W_OUT_PARAM` bedingt kompiliert wird:

Listing 9.1: Mit der *#if*-Directive bedingt kompilieren

```
 1: using System;
 2:
 3: public class SquareSample
 4: {
 5:   public void CalcSquare(int nSideLength, out int nSquared)
 6:   {
 7:     nSquared = nSideLength * nSideLength;
 8:   }
 9:
10:   public int CalcSquare(int nSideLength)
11:   {
12:     return nSideLength * nSideLength;
13:   }
14: }
15:
16: class SquareApp
17: {
18:   public static void Main()
19:   {
20:     SquareSample sq = new SquareSample();
21:
22:     int nSquared = 0;
23:
24: #if CALC_W_OUT_PARAM
25:     sq.CalcSquare(20, out nSquared);
26: #else
27:     nSquared = sq.CalcSquare(15);
28: #endif
29:     Console.WriteLine(nSquared.ToString());
30:   }
31: }
```

Nachdem das Symbol `CALC_W_OUT_PARAM` im Quelltext nicht definiert ist, muss es – je nach gewünschter Version – über den Compiler definiert werden (oder eben nicht):

```
csc /define:CALC_W_OUT_PARAM squaresample.cs
```

Ist das Symbol bekannt, übersetzt der Compiler den Aufruf der Methode `CalcSquare` in Zeile 25, ansonsten den in Zeile 27. Die vom C#-Compiler emulierten Präprozessordirektiven `#if`, `#else` und `#endif` verhalten sich von der Logik her wie die `if`…`else`-Anweisung der Sprache, mit dem Unterschied, dass der Compiler den `false`-Zweig gar nicht erst zu sehen bekommt. Bei Bedarf können Sie auch die Operatoren `&&` und `||` sowie `!` auf Präprozessorebene einsetzen. Das folgende Listing zeigt ein Beispiel:

Listing 9.2: Erweiterte Fallunterscheidung auf Präprozessorebene mit der *#elif*-Direktive

```
 1: // #define DEBUG
 2: #define RELEASE
 3: #define DEMOVERSION
 4:
 5: #if DEBUG
 6:   #undef DEMOVERSION
 7: #endif
 8:
 9: using System;
10:
11: class Demo
12: {
13:   public static void Main()
14:   {
15: #if DEBUG
16:     Console.WriteLine("Debug-Version");
17: #elif RELEASE && !DEMOVERSION
18:     Console.WriteLine("Vollversion");
19: #else
20:     Console.WriteLine("Demoversion");
21: #endif
22:   }
23: }
```

In diesem Beispiel sind alle Symboldefinitionen im Quelltext enthalten. Achten Sie auf die bedingte `#undef`-Direktive in Zeile 6. Da der Fall »Debug-Version für Demo-Version« als eigenständiger Übersetzungslauf nicht vorgesehen ist, schließt ihn die Präprozessorlogik aus – und zwar so, dass er sich auch dann nicht über die Hintertür einschleichen kann, wenn der Compiler das Symbol `DEBUG` definiert.

Die Fallunterscheidung des Präprozessors findet in den Zeilen 15 bis 21 statt. Um mehr als zwei Fälle unterscheiden zu können, empfiehlt sich der Einsatz der `#elif`-Direktive sowie des `&&`-Operators. An Operatoren versteht der Präprozessor neben den bereits genannten logischen Operatoren auch die Vergleichsoperatoren `==` und `!=`.

Fehlermeldungen und Warnungen generieren

Weitere Präprozessordirektiven sind `#warning` und `#error`. Wie zu erwarten, veranlassen diese Direktiven den Compiler zur Ausgabe einer Warnung bzw. Fehlermeldung mit dem auf die Direktive folgenden Text. Es versteht sich, dass die Ausgabe solcher Meldungen im Allgemeinen bedingt erfolgen wird. Listing 9.3 zeigt den Einsatz der beiden Direktiven:

Listing 9.3: Compilerwarnungen und Fehlermeldungen mittels Präprozessordirektiven ausgeben

```
 1: #define DEBUG
 2: #define RELEASE
 3: #define DEMOVERSION
 4:
 5: #if DEMOVERSION && !DEBUG
 6:   #warning Kompiliere Demoversion
 7: #endif
 8:
 9: #if DEBUG && DEMOVERSION
10:   #error Die Debugversion einer Demoversion ist nicht vorgesehen
11: #endif
12:
13: using System;
14:
15: class Demo
16: {
17:   public static void Main()
18:   {
19:     Console.WriteLine("Demoanwendung");
20:   }
21: }
```

Bei der Übersetzung dieses Programms generiert der Compiler regulär eine Warnung, wenn das Symbol `DEMOVERSION` definiert ist und das Symbol `DEBUG` nicht (Zeilen 5 bis 7). Falls beide Symbole definiert sind, resultiert eine Fehlermeldung, die gleichzeitig verhindert, dass der Compiler ausführbaren Code generiert. Verglichen mit dem vorigen Beispiel, das die Definition des nicht erwarteten Symbols `DEBUG` kommentarlos ungeschehen macht, informiert Sie dieser Code über die Situation. Das ist zweifellos der bessere Ansatz.

9.1.2 conditional-Attribut

Die wichtigste Aufgabe des Präprozessors von C++ ist wahrscheinlich die Definition von Makros, die in Abhängigkeit von Symboldefinitionen in bestimmten Übersetzungsläufen Code einflechten, in anderen nicht. Beispiele dafür sind etwa die Makros `ASSERT` und `TRACE`, die zu Funktionsaufrufen werden, wenn das Symbol `DEBUG` definiert ist, und zu Nichts evaluieren, wenn es um die Übersetzung von Release-Versionen geht.

Aus der Tatsache, dass der C#-Präprozessor keine Makros unterstützt, werden Sie vielleicht folgern, dass es in C# generell keine bedingte Funktionalität in dieser Art gibt. Erfreulicherweise ist dem aber nicht so. C# unterstützt ein `conditional`-Attribut für die Deklaration von Methoden, das genau den gewünschten Effekt hat:

```
[conditional("DEBUG")]
  public void SomeMethod() { }
```

Die Methode `SomeMethod` ist für den Compiler nur dann im Code sichtbar, wenn das Symbol `DEBUG` definiert ist. Das gilt natürlich nicht nur für ihre Definition, sondern insbesondere auch für alle Aufrufe. Mit anderen Worten, der Compiler übergeht Aufrufe wie

```
SomeMethod();
```

wenn die Methode mit dem `conditional`-Attribut für ein bestimmtes Symbol deklariert wurde und das Symbol nicht bekannt ist. Wie man sich leicht überlegt, kann die Eliminierung von Funktionsaufrufen aus dem Code nur dann ohne weitere Schwierigkeiten geschehen, wenn für die Methode der Ergebnistyp `void` vereinbart wurde – und genau dies ist Voraussetzung für das `conditional`-Attribut. Hinsichtlich der Parameter gibt es jedoch keine Einschränkungen.

Das folgende Listing zeigt, wie sich das `conditional`-Attribut einsetzen lässt, um die Funktionalität des aus C++ bekannten `ASSERT`-Makros in C# nachzubilden. Der Einfachheit halber geht die Ausgabe an die Konsole – man könnte sie ohne weiteres aber auch woandershin dirigieren, beispielsweise in eine Datei.

Listing 9.4: Implementation der TRACE-Funktionalität mit dem *conditional*-Attribut

```
 1: #define DEBUG
 2:
 3: using System;
 4:
 5: class Info
 6: {
 7:   [conditional("DEBUG")]
 8:   public static void Trace(string strMessage)
 9:   {
10:     Console.WriteLine(strMessage);
11:   }
12:
13:   [conditional("DEBUG")]
14:   public static void TraceX(string strFormat,params object[] list)
15:   {
16:     Console.WriteLine(strFormat, list);
17:   }
18: }
19:
20: class TestConditional
21: {
22:   public static void Main()
23:   {
```

```
24:     Info.Trace("Cool!");
25:     Info.TraceX("{0} {1} {2}", "C", "U", 2001);
26:   }
27: }
```

Die Klasse `Info` enthält zwei bedingte, auf das `DEBUG`-Symbol zugeschnittene Vereinbarungen für statische Methoden: `Trace`, eine einparametrige Methode, und `TraceX`, eine Methode, die mit einer flexiblen Anzahl von Parametern aufgerufen werden kann. Über `Trace` gibt es weiter nichts zu sagen. Die Deklaration von `TraceX` enthält hingegen den Zusatz `params`, den Sie vielleicht von Visual Basic her kennen werden.

Indem Sie das Schlüsselwort `params` vor einen Arraybezeichner setzen, kann die Methode eine beliebige Anzahl von Argumenten verarbeiten – vergleichbar mit der Bedeutung der Ellipse »…« in C++. Zu beachten ist lediglich, dass Sie `params` nur einmal und auch nur für den letzten Parameter in der Parameterliste einer Methode verwenden dürfen. Beide Beschränkungen dürften sich aber von selbst verstehen.

Hinter der Verwendung des `params`-Schlüsselworts steckt natürlich die Absicht, eine `Trace`-Methode zu erhalten, deren Parameterschema, eine Formatzeichenfolge in Kombination mit einer offenen Anzahl an Parameterwerten, genau auf den Aufruf der `WriteLine`-Methode passt (Zeile 16).

Die Ausgabe des kleinen Programms wird ausschließlich durch das Symbol `DEBUG` bestimmt. Ist das Symbol bekannt, übersetzt der Compiler die beiden Methoden sowie ihre Aufrufe in der Methode `Main`, andernfalls nicht.

Bedingt definierte Methoden sind ein wirklich mächtiges Werkzeug, das es Ihnen erlaubt, Ihre Anwendungen und Komponenten auf sehr einfache Weise mit bedingter Funktionalität auszustatten. Mit ein paar zusätzlichen Schnörkeln können Sie auch bedingte Methoden vereinbaren, die von mehreren Symbolen – verbunden durch die logischen Operatoren `&&` und `||` – abhängig sind. Details darüber finden Sie in der C#-Dokumentation.

9.2 Kommentarbasierte Dokumentation in XML

Eine Aufgabe, die den wenigsten Programmierern wirklich Spaß macht, ist das ausführliche Kommentieren und Dokumentieren ihrer Quelltexte. In C# steckt das Potenzial, mit alten Gewohnheiten zu brechen: Sie können sich die Dokumentation automatisiert aus den Kommentaren in Ihren Quelltexten generieren lassen.

Da der Compiler die Ausgabe im XML-Format liefert, lässt sie sich sowohl als Eingabedatei für die weitergehende Dokumentation der Komponente verwenden als auch als Informationsquelle für Werkzeuge, die Hilfe und implementatorische Einblicke zu Ihren Komponenten verfügbar machen. Visual Studio 7 ist beispiels-

weise so ein Werkzeug. Gute Dokumentation kann nun endlich zu dem Verkaufsargument werden, das sie immer schon hätte sein sollen.

Dieser Abschnitt will Ihnen zeigen, wie Sie den Dokumentationsmechanismus von C# optimal für sich einsetzen. Das Beispielmaterial ist bewusst einfach und ausführlich gehalten, um Ihnen die Ausrede zu nehmen, der Abschnitt sei zu kompliziert gewesen, als dass Sie hätten herausfinden können, wie man in einem C#-Quelltext Kommentare für die Dokumentation verwendet. Dokumentation ist ein immens wichtiger Bestandteil von Software, speziell von Komponenten, die auch andere Entwickler verwenden können sollen.

In den folgenden Unterabschnitten lernen Sie am Beispiel der in Kapitel 8 vorgestellten Klasse `RequestWebPage` verschiedene Techniken kennen, wie Sie die einzelnen Bestandteile Ihrer Dokumentation als C#-Kommentare formulieren. Die entsprechenden Überschriften lauten:

- Elemente beschreiben
- Bemerkungen und Auflistungen einfügen
- Beispiele anführen
- Parameter beschreiben
- Eigenschaften beschreiben
- Kompilieren der Dokumenation

9.2.1 Elemente beschreiben

Im ersten Schritt fügen Sie die Beschreibung eines Klassenelements ein – und zwar mit einem <summary>-Tag:

```
/// <summary>Dieses Element fungiert als ... </summary>
```

Kommentare, die etwas zur Dokumentation beitragen, beginnen mit einem dreifachen Schrägstrich `///` und werden *vor* das beschriebene Element gesetzt:

```
/// <summary>Klasse fordert eine Webseite von einem Webserver an</summa-
ry>
public class RequestWebPage
```

Um einen Absatz in eine Beschreibung einzufügen, verwenden Sie das Tag-Paar <para> und </para>. Und das Tag <see> ermöglicht Verweise auf andere Elemente:

```
///   <para>Gehört der Klasse <see cref="RequestWebPage"/> an</para>
```

Dieser Kommentar enthält somit einen Verweis auf die Beschreibung der Klasse `RequestWebPage`. Beachten Sie, dass die Tags der XML-Syntax unterworfen sind, also die Unterscheidung zwischen Groß- und Kleinschreibung eine Rolle spielt.

Ein anderes interessantes Tag für Dokumentation von Elementen ist das `<seealso>`-Tag. Es gestattet Ihnen, Querverweise auf andere Themen einzufügen, die für den Leser interessant sein könnten:

```
/// <seealso cref="System.Net"/>
```

Der Querverweis in diesem Beispiel ermöglicht es dem Benutzer, in die Dokumentation für den Namensraum `System.Net` zu verzweigen. Sofern Sie auf ein Element verweisen, das im aktuellen Namensraum nicht sichtbar ist, müssen Sie den vollständig qualifizierten Bezeichner angegeben.

Das folgende Listing demonstriert wie angekündigt am Beispiel der Klasse `RequestWebPage`, wie die professionelle Dokumentation von Elementen konkret aussehen kann. Beachten Sie, wie die einzelnen XML-Tags angeordnet und verschachtelt sind:

Listing 9.5: Dokumentation von Elementen mit den Tags *<summary>*, *<see>*, *<para>* und *<seealso>*

```
 1: using System;
 2: using System.Net;
 3: using System.IO;
 4: using System.Text;
 5:
 6: /// <summary>Klasse fordert eine Webseite von einem Webserver an
       </summary>
 7: public class RequestWebPage
 8: {
 9:   private const int BUFFER_SIZE = 128;
10:
11:   /// <summary>m_strURL speichert den URL der Webseite</summary>
12:   private string m_strURL;
13:
14:   /// <summary>RequestWebPage() ist der Konstruktor der Klasse
15:   /// <see cref="RequestWebPage"/> für Aufruf ohne Argumente.
          </summary>
16:   public RequestWebPage()
17:   {
18:   }
19:
20:  /// <summary>RequestWebPage(string strURL) ist der Konstruktor der
         Klasse
21:  /// <see cref="RequestWebPage"/> bei Initialisierung mit URL.
         </summary>
```

```
22:   public RequestWebPage(string strURL)
23:   {
24:     m_strURL = strURL;
25:   }
26:
27:   public string URL
28:   {
29:     get { return m_strURL;  }
30:     set { m_strURL = value; }
31:   }
32:
33:   /// <summary>Methode GetContent(out string strContent):
34:   ///   <para>Ist Element der Klasse <see cref="RequestWebPage"/>
            </para>
35:   ///   <para>Verwendet die Variable <see cref="m_strURL"/></para>
36:   ///   <para>Fordert den Inhalt einer Webseite an. </para>
37:   ///   <para>Setzt voraus, dass URL (http://...) für Webseite
            zuvor in
38:   ///     der privaten Elementvariable m_strURL gespeichert wurde.
              Diese
39:   ///     Initialisierung kann bei Konstruktion des Objekts
              geschehen,
40:   ///     aber auch durch Setzen der Eigenschaft <see cref="URL"/>.
              </para>
41:   /// </summary>
42:   /// <seealso cref="System.Net"/>
43:   /// <seealso cref="System.Net.WebResponse"/>
44:   /// <seealso cref="System.Net.WebRequest"/>
45:   /// <seealso cref="System.Net.WebRequestFactory"/>
46:   /// <seealso cref="System.IO.Stream"/>
47:   /// <seealso cref="System.Text.StringBuilder"/>
48:   /// <seealso cref="System.ArgumentException"/>
49:
50:   public bool GetContent(out string strContent)
51:   {
52:     strContent = "";
53:     // ...
54:     return true;
55:   }
56: }
```

9.2.2 Bemerkungen und Auflistungen einfügen

Das Tag `<summary>` sollte nur die essenzielle Information zu einem Element zusammenfassen. Für den ausführlichen Teil der Dokumentation, sonstige Prosa, Hinweise und Bemerkungen steht das Tag `<remarks>` zur Verfügung.

Ferner sind Sie bei der Absatzformatierung nicht allein auf das <para>-Tag beschränkt. So können Sie in einem <remarks>-Abschnitt beispielsweise auch Auflistungen (mit Auflistungszeichen oder auch nummeriert) nach folgendem Muster einfügen

```
///     <list type="bullet">
///       <item>Konstruktoren
///         <see cref="RequestWebPage()"/> und
///         <see cref="RequestWebPage(string)"/>
///        </item>
///     </list>
```

Beachten Sie, dass diese Auflistung aus nur einem Element besteht, aber zwei Querverweise enthält – nämlich auf die Beschreibungen der beiden Konstruktoren der Klasse. Natürlich kann eine Auflistung beliebig viele Elemente enthalten und auch die Inhalte der einzelnen Auflistungselemente lassen sich nach Belieben gestalten.

Ein weiteres Tag, das sich ganz gut in <remarks>-Abschnitten macht, ist <paramref>. Sie verwenden es, um im Rahmen einer Beschreibung auf die Deklaration eines Parameters zu verweisen. Die Dokumentation für den Parameter des zweiten Konstruktors könnte beispielsweise so aussehen:

```
/// <remarks>Überträgt den im Parameter <paramref name="strURL"/>
/// übergebenen URL in Elementvariable <see cref="m_strURL"/>.
/// </remarks>
public RequestWebPage(string strURL)
```

Das folgende Listing zeigt all diese Tags – zusammen mit den bereits vorgestellten – in Aktion:

Listing 9.6: Bemerkungen und Auflistungen in die Dokumentation einfügen

```
 1: using System;
 2: using System.Net;
 3: using System.IO;
 4: using System.Text;
 5:
 6: /// <summary>Klasse fordert eine Webseite von einem Webserver an
        </summary>
 7: /// <remarks>Die Klasse RequestWebPage hat die Elemente:
 8: ///   <para>Methoden:
 9: ///     <list type="bullet">
10: ///       <item>Konstruktoren
11: ///         <see cref="RequestWebPage()"/> und
12: ///         <see cref="RequestWebPage(string)"/>
13: ///        </item>
14: ///     </list>
```

```
15: ///   </para>
16: ///   <para>Eigenschaften:
17: ///       <list type="bullet">
18: ///       <item>
19: ///         <see cref="URL"/>
20: ///        </item>
21: ///      </list>
22: ///   </para>
23: /// </remarks>
24: public class RequestWebPage
25: {
26:   private const int BUFFER_SIZE = 128;
27:
28:   /// <summary> m_strURL speichert den URL der Webseite</summary>
29:   private string m_strURL;
30:
31:   /// <summary>RequestWebPage() ist der Konstruktor der Klasse
32:   /// <see cref="RequestWebPage"/> für Aufruf ohne Argumente.
          </summary>
33:   public RequestWebPage()
34:   {
35:   }
36:
37: /// <summary>RequestWebPage(string strURL) ist der Konstruktor der
        Klasse
38: /// <see cref="RequestWebPage"/> bei Initialisierung mit URL.
        </summary>
39: /// <remarks>Überträgt den im Parameter <paramref name="strURL"/>
40: /// übergebenen URL in Elementvariable <see cref="m_strURL"/>.
        </remarks>
41:   public RequestWebPage(string strURL)
42:   {
43:     m_strURL = strURL;
44:   }
45:
46:   /// <remarks>Setzt den Wert von <see cref="m_strURL"/>.
47:   ///       Liefert den Wert von <see cref="m_strURL"/>.</remarks>
48:   public string URL
49:   {
50:     get { return m_strURL;  }
51:     set { m_strURL = value; }
52:   }
53:
54:   /// <summary>Methode GetContent(out string strContent):
55:   ///   <para>Ist Element der Klasse <see cref="RequestWebPage"/>
            </para>
56:   ///   <para>Verwendet die Variable <see cref="m_strURL"/></para>
```

```
57:   ///   <para>Fordert den Inhalt einer Webseite an. </para>
58:   ///   <para>Setzt voraus, dass URL (http://...) für Webseite
            zuvor in
59:   ///     der privaten Elementvariable m_strURL gespeichert wurde.
              Diese
60:   ///     Initialisierung kann bei Konstruktion des Objekts gesche-
hen,
61:   ///     aber auch durch Setzen der Eigenschaft <see cref="URL"/>.
              </para>
62:   /// </summary>
63:   /// <remarks>Fordert den Inhalt der in der Eigenschaft
          <see cref="URL"/>
64:   ///    spezifizierten Webseite an und gibt diesen im Ausgabe-
             parameter
65:   ///    <paramref name="strContent"/> zurück.
66:   /// Die Implementation der Methode benutzt die Elemente:
67:   /// <list>
68:   /// <item><see cref="System.Net.WebRequestFactory.Create"/>.
          </item>
69:   /// <item><see cref="System.Net.WebRequest.GetResponse"/></item>
70:   /// <item><see cref="System.Net.WebResponse.GetResponseStream"/>
          </item>
71:   /// <item><see cref="System.IO.Stream.Read"/></item>
72:   /// <item><see cref="System.Text.StringBuilder.Append"/></item>
73:   /// <item><see cref="System.Text.Encoding.ASCII"/> und dessen
          Methode
74:   /// <see cref="System.Text.Encoding.ASCII.GetString"/></item>
75:   /// <item><see cref="System.Object.ToString"/> für das Objekt
76:   /// <see cref="System.IO.Stream"/>.</item>
77:   ///    </list>
78:   ///  </remarks>
79:   /// <seealso cref="System.Net"/>
80:   public bool GetContent(out string strContent)
81:   {
82:     strContent = "";
83:     // ...
84:     return true;
85:   }
86: }
```

9.2.3 Beispiele anführen

Es gibt keinen besseren Weg, den Gebrauch eines Objekts oder einer Methode zu dokumentieren, als über prototypisches Beispiel. Somit dürfte es nicht verwundern, dass Sie für die Dokumentation auch die Tags `<example>` und `<code>` ver-

wenden können. Das <example>-Tag umschließt das gesamte Beispiel – also Beschreibung und Code –, das <code>-Tag (welch Überraschung) nur den Code des Beispiels.

Das folgende Listing demonstriert anhand der Dokumentation für die Konstruktoren, wie man Codebeispiele in die Dokumentation einfügt. Das Beispiel für den Aufruf der Methode `GetContent` können Sie dann selbst ergänzen.

Listing 9.7: Konzepte mit Beispielcode dokumentieren

```
using System;
using System.Net;
using System.IO;
using System.Text;

/// <summary>Klasse fordert eine Webseite von einem Webserver an
    </summary>
/// <remarks> ... </remarks>
public class RequestWebPage
{
 private const int BUFFER_SIZE = 128;

 /// <summary> m_strURL speichert den URL der Webseite</summary>
 private string m_strURL;

 /// <summary>RequestWebPage() ist der Konstruktor ... </summary>
 /// <example>Dieses Beispiel demonstriert den Aufruf des parameter-
     losen
 /// Konstruktors der Klasse:
 ///  <code>
 ///   public class MyClass
 ///   {
 ///    public static void Main()
 ///    {
 ///     public
 ///     string strContent;
 ///     RequestWebPage objRWP = new RequestWebPage();
 ///     objRWP.URL = "http://www.alphasierrapapa.com";
 ///     objRWP.GetContent(out strContent);
 ///     Console.WriteLine(strContent);
 ///    }
 ///   }
 ///  </code>
 /// </example>
 public RequestWebPage()
 {
 }

```

```
37:  /// <summary>RequestWebPage(string strURL) ist ... </summary>
38:  /// <remarks> ... </remarks>
39:  /// <example>Dieses Beispiel zeigt den Aufruf des zweiten Konstruk-
         tors
40:  ///  der Klasse, der gleichzeitig die Eigenschaft URL initia-
         lisiert:
41:  ///  <code>
42:  ///   public class MyClass
43:  ///   {
44:  ///    public static void Main()
45:  ///    {
46:  ///     string strContent;
47:  ///     RequestWebPage objRWP = new
     ///       RequestWebPage("http://www.alphasierrapapa.com");
48:  ///     objRWP.GetContent(out strContent);
49:  ///  Console.WriteLine("\n\nInhalt der Webseite "+ objRWP.
         URL+":\n\n");
50:  ///     Console.WriteLine(strContent);
51:  ///    }
52:  ///   }
53:  ///  </code>
54:  /// </example>
55:  public RequestWebPage(string strURL)
56:  {
57:   m_strURL = strURL;
58:  }
59:
60:  /// <remarks> ... </remarks>
61:  public string URL
62:  {
63:   get { return m_strURL;  }
64:   set { m_strURL = value; }
65:  }
66:
67:  /// <summary> Methode GetContent(out string strContent): ...
         </summary>
68:  /// <remarks> ... </remarks>
69:  /// <seealso cref="System.Net"/>
70:  public bool GetContent(out string strContent)
71:  {
72:   strContent = "";
73:   // ...
74:   return true;
75:  }
76: }
```

9.2.4 Parameter beschreiben

Eine weitere sehr wichtige Aufgabe der Dokumentation, die bisher noch keine Erwähnung fand, ist die Beschreibung der Parameter von Konstruktoren und Methoden. Die Umsetzung ist geradlinig. Sie verwenden einfach das <param>-Tag nach folgendem Muster:

```
/// <param name="strURL">
///  Wird dazu verwendet, die Eigenschaft URL gleich bei Konstruktion
///  eines Objekts auf den URL der Webseite zu setzen. Der Wert wird in
///  der Elementvariable <see cref="m_strURL"/> gespeichert.</param>
```

Das war die Beschreibung für einen einfachen Eingabeparameter. Beachten Sie, dass einem <param>-Tag auch <para>-Tags untergeordnet sein können.

Ein Rückgabeparameter wird etwas anders beschrieben.

```
/// <returns>
///  <para>true: Webseite erfolgreich gelesen. </para>
///  <para>false: Webseite nicht gelesen. </para>
/// </returns>
```

Wie Sie sehen, geschieht die Beschreibung von Rückgabeparametern mit <returns>-Tags. Das vollständige Beispiel zeigt Listing 9.8

Listing 9.8: Parameter von Methoden und Ergebniswerte beschreiben

```
 1: using System;
 2: using System.Net;
 3: using System.IO;
 4: using System.Text;
 5:
 6: /// <summary>Klasse fordert eine Webseite von einem Webserver an
        </summary>
 7: /// <remarks> ... </remarks>
 8: public class RequestWebPage
 9: {
10:  private const int BUFFER_SIZE = 128;
11:
12:  ///  <summary> m_strURL speichert den URL der Webseite</summary>
13:  private string m_strURL;
14:
15:  /// <summary>RequestWebPage() ist der Konstruktor ... </summary>
16:  /// <example>Dieses Beispiel demonstriert ...
17:  ///  <code>
18:  ///   public class MyClass
19:  ///   {
20:  ///    ...
```

```
21:  ///   }
22:  ///  </code>
23:  /// </example>
24:  public RequestWebPage()
25:  {
26:  }
27:
28:  /// <summary>RequestWebPage(string strURL) ist ... </summary>
29:  /// <remarks> ... </remarks>
30:  /// <param name="strURL">
31:  ///  Wird dazu verwendet, die Eigenschaft URL gleich bei Konstruk-
          tion
32:  ///  eines Objekts auf den URL der Webseite zu setzen. Der Wert
          wird in
33:  ///  der Elementvariable <see cref="m_strURL"/> gespeichert.
          </param>
34:  /// <example> ... </example>
35:  public RequestWebPage(string strURL)
36:  {
37:   m_strURL = strURL;
38:  }
39:
40:  /// <remarks> ... </remarks>
41:  public string URL
42:  {
43:   get { return m_strURL;  }
44:   set { m_strURL = value; }
45:  }
46:
47:  /// <summary> Methode GetContent(out string strContent): ...
         </summary>
48:  /// <remarks> Fordert den Inhalt der in der Eigenschaft
         <see cref="URL"/>
49:  ///   spezifizierten Webseite an und gibt diesen im Ausgabepara-
           meter
50:  ///   <paramref name="strContent"/> zurück.
51:  /// Die Implementation der Methode benutzt die Elemente: ...
52:  /// </remarks>
53:  /// <param name="strContent">Enthält den Inhalt der Webseite.
         </param>
54:  /// <returns>
55:  ///  <para>true: Webseite erfolgreich gelesen. </para>
56:  ///  <para>false: Webseite nicht gelesen. </para>
57:  /// </returns>
58:  /// <seealso cref="System.Net"/>
59:  public bool GetContent(out string strContent)
60:  {
```

```
61:   strContent = "";
62:   // ...
63:     return true;
64:  }
65: }
```

9.2.5 Eigenschaften beschreiben

Zur Beschreibung der Eigenschaften einer Klasse benutzen Sie anstelle des <summary>-Tags das speziell für diesen Elementtyp zuständige <value>-Tag.

Listing 9.9 demonstriert den Einsatz dieses Tags für die Eigenschaft `URL` der Klasse `RequestWebPage` (ab Zeile 30). Darüber hinaus gibt das Listing nun auch einen guten Überblick über die gesamte Struktur der kommentarbasierten Dokumentation für die Komponente `RequestWebPage` (auch wenn die einzelnen Teile aus Platzgründen fehlen):

Listing 9.9: Eigenschaften mit dem <*value*>-Tag dokumentieren

```
 1: using System;
 2: using System.Net;
 3: using System.IO;
 4: using System.Text;
 5:
 6: /// <summary>Klasse fordert eine Webseite von einem Webserver an
        </summary>
 7: /// <remarks> ... </remarks>
 8: public class RequestWebPage
 9: {
10:  private const int BUFFER_SIZE = 128;
11:
12:  ///  <summary> m_strURL speichert den URL der Webseite</summary>
13:  private string m_strURL;
14:
15:  /// <summary>RequestWebPage() ist der Konstruktor ... </summary>
16:  /// <example> ... </example>
17:  public RequestWebPage()
18:  {
19:  }
20:
21:  /// <summary>RequestWebPage(string strURL) ist ... </summary>
22:  /// <remarks> ... </remarks>
23:  /// <param name="strURL"> ... </param>
24:  /// <example>Dieses Beispiel ... </example>
25:  public RequestWebPage(string strURL)
26:  {
27:   m_strURL = strURL;
28:  }
```

```
29:
30:  /// <value>Die Eigenschaft URL liefert/setzt den URL der Webseite
31:  /// </value><remarks>Setzt den Wert von <see cref="m_strURL"/>.
32:  ///    Liefert den Wert von <see cref="m_strURL"/>.</remarks>
33:  public string URL
34:  {
35:   get { return m_strURL;  }
36:   set { m_strURL = value; }
37:  }
38:
39: /// <summary>Methode GetContent(out string strContent): ...
       </summary>
40: /// <remarks>Fordert den Inhalt der in der Eigenschaft
       <see cref="URL"/>
41:  /// spezifizierten Webseite an und gibt diesen im Ausgabeparameter
42:  ///  <paramref name="strContent"/> zurück.
43:  /// Die Implementation der Methode benutzt die Elemente: ...
44:  /// </remarks>
45:  /// <param name="strContent">Enthält den Inhalt der Webseite.
        </param>
46:  /// <returns>
47:  ///  <para>true: Webseite erfolgreich gelesen. </para>
48:  ///  <para> false: Webseite nicht gelesen. </para>
49:  /// </returns>
50:  /// <seealso cref="System.Net"/>
51:  public bool GetContent(out string strContent)
52:  {
53:   strContent = "";
54:   // ...
55:   return true;
56:  }
57: }
```

9.2.6 Kompilieren der Dokumentation

Die Dokumentation Ihrer Komponente umfasst nun eine ausführliche Erläuterung aller Konstruktoren, Methoden und deren Parameter sowie der Eigenschaften, Elementvariablen usw. Damit wäre sie vollständig und kann in eine XML-Datei kompiliert werden, welche Sie schließlich Ihren Kunden als Informationsquelle über das Innenleben Ihrer Komponente überlassen können. Damit der Compiler im Rahmen eines gewöhnlichen Übersetzungslaufs die XML-Datei aus den entsprechenden Kommentaren im Quellcode zusammenstellt, ist nichts weiter als die zusätzliche Angabe des Schalters `/doc:` zusammen mit dem Namen der zu erstellenden XML-Datei erforderlich.

```
csc /r:System.Net.dll /doc:wrq.xml /t:library /out:wrq.dll requestweb-
page.cs
```

Vorausgesetzt, die Dokumentation enthält keine Fehler (ja, sie wird auf ihre Gültigkeit hin überprüft), generiert der Compiler bei diesem Übersetzungslauf unter anderem die Datei `wrq.xml`, in der schließlich die gesamte Dokumentation Ihrer Komponente enthalten ist.

Sie können sich das Ergebnis nun zwar wiederum als Listing ansehen, interessanter dürfte es aber sein, wenn Sie die Datei im Internet Explorer betrachten. Wie in Abbildung 9.1 gezeigt, gibt der Internet Explorer die hierarchische Struktur der im Quelltext befindlichen Dokumentation wieder und gestattet es auch, sie interaktiv zu durchforsten.

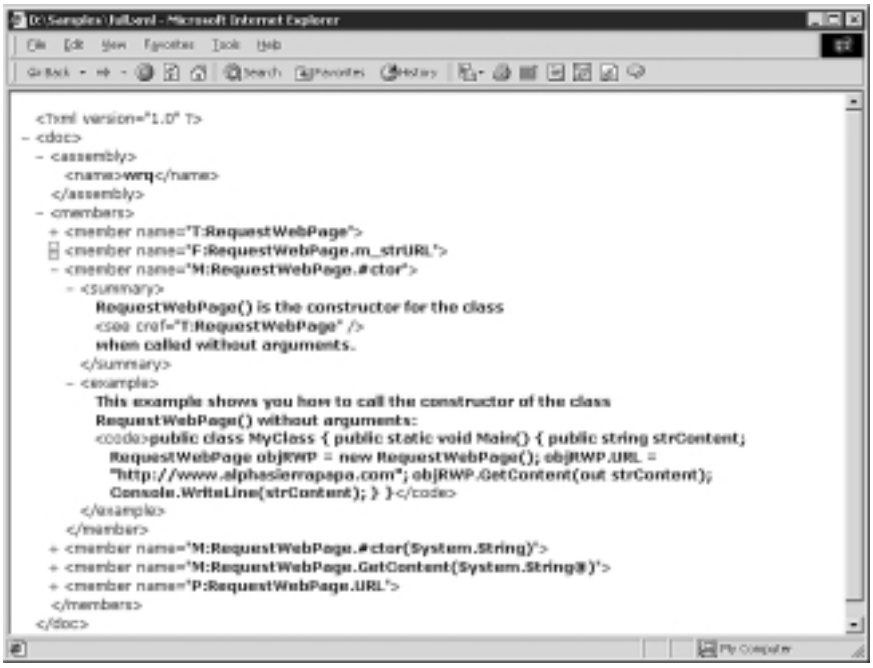


Bild 9.1: Betrachten der ins XML-Format übersetzten Dokumentation im Internet Explorer

Ohne dass es viel Sinn machen würde, an dieser Stelle tiefer in die Semantik der XML-Datei einzusteigen, sollten Sie aber dennoch wissen, wie das `name`-Attribut eines `member`-Tags zu lesen ist. Der erste Teil des Attributs, der Buchstabe vor dem Doppelpunkt, gibt Auskunft über die Art des Elements, der zweite ist der im Quellcode verwendete Bezeichner:

- `N` – Bezeichner steht für einen Namensraum
- `T` – Bezeichner steht für einen Datentyp (Klasse, Schnittstelle, Strukturtyp, Aufzählungstyp oder Delegation)

- `F` – Bezeichner steht für eine Elementvariable einer Klasse
- `P` – Bezeichner steht für eine Eigenschaft (kann auch Indizierer oder indizierte Eigenschaft sein)
- `M` – Bezeichner steht für eine Methode (Operatoren, Konstruktoren, Destrukturen inbegriffen)
- `E` – Bezeichner steht für ein Ereignis
- `!` – Bezeichner steht für eine Fehlermeldung über einen Verweis, den der C#-Compiler nicht auflösen konnte

Alle Bezeichner sind vollständig qualifiziert, das heißt, sie enthalten alle Namensanteile und Typzusätze bis hin zur Wurzel des Namensraums. Der Konstruktor ist durch `#ctor` und der Destruktor durch `Finalize` notiert. Die Parameter einer Methode folgen in Klammern auf den Bezeichner, wie üblich durch Kommas getrennt. Jeder einzelne Parameter ist als reine Typspezifikation notiert (NGWS-Signatur, wenn der Typ dem NGWS-Subsystem entstammt). Die Typspezifikation eines Ausgabeparameters trägt das Suffix `@`.

Unter normalen Umständen müssen Sie sich jedoch nicht weiter um die Details der XML-Dokumentation kümmern. Erzeugen Sie einfach die Datei und packen Sie sie Ihrer Komponente bei. Wer Programmierwerkzeuge benutzt, wird dann zweifelsohne seine Freude mit Ihrer Software haben.

9.3 Versionspflege

Die Versionspflege gerät inzwischen mehr und mehr zum Balanceakt über einer sich ständig ändernden Landschaft von DLLs. Einzelne Anwendungen installieren immer wieder neue Versionen gemeinsam genutzter Komponenten, bis plötzlich die eine oder andere Anwendung ihren Dienst versagt, weil sie mit der aktuell installierten Version einer Komponente nicht mehr zurechtkommt. In der Tat bereiten gemeinsam genutzte Komponenten heutzutage mehr Probleme als sie lösen.

Eines der primären Ziele der NGWS-Laufzeitumgebung ist die Beseitigung leidiger Versionskonflikte. Zentrales Moment in dem Lösungsansatz ist das Konzept der NGWS-Komponenten (wobei hier die Verpackung und nicht der Inhalt im Vordergrund steht), das es einem Entwickler unter anderem ermöglicht, Regeln für die Versionsabhängigkeiten zwischen den verschiedenen Teilen seiner Software zu spezifizieren – wobei das NGWS-Subsystem diese Regeln zur Laufzeit durchsetzt.

Dazu erst einmal ein wenig mehr über NGWS-Komponenten, damit Sie eine Vorstellung davon erhalten, wofür man sie einsetzt und wie sie sich – mit Blick auf die Versionspflege – von DLLs unterscheiden.

9.3.1 NGWS-Komponenten

Obwohl es an der entsprechenden Stelle in Kapitel 8 nicht eigens erwähnt ist: Die erste Bibliothek, die Sie kompiliert haben, war nichts anderes als eine NGWS-Komponente, aus dem schlichten Grund, weil der C#-Compiler Komponenten grundsätzlich als NGWS-Komponenten übersetzt. Was also ist eine NGWS-Komponente?

Die *NGWS-Komponente* ist in erster Linie die grundlegende Einheit für die gemeinsame Nutzung von Code im Rahmen der NGWS-Laufzeitumgebung. Wie nicht anderes zu erwarten, bildet sie auch die Grenze für die Versionskontrolle, die Sicherheitsprüfung, die Klassengliederung und die Typauflösung. C#-Anwendungen setzen sich somit typischerweise aus mehreren NGWS-Komponenten zusammen.

Nachdem es in diesem Abschnitt um die Versionsverwaltung geht, stellt sich die Frage »Wie ist die Versionsinformation überhaupt beschaffen?«. Die Versionsnummer einer NGWS-Komponente setzt sich allgemein aus vier Teilen zusammen:

```
MajorVersion.MinorVersion.BuildNumber.Revision
```

Diese Versionsnummer heißt *Kompatibiliätsversion* und entscheidet darüber, welche Version einer NGWS-Komponente die Laderoutine des Laufzeitsystems bei Anforderung der Komponente auswählt. Eine Version wird als inkompatibel eingestuft, wenn sie in den Bestandteilen *MajorVersion* und *MinorVersion* nicht mit der angeforderten Version übereinstimmt. Falls dagegen der Bestandteil *BuildNumber* abweicht, gilt die Komponente noch als kompatibel. Vollständige Kompatibilität liegt vor, wenn Abweichungen auf den Bestandteil *Revision* beschränkt bleiben – unterschiedliche Revisionsnummern resultieren aus dem sogenannten *QuickFix-Engineering*, das sich auf die reine Fehlerbeseitigung beschränkt.

Darüber hinaus enthält eine NGWS-Komponente noch eine zweite Versionsinformation, die sogenannte *informale Version*. Wie der Name bereits vermuten lässt, dient diese Information – die etwa nach dem Muster `MeinSuperControl Build 1890` gestrickt ist – nur dem Zweck der Dokumentation und bleibt für die maschinelle Versionsprüfung ohne Bedeutung.

Bevor es in den folgenden beiden Abschnitten um private und gemeinsam genutzte NGWS-Komponenten geht, sollten Sie noch den Compilerschalter `/a.version` zum Setzen der Kompatibilitätsversion kennenlernen. Wenn Sie der Komponente `RequestWebpage` beispielsweise die Versionsinformation `1.0.1.0` zuordnen wollen, lautet der Compileraufruf:

```
csc /a.version:1.0.1.0 /t:library /out:wrq.dll requestwebpage.cs
```

Sie überprüfen die Versionsnummer, indem Sie den *Eigenschaften*-Dialog der resultierenden Datei `wrq.dll` über ein Explorer-Fenster öffnen und die Registerkarte *Version* anzeigen.

Private NGWS-Komponenten

Wenn Sie eine Anwendung (durch Angabe des Compilerschalters `/reference:`) mit einer Komponente verknüpfen, hinterlegt der Compiler im Ladeverzeichnis der ausführbaren Datei bzw. Bibliothek eine *Abhängigkeitsinformation*, die insbesondere auch die Versionsnummern der verknüpften Bibliotheken umfasst. Diese Abhängigkeitsinformation wird beim Start der Anwendung ausgewertet, um dann zur Laufzeit die richtige Version der NGWS-Komponente bereitstellen zu können.

Falls Sie nun der Meinung sind, dass auch die als Beispiel diskutierte NGWS-Komponente `wrq.dll` der Versionsprüfung unterliegt, befinden Sie sich im Irrtum: NGWS-Komponenten, die im Ausführungspfad der Anwendung zu finden sind, betrachtet das Laufzeitsystem als *private Komponenten* und unterwirft sie daher keiner Versionsprüfung. Der Grund dafür ist einleuchtend: Was Sie zusammen mit der ausführbaren Datei Ihrer Anwendung in ein Verzeichnis packen, bleibt ebenso Ihnen überlassen, wie die Sicherstellung der Kompatibilität unter den gelieferten Dateien.

Nachteile sind bei der Verwendung privater NGWS-Komponenten keine zu erwarten. Denn selbst wenn eine andere Anwendung die gleiche Komponente in einer anderen Version für die gemeinsame Nutzung installieren sollte, wird Ihre Anwendung von dieser keinen Gebrauch machen, da sie unter Angabe einer privaten NGWS-Komponente übersetzt wurde. Und den zusätzlichen Speicherplatzbedarf dürften die ausbleibenden Versionskonflikte mehr als aufwiegen.

Gemeinsam genutzte NGWS-Komponenten

Wenn Sie Software für die gemeinsame Nutzung durch mehrere Anwendungen schreiben wollen, bieten sich gemeinsam genutzte NGWS-Komponenten an. Allerdings müssen Sie dazu eine Reihe zusätzlicher Vorkehrungen treffen.

Zunächst einmal müssen Sie sich einen eindeutigen Namen für die Komponente aussuchen. Vielleicht werden Sie sich schon gefragt haben, wann endlich das Gegenstück der aus dem COM bekannten GUID (global einzigartiger Bezeichner) ins Spiel kommt – hier ist es. Solange Sie mit privaten NGWS-Komponenten hantieren, sind keine global eindeutigen Bezeichner erforderlich; erst wenn Sie eine NGWS-Komponente für die gemeinsame Nutzung installieren wollen, benötigen Sie einen eindeutigen Namen.

Die Eindeutigkeit des Namens wird durch ein kryptografisches Standardverfahren auf Basis eines öffentlichen Schlüssels garantiert: Sie als Entwickler signieren Ihre NGWS-Komponente mit einem privaten Schlüssel, während die Anwendung

einen öffentlichen Schlüssel benutzt, um die Authentizität des Urhebers der NGWS-Komponente zu prüfen. Nachdem Sie Ihre NGWS-Komponente signiert haben, können Sie sie in den allgemeinen Cache für NGWS-Komponenten oder in das Verzeichnis Ihrer Anwendung kopieren. Alles Weitere erledigt das Laufzeitsystem: Es kümmert sich darum, dass die Komponente fortan allen Anwendungen zur Verfügung steht.

Zahlt sich die gemeinsame Nutzung von NGWS-Komponenten denn überhaupt aus? Nach Einschätzung des Autors eher nicht Vielmehr setzen Sie Ihren Code erneut potenziellen Versionskonflikten aus – obwohl sich Konflikte natürlich weitgehend dadurch ausschalten lassen, dass andere Entwickler, die Ihre NGWS-Komponente einsetzen, geeignete Bindungsregeln verwenden. Da der Speicherplatz, den eine Komponente benötigt, heutzutage wohl nicht mehr sonderlich ins Gewicht fällt, sei Ihnen weitgehend die Verwendung privater NGWS-Komponenten empfohlen, denen Sie ja trotzdem eindeutige Namen geben können.

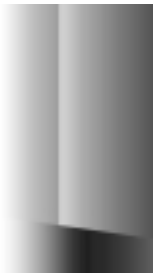
9.4 Zusammenfassung

Dieses Kapitel hat Ihnen drei Techniken vorgestellt, die Sie Ihren Komponenten und Anwendungen vor der Verbreitung angedeihen lassen können. Die erste Technik ist die bedingte Übersetzung: Der Einsatz des (vom Compiler emulierten) C#-Präprozessors sowie des `conditional`-Attributs ermöglichen es Ihnen, verschiedene Bestandteile des Codes in Abhängigkeit von der Definition einzelner oder mehrerer Symbole ein- und auszublenden. Auf diese Weise können Sie denselben Quelltext verwenden, um unterschiedliche Versionen (beispielsweise Debug-, Demo- und Release-Version) zu kompilieren.

Die zweite Technik ist die kommentarbasierte Dokumentation. Da die korrekte und ausführliche Dokumentation heutzutage als wichtiger Bestandteil des Entwicklungsvorgangs gilt und nicht mehr nur als leidiges Nachspiel, sollte man auf die mit C# verfügbare automatisierte Erstellung einer Dokumentation keinesfalls verzichten, zumal das vom Compiler generierte XML-Format eine direkte Integration der Informationen in das Hilfesystem anderer Programmierwerkzeuge, beispielsweise Visual Studio 7, ermöglicht.

Die dritte Technik betrifft die Versionspflege von NGWS-Komponenten, der kleinsten eigenständigen Einheit des NGWS-Subsystems. Prinzipiell haben Sie die Wahl zwischen privaten und gemeinsam genutzten NGWS-Komponenten. Obwohl das NGWS-Laufzeitsystem für gemeinsam genutzte Komponenten eine Versionsprüfung zur Ladezeit vorsieht, beugt eine Beschränkung auf private Komponenten Versionskonflikten immer noch am einfachsten und wirksamsten vor.

Kapitel 10



Integration von nicht verwaltetem Code

Das NGWS-Laufzeitsystem ist fürwahr ein Triumph der Technik – und wäre keinen rostigen Heller wert, wenn es sich gegen die Wiederverwendung existierenden nicht verwalteten Codes sperren würde; unabhängig davon, ob es dabei um COM-Komponenten oder um exportierte Funktionen aus C-DLLs geht. Außerdem mag es Situationen geben, in denen verwalteter Code der Performance im Wege steht.

NGWS und C# bieten Ihnen die folgenden Möglichkeiten zur Integration nicht verwalteten Codes:

- Zusammenarbeit mit COM
- Aufruf plattformspezifischer Dienste
- unsicherer Code

10.1 Zusammenarbeit mit COM

Da COM und NGWS wohl für längere Zeit nebeneinander existieren werden, ist die Interoperabilität mit COM wohl das wichtigste Teilgebiet, wenn es um nicht verwalteten Code geht: Clients des NGWS-Laufzeitsystems müssen in der Lage sein, existierende COM-Komponenten zu verwenden, und COM-Clients muss die Möglichkeit offen stehen, sich der neuen NGWS-Komponenten zu bedienen.

Die folgenden Abschnitte setzen sich mit beiden Wegen auseinander:

- Verfügbarkeit von NGWS-Objekten für COM
- Verfügbarkeit von COM-Objekten für NGWS-Anwendungen

Auch wenn sich die Diskussion auf C# konzentriert, behalten Sie bitte im Kopf, dass diese Techniken und Prinzipien in gleicher Weise auf VB sowie auf verwaltetes C++ anwendbar sind. Tatsächlich geht es hier um die Fähigkeiten des NGWS-Laufzeitsystems und nicht um die Möglichkeiten einer einzelnen Programmiersprache.

10.1.1 Verfügbarkeit von NGWS-Objekten für COM

Die eine Richtung in der Zusammenarbeit zwischen den beiden Systemen besteht darin, COM-Clients die Benutzung von Komponenten des NGWS-Laufzeitsystems zu erlauben (die, wie gesagt, in einer der Programmiersprachen C#, VB oder verwaltetem C++ geschrieben sein können). Die folgenden Abschnitte demonstrieren, wie das in der Praxis aussieht, und verwenden als Beispiel die in Kapitel 8, »Komponenten mit C# schreiben«, erstellten Komponenten `RequestWebPage` und `WhoisLookup` (beide in der Version mit eigenem Namensraum).

Im Wesentlichen geht es dabei um zwei Punkte:

- Registrieren eines Objekts beim NGWS-Laufzeitsystem
- Aufruf eines NGWS-Objekts durch einen COM-Client

Registrieren eines Objekts beim NGWS-Laufzeitsystem

COM-Objekte müssen registriert (also in der Registrierung des Systems eingetragen) werden, bevor man sie verwenden kann. Für COM 1.0 geschieht das üblicherweise über das Windows-Utility *RegSvr32*, das sich für COM+ (also die COM-Version 2.0 und damit die NGWS) nicht verwenden lässt. Hierfür gibt es ein eigenes Hilfsprogramm namens *RegAsm.Exe*.

RegAsm trägt NGWS-Komponenten in die Registrierung des Systems ein und berücksichtigt dabei sämtliche in einer Komponente enthaltenen Klassen, soweit sie öffentlich verfügbar sind. Außerdem erzeugt dieses Programm optional eine Registrierungsdatei (.REG), die sich nicht nur bei der Installation auf anderen Systemen nützlich macht, sondern auch die schnelle Prüfung ermöglicht, welche Einträge der Registrierung hinzugefügt wurden.

Der Aufruf für die in Kapitel 8 erstellte Bibliothek sieht so aus:

```
regasm presenting.csharp.dll /reg:presenting.csharp.reg
```

Listing 10.1 gibt die dabei erzeugte Registrierungsdatei (`csharp.reg`) wieder. Wer Erfahrung mit COM hat, dem dürfte die Art der meisten Einträge geläufig sein. Die ID der Komponenten wird hier aus dem Namensraum und dem Klassennamen generiert.

Listing 10.1: Die von *regasm.exe* erzeugte Registrierungsdatei

```
 1: REGEDIT4
 2:
 3: [HKEY_CLASS_ROOT\Presenting.CSharp.RequestWebPage]
 4: @="COM+ class: Presenting.CSharp.RequestWebPage"
 5:
 6: [HKEY_CLASS_ROOT\Presenting.CSharp.RequestWebPage\CLSID]
 7: @="{6B74AC4D-4489-3714-BB2E-58F9F5ADEEA3}"
 8:
 9: [HKEY_CLASS_ROOT\CLSID\{6B74AC4D-4489-3714-BB2E-58F9F5ADEEA3}]
10: @="COM+ class: Presenting.CSharp.RequestWebPage"
11:
12: [HKEY_CLASS_ROOT\CLSID\{6B74AC4D-4489-3714-BB2E-58F9F5ADEEA3}
    \InprocServer32]
13: @="D:\WINNT\System32\MSCorEE.dll"
14: "ThreadingModel"="Both"
15: "Class"="Presenting.CSharp.RequestWebPage"
16: "Assembly"="csharp, Ver=1.0.1.0"
```

```
17:
18: [HKEY_CLASS_ROOT\CLSID\{6B74AC4D-4489-3714-BB2E-_å58F9F5ADEEA3}
    \ProgId]
19: @="Presenting.CSharp.RequestWebPage"
20:
21: [HKEY_CLASS_ROOT\Presenting.CSharp.WhoisLookup]
22: @="COM+ class: Presenting.CSharp.WhoisLookup"
23:
24: [HKEY_CLASS_ROOT\Presenting.CSharp.WhoisLookup\CLSID]
25: @="{8B5D2461-07DB-3B5C-A8F9-8539A4B9BE34}"
26:
27: [HKEY_CLASS_ROOT\CLSID\{8B5D2461-07DB-3B5C-A8F9-8539A4B9BE34}]
28: @="COM+ class: Presenting.CSharp.WhoisLookup"
29:
30: [HKEY_CLASS_ROOT\CLSID\{8B5D2461-07DB-3B5C-A8F9-8539A4B9BE34}
    \InprocServer32]
31: @="D:\WINNT\System32\MSCorEE.dll"
32: "ThreadingModel"="Both"
33: "Class"="Presenting.CSharp.WhoisLookup"
34: "Assembly"="csharp, Ver=1.0.1.0"
35:
36: [HKEY_CLASS_ROOT\CLSID\{8B5D2461-07DB-3B5C-A8F9-8539A4B9BE34}
    \ProgId]
37: @="Presenting.CSharp.WhoisLookup"
```

Wie ein Blick auf die Zeilen 30 bis 34 zeigt, wird zum Anlegen von Objektvariablen nicht die Bibliothek selbst, sondern die Execution Engine (`MSCorEE.dll`) aufgerufen. Sie ist hier in erster Linie für das Erzeugen einer COM-kompatiblen Hülle (CCW = COM Callable Wrapper) verantwortlich.

Wenn Sie die Komponente ohne den Umweg über eine separate Registrierungsdatei direkt in die Registrierung eintragen wollen, reicht der Aufruf:

```
regasm presenting.csharp.dll
```

Nach dieser Registrierung lässt sich die Komponente in jeder Programmiersprache verwenden, die späte Bindung unterstützt. Wenn Sie mit später Bindung nicht zufrieden sind – was Sie übrigens auch nicht sein sollten –, verwenden Sie das Utility *TlbExp* zum Anlegen einer Typbibliothek für die NGWS-Komponente:

```
tlbexp presenting.csharp.dll /out:presenting.csharp.tlb
```

Diese Typbibliothek lässt sich mit Programmiersprachen verwenden, die frühe Bindung unterstützen, und macht die NGWS-Komponente sozusagen zu einem gesetzestreuen neuen Bürger im Staate COM.

Und nachdem Sie damit endgültig in der Welt von COM angekommen sind, sollten Sie einen kurzen Ausflug mitten in die Typbibliothek unternehmen, weil sich

dabei einige wichtige Dinge zeigen lassen. Die in Listing 10.2 gezeigte IDL-Datei (Interface Description Language) lässt sich beispielsweise über das zu Visual Studio gehörige Utility *OLE View* gewinnen, indem Sie damit auf die (mit *TlbExp* erzeugte) Typbibliothek der NGWS-Komponente losgehen.

Listing 10.2: Die IDL-Datei für die Klassen *WhoisLookup* und *RequestWebPage*

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: <could not determine filename>

[
  uuid(A4466FD5-EB56-3C07-A0D8-43153AC4FD06),
  version(1.0)
]
library csharp
{
    // TLib :     // TLib :  : {BED7F4EA-1A96-11D2-8F08-
       00A0C9A6186D}
    importlib("mscorlib.tlb");
    // TLib : OLE Automation : {00020430-0000-0000-C000-
       000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface _RequestWebPage;
    interface _WhoisLookup;

    [
      uuid(6B74AC4D-4489-3714-BB2E-58F9F5ADEEA3),
        custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        Presenting.CSharp.RequestWebPage")
    ]
    coclass RequestWebPage {
        [default] interface _RequestWebPage;
        interface _Object;
    };

    [
      odl,
      uuid(1E8F7AAB-FA6C-315B-9DFE-59C80C6483A9),
      hidden,
      dual,
      nonextensible,
      oleautomation,
        custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "Presenting.CSharp.RequestWebPage")

```

```
38:     ]
39:     interface _RequestWebPage : IDispatch {
40:         [id(00000000), propget]
41:         HRESULT ToString([out, retval] BSTR* pRetVal);
42:         [id(0x60020001)]
43:         HRESULT Equals(
44:                         [in] VARIANT obj,
45:                         [out, retval] VARIANT_BOOL* pRetVal);
46:         [id(0x60020002)]
47:         HRESULT GetHashCode([out, retval] long* pRetVal);
48:         [id(0x60020003)]
49:         HRESULT GetType([out, retval] _Type** pRetVal);
50:         [id(0x60020004), propget]
51:         HRESULT URL([out, retval] BSTR* pRetVal);
52:         [id(0x60020004), propput]
53:         HRESULT URL([in] BSTR pRetVal);
54:         [id(0x60020006)]
55:         HRESULT GetContent([out] BSTR* strContent);
56:     };
57:
58:     [
59:       uuid(8B5D2461-07DB-3B5C-A8F9-8539A4B9BE34),
60:         custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
          "Presenting.CSharp.WhoisLookup")
61:     ]
62:     coclass WhoisLookup {
63:         [default] interface _WhoisLookup;
64:         interface _Object;
65:     };
66:
67:     [
68:       odl,
69:       uuid(07255177-A6E5-3E9F-BAB3-1B3E9833A39E),
70:       hidden,
71:       dual,
72:       nonextensible,
73:       oleautomation,
74:         custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
          "Presenting.CSharp.WhoisLookup")
75:
76:     ]
77:     interface _WhoisLookup : IDispatch {
78:         [id(00000000), propget]
79:         HRESULT ToString([out, retval] BSTR* pRetVal);
80:         [id(0x60020001)]
81:         HRESULT Equals(
82:                         [in] VARIANT obj,
```

```
83:                          [out, retval] VARIANT_BOOL* pRetVal);
84:         [id(0x60020002)]
85:         HRESULT GetHashCode([out, retval] long* pRetVal);
86:         [id(0x60020003)]
87:         HRESULT GetType([out, retval] _Type** pRetVal);
88:     };
89: };
```

Wenn Sie bereits mit C++ Erfahrungen gesammelt haben, sind Sie derartige Monstren wohl gewohnt. Für VB-ProgrammiererInnen stellt dieser Blick in die Innereien von IDL-Dateien dagegen vermutlich eine Erstbegegnung dar.

Wie in den Zeilen 24 und 62 von Listing 10.2 zu sehen, verfügen beide Co-Klassen über eine von `IDispatch` abgeleitete Schnittstelle sowie eine Schnittstelle namens `Object`. Die Standardschnittstelle ist ebenfalls von `IDispatch` abgeleitet und enthält alle (öffentlichen) Methoden der jeweiligen NGWS-Komponente sowie die Methoden der `Object`-Schnittstelle. Außerdem dürfte auffallen, dass die Methoden nun sämtlich mit den allseits bekannten und beliebten Typen `BSTR` und `VARIANT` arbeiten.

Abbildung 10.1 zeigt, wie *OLE View* die Schnittstelle von `RequestWebPage` darstellt, und bietet wenig Überraschendes: Die Eigenschaft `URL` steht über die `get`- und `set`-Methoden zur Verfügung, außerdem gibt es offensichtlich vier Methoden der `Object`-Schnittstelle. Tatsächlich sieht die gesamte Definition fast genauso wie in C# aus:

Bei der Schnittstelle für die Klasse `WhoisLookup` (siehe Abbildung 10.2) liegen die Verhältnisse ein klein wenig anders. Auch hier sind die vier Methoden der `Object`-Schnittstelle mit von der Partie – aber wo ist `Query`?

Der Grund dafür, dass `Query` hier nicht erscheint: Diese Methode ist statisch, gehört also zur Klasse als Typ, nicht zu ihren Instanzen – und kann deshalb in COM nicht eingesetzt werden. Tatsächlich können Sie die Klasse `WhoisLookup` erst nach Umgestaltung der Methode `Query` in eine nicht statische Methode als COM-Objekt einsetzen. Leider gilt diese Beschränkung universell: Wenn Sie also planen, NGWS-Klassen gelegentlich auch außerhalb der NGWS-Laufzeitumgebung zu verwenden, sollten Sie sorgfältig abwägen, inwieweit Sie die unbestreitbaren Vorteile statischer Methoden überhaupt nutzen wollen.

Aufruf eines NGWS-Objekts durch einen COM-Client

Sobald die NGWS-Komponente registriert ist und Sie für Entwicklungsumgebungen mit früher Bindung eine Typbibliothek erzeugt haben, sind Sie für alle Fälle gerüstet. Wie einfach NGWS-Komponenten einsetzbar sind, lässt sich recht gut mit Microsoft Excel demonstrieren.

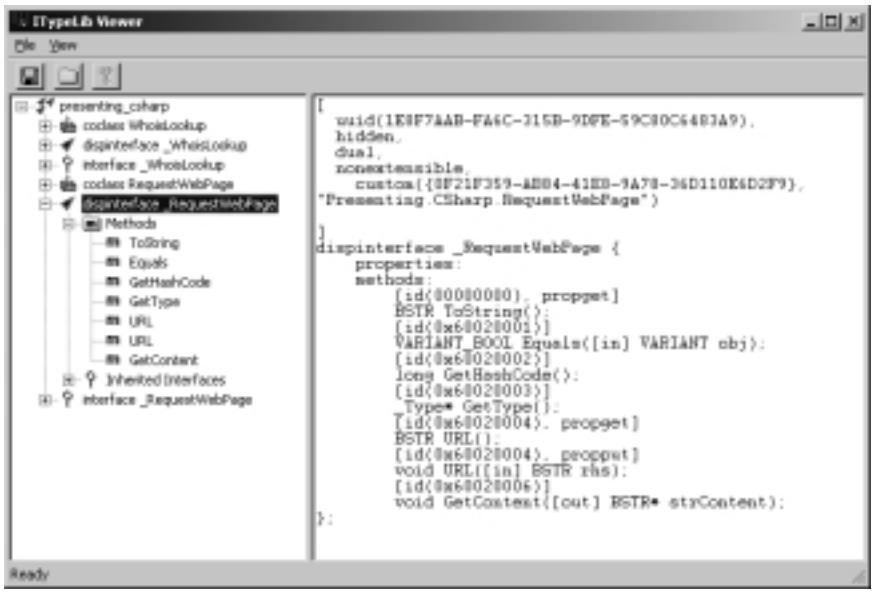


Bild 10.1: Die Klasse *RequestWebPage* mit der Eigenschaft *URL* und der Methode *GetContent*

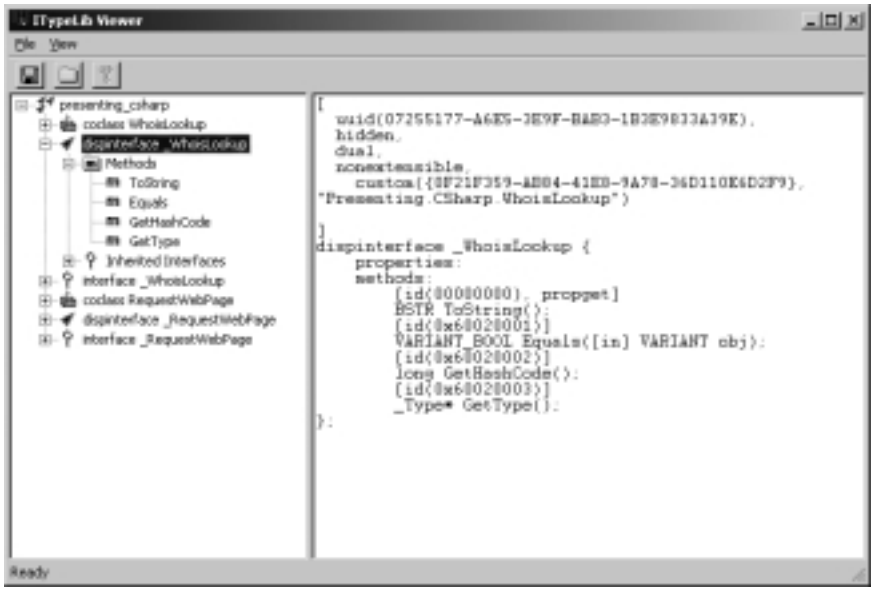


Bild 10.2: Statische Methoden sind nichts für COM.

Die einzige zusätzliche Vorbedingung für das Scripting von Komponenten in Excel (mit früher Bindung, wohlgemerkt!) ist eine Referenz auf die Typbibliothek. Starten Sie also den VBA-Editor (über *Extras/Makro/Visual Basic-Editor*), wählen Sie dort im Menü *Extras* den Befehl *Verweise*, klicken Sie im daraufhin erscheinenden Dialogfeld *Verweise – VBA-Projekt* auf *Durchsuchen* und wählen Sie die Typbibliothek der NGWS-Komponente aus.

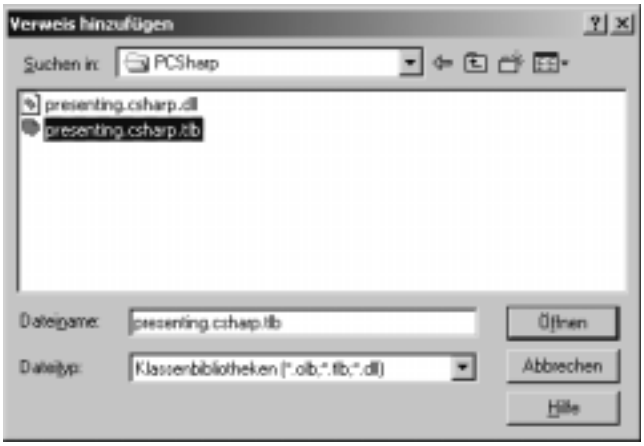


Bild 10.3: Import der Typbibliothek für die Komponente

Wie Listing 10.3 zeigt, gestaltet sich eine Abfrage über `RequestWebPage` nach diesen Vorbereitungen recht geradlinig; die Behandlung von COM-Ausnahmen übernimmt hier ein `On Error GoTo`.

Listing 10.3: Einsatz der Klasse *`RequestWebPage`* in einem Excel-Modul

```
 1: Option Explicit
 2:
 3: Sub GetSomeInfo()
 4: On Error GoTo Err_GetSomeInfo
 5:     Dim wrq As New csharp.RequestWebPage
 6:     Dim strResult As String
 7:
 8:     wrq.URL = "http://www.alphasierrapapa.com/iisdev/"
 9:     wrq.GetContent strResult
10:     Debug.Print strResult
11:
12:     Exit Sub
13: Err_GetSomeInfo:
14:     MsgBox Err.Description
15:     Exit Sub
16: End Sub
```

Vielleicht werden Sie sich fragen, wie Visual Basic an die Ausnahmen des NGWS-Laufzeitsystems herankommt? Solche Ausnahmen werden erst einmal in entsprechende `HRESULT`-Werte umgesetzt, auf die Excel dann mit einer Fehlermeldung reagiert. Für die Übermittlung zusätzlicher Fehlerinformationen (wie der textuellen Beschreibung) sind wiederum andere Schnittstellen zuständig.

Die in Listing 10.3 gezeigte Routine zeigt die abgerufenen HTML-Daten im Direktfenster von Excel an. Wenn Sie sehen wollen, wie das NGWS-Laufzeitsystem eine Ausnahme an einen COM-Client weiterreicht, probieren Sie es einmal mit einem absichtlich ungültigen URL.

10.1.2 Verfügbarkeit von COM-Objekten für NGWS-Anwendungen

Wie zu Anfang dieses Kapitels erwähnt, ist die Interoperabilität auch in der umgekehrten Richtung gegeben: NGWS-Clients können sich also auch existierender COM-Komponenten bedienen. In der Praxis wird man diesen Weg zumindest in der Übergangszeit von COM nach NGWS wohl recht häufig gehen.

Auch im Zusammenhang mit NGWS-Clients gilt die für COM-Objekte typische Unterscheidung nach der Art der Bindung:

- Aufruf von COM-Objekten mit früher Bindung
- Aufruf von COM-Objekten mit später Bindung

Die in diesem Abschnitt zur Demonstration verwendete COM-Komponente *AspTouch* ändert das Erstellungsdatum einer Datei. Sie hat eine duale Schnittstelle sowie eine Typbibliothek, ist Freeware – und Sie sind herzlich eingeladen, diese Komponente von `http://www.alphasierrapapa.com/iisdev/components/` herunterzuladen, um die folgenden Erläuterungen auch praktisch nachzuvollziehen.

Aufruf von COM-Objekten mit früher Bindung

Für die frühe Bindung einer COM-Komponente wird eine Typbibliothek benötigt. In der NGWS-Laufzeitumgebung finden sich analoge Informationen in den Metadaten, die zusammen mit dem jeweiligen Typ gespeichert sind. Womit sich natürlich die Frage stellt, woher das NGWS-Laufzeitsystem die Typinformationen für eine COM-Komponente bezieht – und wie es mit nicht verwaltetem Code überhaupt zurechtkommt.

Beide Aufgaben übernimmt eine Hülle um die COM-Komponente, die auch als RCW (Runtime Callable Wrapper) bezeichnet und mit Hilfe der Informationen aus der Typbibliothek aufgebaut wird. Für das Anlegen solcher Hüllen hält das NGWS SDK ein Dienstprogramm namens `tlbimp` (type library import) bereit, dessen Aufruf sich recht unspektakulär gestaltet:

```
tlbimp asptouch.dll /out:asptouchlib.dll
```

Dieser Befehl liest die COM-Typbibliothek von `asptouch.dll` (die in dieser DLL als Ressource gespeichert ist), generiert mit diesen Informationen einen RCW und speichert ihn in der Datei `asptouchlib.dll`. Mit einem weiteren Dienstprogramm – `ildasm.exe` – können Sie untersuchen, welche Metadaten die neu angelegte DLL enthält (siehe Abbildung 10.4). Mehr zu `ILDasm` und den Möglichkeiten dieses Werkzeugs finden Sie in Kapitel 11, »C#-Code debuggen«.

Bild 10.4: Die mit *`ILDasm`* dargestellten Metadaten von *`asptouchlib.dll`*

Offensichtlich ist `ASPTOUCHlib` der Namensraum (der sich aus dem Namen der Typbibliothek ergibt) und `TouchIt` der Name der Stellvertreterklasse, die für die COM-Komponente erzeugt wurde. Mit diesen Informationen bewaffnet, können Sie nun einen NGWS-Client schreiben, der die COM-Komponente verwendet. Das folgende Listing gibt ein Beispiel:

Listing 10.4: Einsatz einer COM-Komponente über einen RCW in einem C#-Programm

```
 1: using System;
 2: using ASPTOUCHLib;
 3:
 4: class TouchFile
 5: {
 6:  public static void Main()
 7:  {
 8:   TouchIt ti = new TouchIt();
 9:   bool bResult = false;
10:   try
11:   {
12:    bResult = ti.SetToCurrentTime("touch.cs");
```

```
13:   }
14:   catch(Exception e)
15:   {
16:    Console.WriteLine(e);
17:   }
18:   finally
19:   {
20:    if (true == bResult)
21:    {
22:     Console.WriteLine("Datum und Uhrzeit von touch.cs neu
       gesetzt.");
23:    }
24:   }
25:  }
26: }
```

Unterschiede zu anderen C#-Programmen werden Sie hier keine finden: Der Code enthält eine `using`-Anweisung, ruft Methoden der Klasse *TouchIt* auf, und behandelt Ausnahmezustände (die hier gegebenenfalls aus `HRESULT`-Werten generiert werden). Der Aufruf des Compilers enthält ebenfalls keine Besonderheiten:

```
csc /r:asptouchlib.dll /out:touch.exe touch.cs
```

Kurz und gut: Von RCWs umhüllte COM-Komponenten gleichen in jeder Hinsicht NGWS-Komponenten. Mehr gibt es dazu erfreulicherweise nicht zu sagen.

Aufruf von COM-Objekten mit später Bindung

Wenn Ihnen für eine COM-Komponente keine Typbibliothek zur Verfügung steht oder Ihr Programm aus anderen Gründen ohne RCW auskommen muss, hilft ein nettes NGWS-Feature weiter, das als *Reflexion* bezeichnet wird und es Ihrer Anwendung erlaubt, die notwendigen Informationen zur Laufzeit zu beschaffen.

Das in Listing 10.5 wiedergegebene Programm demonstriert diese Technik: Es führt dieselben Aktionen aus wie das vorangehende Beispiel, kommt aber ohne eine Hüllklasse aus.

Listing 10.5: Einsatz eines COM-Objekts über Reflexion

```
 1: using System;
 2: using System.Reflection;
 3:
 4: class TestLateBound
 5: {
 6:  public static void Main()
 7:  {
 8:   Type tTouch;
 9:   tTouch = Type.GetTypeFromProgID("AspTouch.TouchIt");
10:
```

```
11:   Object objTouch;
12:   objTouch = Activator.CreateInstance(tTouch);
13:
14:   Object[] parameters = new Object[1];
15:   parameters[0] = "reflect.cs";
16:   bool bResult = false;
17:
18:   try
19:   {
20:    bResult = (bool)tTouch.InvokeMember("SetToCurrentTime",
21:     BindingFlags.InvokeMethod,
22:     null, objTouch, parameters);
23:   }
24:   catch(Exception e)
25:   {
26:    Console.WriteLine(e);
27:   }
28:
29:   if (bResult)
30:    Console.WriteLine("Datum und Uhrzeit von reflect.cs neu
      gesetzt.");
31:  }
32: }
```

Die für die Reflexion eingesetzte Klasse hat den Namen `Type` und ist im Namensraum `System.Reflection` definiert. In Zeile 9 wird die Methode `GetTypeFromProgID` dieser Klasse mit der `ProgId` der fraglichen COM-Komponente aufgerufen: Sie sollte den tatsächlichen Typ ermitteln. Das Beispiel enthält der Übersichtlichkeit halber keine Ausnahmebehandlung, die man in ernsthaften Anwendungen aber auf jeden Fall einbauen sollte, weil `GetTypeFromProgID` eine Ausnahme signalisiert, falls sich der Typ nicht bestimmen und laden lässt.

Nachdem der Typ geladen ist, lässt sich über die statische Methode `CreateInstance` der Klasse `Activator` ein Objekt dieses Typs erzeugen. Danach wird es leider etwas unübersichtlicher: Der Aufruf von Methoden über späte Bindung hat es nämlich in sich.

Wer dieses Verfahren von C++ und COM her kennen und lieben gelernt hat, wird sich hier sofort zu Hause fühlen: Sämtliche Parameter – im gegebenen Beispiel der Dateiname – müssen in ein Array gepackt werden (Zeilen 14 und 15), der Aufruf der Methode geschieht indirekt über `InvokeMember` des `Type`-Objekts (Zeilen 20 bis 22). Zu übergeben sind hier der Name der Methode, die Bindungsflags, ein Bindungsobjekt (oder eben NULL), das anzusprechende Objekt und das Parameter-Array. Das Funktionsergebnis dieses Aufrufs muss schließlich in einen entsprechenden Typ von C# bzw. der NGWS-Laufzeitumgebung umgewandelt werden.

Das sieht nicht nur übel aus, sondern ist es auch – obwohl das Beispiel eine ausgesprochen harmlose Variante darstellt. Wenn Sie Lust auf einen wirklich erbaulichen Nachmittag verspüren, dann probieren Sie es doch einmal mit der Übergabe von Referenzen.

Kurz und schlecht: Man *kann* mit später Bindung arbeiten, wenn es denn gar nicht anders geht. Der Hauptgrund, wieso Sie wo immer möglich mit RCWs arbeiten sollten, ist allerdings nicht die einfachere Handhabung, sondern die Geschwindigkeit: Methodenaufrufe über späte Bindung sind um Größenordnungen zeitaufwändiger als bei früher Bindung.

10.2 Aufruf plattformspezifischer Dienste

Obwohl die NGWS-Klassen eine Menge zu bieten haben und es beim Einsatz von COM-Komponenten praktisch keine Grenzen gibt, werden Sie ab und an einmal auf Funktionen der Win32-API oder einer anderen nicht verwalteten DLL zurückgreifen wollen (oder müssen). Speziell für diesen Zweck wurden die *Platform Invocation Services* geschaffen, die in der SDK-Dokumentation unter dem Stichwort *PInvoke* zusammengefasst sind. Im Wesentlichen geht es dabei um das Heraussuchen und Aufrufen der gewünschten Funktion sowie die Übergabe von Parametern und Ergebnissen zwischen verwaltetem und nicht verwaltetem Code (Stichwort: Marshaling).

Bei der Definition von `extern`-Methoden in C# geben Sie zusätzlich das Attribut `sysimport` an, dessen Syntax so aussieht:

```
[sysimport(
  dll=dllname,
  name=Einsprungpunkt,
  charset=Zeichensatz
)]
```

Der Parameter `dll` ist obligatorisch, die beiden anderen sind optional. Wenn Sie das Attribut `name` nicht angeben, wird der Name eingesetzt, den Sie der Funktion im C#-Quelltext gegeben haben (und der dann seinerseits mit dem Namen der gewünschten Funktion aus der DLL übereinstimmen muss).

Listing 10.6 zeigt am Beispiel der Win32-Funktion `MessageBox`, wie einfach sich die Sache gestalten kann.

Listing 10.6: Aufruf einer Win32-Funktion über *PInvoke*

```
1: using System;
2:
3: class TestPInvoke
4: {
5:  [sysimport(dll="user32.dll")]
```

```
 6:  public static extern int MessageBoxA(int hWnd, string strMsg,
 7:       string strCaption, int nType);
 8:
 9:  public static void Main()
10:  {
11:   int nMsgBoxResult;
12:   nMsgBoxResult = MessageBoxA(0, "Hello C#", "PInvoke", 0);
13:  }
14: }
```

Zeile 5 gibt über das Attribut `sysimport` an, dass sich die gesuchte Funktion in `user32.dll` befindet. Da hier ein `name=` fehlt, verwendet der C#-Compiler den in der folgenden `extern`-Definition angegebenen Funktionsnamen (`MessageBoxA`, wobei das `A` für die ANSI-Version der Funktion steht). Das Ergebnis des Programmchens ist ein einfaches Meldungsfenster mit der Überschrift »PInvoke« und dem Text »Hello C#«.

Wenn Sie den Namen der Funktion als Parameter von `sysimport` angeben, ist der interne Name der Funktion beliebig:

Listing 10.7: Der name-Parameter von sysimport

```
1: using System;
2:
3: class TestPInvoke
4: {
5:  [sysimport(dll="user32.dll", name="MessageBoxA")]
6:  public static extern int PopupBox(int h, string m, string c, int
    type);
7:
8:  public static void Main()
9:  {
10:  int nMsgBoxResult;
11:  nMsgBoxResult = PopupBox(0, "Hello C#", "PInvoke", 0);
12:  }
13: }
```

Auch wenn sich diese beiden Beispiele auf eine ausgesprochen einfache Win32-Funktion beschränken: Grenzen gibt es hier definitiv keine. Niemand hält Sie davon ab, mit dieser Technik auf Win32-Ressourcen zuzugreifen oder gar ein eigenes Marshaling zu implementieren. (Bei Vorhaben dieses Kalibers empfiehlt sich allerdings ein ausgiebiger Blick in die Dokumentation des NGWS-SDKs.)

10.3 Unsicherer Code

Unsicheren Code zu schreiben, dürfte sicher nicht Ihr tägliches Brot werden, wenn Sie mit C# programmieren. Falls Sie aber eben doch einmal Zeiger brauchen sollten, dann steht dem vom Prinzip her nichts im Wege. C# definiert in diesem Zusammenhang zwei Schlüsselwörter:

- `unsafe` - kennzeichnet einen unsicheren Kontext und muss als Modifizierer für Elementfunktionen verwendet werden, die unsicheren Code enthalten. Anwendbar ist dieser Modifizierer auf Konstruktoren, Methoden und Eigenschaften.
- `fixed` – kennzeichnet Variablen, die der Garbage Collector nicht verschieben soll

Solange Sie nicht wirklich mit nackten Speicherbereichen arbeiten – also Zeiger verwenden – müssen, sollten Sie allerdings in fast allen Fällen mit Aufrufen plattformspezifischer Dienste und der von NGWS gebotenen COM-Interoperabilität auskommen, wenn es um Win32-Funktionen und/oder COM-Objekte geht.

Um Ihnen dennoch eine Vorstellung davon zu vermitteln, wie die Schlüsselwörter `unsafe` und `fixed` praktisch eingesetzt werden, zeigt Listing 10.8 eine Klasse, die Quadrate so berechnet, wie man es von C und C++ her gewohnt ist. Weitere Details zu unsicherem Code finden Sie in der C#-Referenz des NGWS SDK.

Listing 10.8: Ein Beispiel für unsicheren Code

```
 1: using System;
 2:
 3: public class SquareSampleUnsafe
 4: {
 5:  unsafe public void CalcSquare(int nSideLength, int *pResult)
 6:  {
 7:   *pResult = nSideLength * nSideLength;
 8:  }
 9: }
10:
11: class TestUnsafe
12: {
13:  public static void Main()
14:  {
15:   int nResult = 0;
16:
17:   unsafe
18:   {
19:    fixed(int* pResult = &nResult)
20:    {
21:     SquareSampleUnsafe sqsu = new SquareSampleUnsafe();
```

```
22:      sqsu.CalcSquare(15, pResult);
23:      Console.WriteLine(nResult);
24:    }
25:   }
26:  }
27: }
```

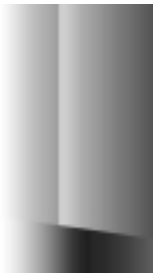
10.4 Zusammenfassung

In diesem Kapitel ging es ausschließlich um die verschiedenen Aspekte der Zusammenarbeit zwischen verwaltetem und nicht verwaltetem Code. NGWS-Komponenten lassen sich auch von COM-Clients aus benutzen, und COM-Komponenten stehen NGWS-Anwendungen zur Verfügung. Bei der – in allen Fällen empfehlenswerten – frühen Bindung von COM-Komponenten kommen Hüllklassen zum Einsatz, die über COM-Typbibliotheken erzeugt werden und die entsprechenden Informationen als Metadaten mitführen.

Für den Aufruf plattformspezifischer Funktionen übernimmt das NGWS-Laufzeitsystem mit `Pinvoke` den Transfer von Parametern und Ergebnissen. Bei Bedarf steht C#-Anwendungen auf diese Weise die gesamte Win32-API (und jede andere exportierte DLL-Funktion) zur Verfügung.

Im letzten Abschnitt des Kapitels ging es schließlich um unsicheren Code. Auch wenn verwalteter Code natürlich die bessere Wahl darstellt, können Sie bei Bedarf mit Zeigern arbeiten und Speicherbereiche fixieren – also all die Dinge tun, die verwaltetes C# sinnvollerweise nicht erlaubt.

Kapitel 11



C#-Code debuggen

Kommt es bei Ihnen oft vor, dass Sie Code schreiben, ihn ein- oder zweimal ausführen und, wenn er die gewünschte Ausgabe liefert, als »getestet« klassifizieren? Hoffentlich nicht. Ein anständiger Codetest sollte mindestens ein zeilenweises Durchlaufen des Quellcodes mit dem Debugger beinhalten und wird auch dann nur die Existenz von Fehlern nachweisen können, nicht jedoch Fehlerfreiheit.

Das Debuggen von Code ist nicht nur die wichtigste sondern leider auch zeitaufwändigste Aufgabe bei der Entwicklung von Software. Wie nicht anders zu erwarten, stellt das NGWS-SDK verschiedene Werkzeuge bereit, mit denen Sie Ihren Code nach allen Regeln der Kunst analysieren und »entwanzen« können. Machen Sie Gebrauch davon! Dieses Kapitel demonstriert Ihnen den Einsatz der folgenden Werkzeuge:

- SDK-Debugger
- IL-Disassembler

11.1 Aufgabenfeld des Debuggers

Das zum NGWS-Subsystem gehörige SDK umfasst zwei Werkzeuge zur Fehlersuche: einen Kommandozeilen-Debugger namens *CORDBG* und den Benutzerschnittstellen-Debugger *Microsoft URT-Debugger*, der kurz *SDK-Debugger* genannt wird. Letzterer, eine abgespeckte Version des zu Visual Studio 7 gehörigen Debuggers, wird in diesem Abschnitt eingehender vorgestellt. Im Vergleich mit dem Debugger von Visual Studio weist der SDK-Debugger folgende Beschränkungen auf:

- Der SDK-Debugger unterstützt keine Fehlersuche auf der Ebene von Maschinencode. Er ist auf verwalteten Code beschränkt.
- Der SDK-Debugger unterstützt kein *Remote Debugging* von einer anderen Maschine aus. Für diesen Zweck müssen Sie auf den Debugger von Visual Studio zurückgreifen.
- Das Register-Fenster ist zwar vorhanden, jedoch ohne Funktion.
- Das Disassembler-Fenster ist zwar vorhanden, jedoch ohne Funktion.

Diese Beschränkungen wirken sich nur dann als solche aus, wenn die Fehlersuche in gemischtsprachliche Umgebungen führt oder eben Remote Debugging erforderlich ist. Ansonsten bietet der SDK-Debugger die gleichen Möglichkeiten wie andere Debugger:

- Ausführen einer Debug-Version der Anwendung
- Auswählen der ausführbaren Datei
- Setzen von Haltepunkten

- schrittweise Programmausführung
- Einklinken in eine laufende Anwendung
- Variablen inspizieren und modifizieren
- Verwaltung von Ausnahmen
- Just-in-time-Debugging
- Fehlersuche in Komponenten

11.1.1 Eine Debug-Version erzeugen

Um den vollen Funktionsumfang des Debuggers zu nutzen, müssen Sie eine Debug-Version Ihrer Anwendung erstellen. Die Debug-Version enthält alle notwendigen Debug-Informationen, ist nicht optimiert und wird durch eine zusätzliche PDB-Datei (Programmdatenbank) ergänzt, in der alle Quelltextverweise und aktuellen Projektparameter aufgeführt sind.

Um den Compiler zur Erstellung einer Debug-Version zu veranlassen, geben Sie zwei zusätzliche Schalter bei seinem Aufruf an:

```
csc /optimize- /debug+ whilesample.cs
```

Dieses Kommando erzeugt die beiden für den Aufruf des Debuggers geforderten Dateien `whilesample.exe` und `whilesample.pdb`. Das folgende Listing gibt den Quellcode von `whilesample.cs` wieder, der auch in den folgenden Abschnitten verschiedentlich erscheinen wird:

Listing 11.1: Quelltext des in der Debugger-Sitzung analysierten Programms

```
 1: using System;
 2: using System.IO;
 3:
 4: class WhileDemoApp
 5: {
 6:   public static void Main()
 7:   {
 8:     StreamReader sr = File.OpenText ("whilesample.cs");
 9:     String strLine = null;
10:
11:     while (null != (strLine = sr.ReadLine()))
12:     {
13:         Console.WriteLine(strLine);
14:     }
15:
16:     sr.Close();
17:   }
18: }
```

11.1.2 Auswählen der ausführbaren Datei

Der erste Schritt in einer Debugger-Sitzung ist die Auswahl der Anwendung, für die die Fehlersuche erfolgen soll. Obwohl sich der Debugger auch in eine laufende Anwendung einklinken kann (mehr dazu später) ist der üblichere Fall der, dass Sie die Anwendung mit der Absicht starten, eine Fehlersuche durchzuführen. Auswahl und Start der Anwendung geschehen somit vom Fenster des Debuggers aus.

Die Debug-Version des zu testenden Programms `whilesample.exe` wurde ja bereits im vorigen Abschnitt erzeugt. Für den Start des SDK-Debuggers müssen Sie die Anwendung `DbgUrt.exe` ausführen, die im Verzeichnis *`Laufw`*`:\Program Files\NGWSSDK\GuiDebug` zu finden ist. Abbildung 11.1 zeigt das Fenster des Debuggers.

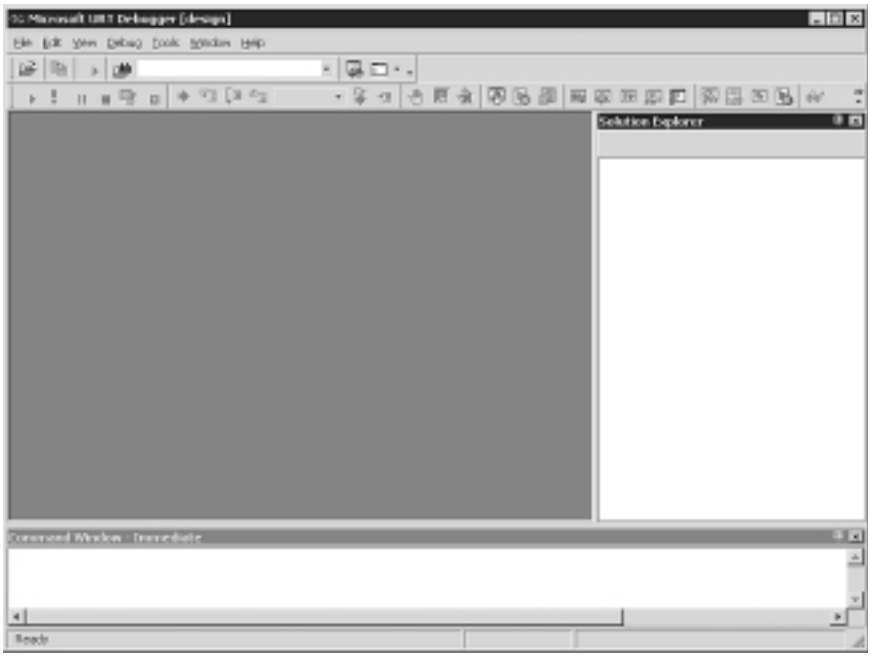


Bild 11.1: Das Fenster des Debuggers

Um die ausführbare Datei für die Debugger-Sitzung auszuwählen, öffnen Sie über den Befehl *Debug/Program to Debu*g das Dialogfeld *Program To Debu*g (Abbildung 11.2) und geben darin den Ausführungspfad der Anwendung sowie gegebenenfalls ein vom Anwendungsverzeichnis abweichendes Arbeitsverzeichnis an.

Bild 11.2: Auswahl der ausführbaren Datei für die Debuggersitzung

Beachten Sie, dass Sie in dem Dialogfeld auch Kommandozeilenargumente angeben können, die der Debugger dem Programm beim Aufruf mitgibt. Da die Beispielanwendung nicht auf eine Auswertung von Kommandozeilenargumenten ausgelegt ist, können Sie das Textfeld leer lassen.

Vom Prinzip her können Sie die Anwendung nach diesem Arbeitsgang sofort starten. In der Praxis werden Sie zuvor noch einen oder mehrere Haltepunkte an Stellen setzen, die Sie im Verlauf der Programmausführung überwachen wollen.

11.1.3 Setzen von Haltepunkten

Der SDK-Debugger unterstützt vier Arten von Haltepunkten:

- Codebasierter Haltepunkt – der Debugger unterbricht das Programm, wenn der Code einer bestimmten Quelltextzeile zur Ausführung kommen soll.
- Datenbasierter Haltepunkt – der Debugger unterbricht das Programm, wenn eine Variable (beispielsweise eine Schleifenvariable) einen spezifischen Wert annimmt.
- Funktionsbasierter Haltepunkt – der Debugger unterbricht die Ausführung an einer spezifischen Stelle innerhalb einer Funktion.
- Adressbasierter Haltepunkt – der Debugger unterbricht die Ausführung, wenn der Code eine spezifische Adresse im Hauptspeicher anspricht.

Mit Abstand am häufigsten werden codebasierte Haltepunkte bei der Fehlersuche verwendet. Um beispielsweise einen solchen Haltepunkt in die Zeile 11 (also an den Beginn der Schleife) des Beispielprogramms zu setzen, gehen Sie wie folgt vor:

1. Öffnen Sie die Quelldatei `whilesample.cs` über den Befehl *File/Open*.
2. Klicken Sie mit der rechten Maustaste in die bewusste Zeile und wählen Sie aus dem erscheinenden Kontextmenü den Befehl *Insert Breakpoint*. Das Fenster Ihres SDK-Debuggers sollte wie in Abbildung 11.3 aussehen: Ein roter Punkt am linken Bereichsrand zeigt an, dass die Zeile einen Haltepunkt enthält (nur datenbasierte Haltepunkte sind nicht auf diese Weise gekennzeichnet).

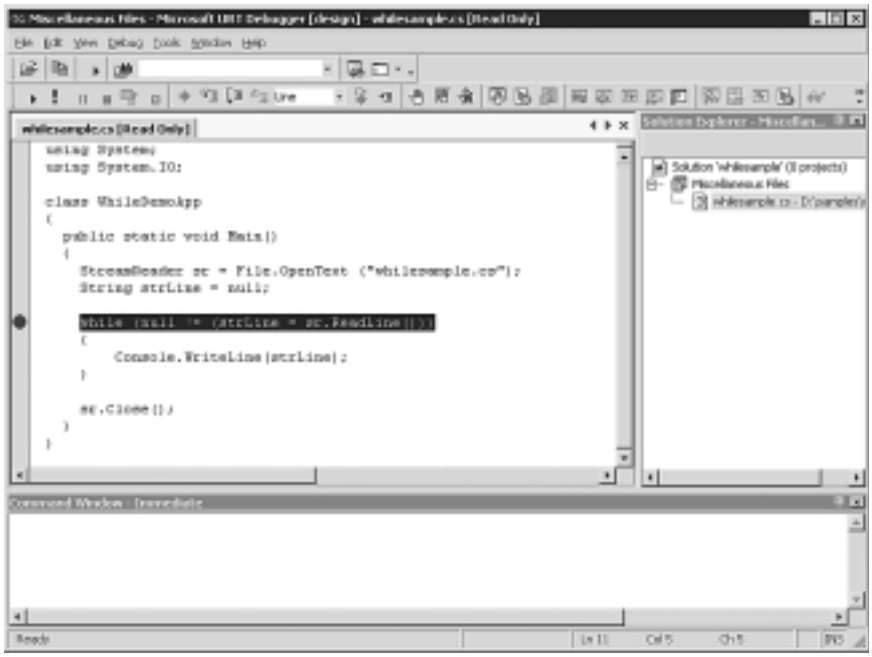


Bild 11.3: Einen Haltepunkt im SDK-Debugger einfügen

Das war es bereits. Falls Sie die Eigenschaften des Haltepunkts bearbeiten wollen, wählen Sie den Befehl *Breakpoint Properties* im Kontextmenü der Zeile. Der daraufhin erscheinende Dialog gestattet es Ihnen, eine Bedingung für den Haltepunkt festzulegen sowie eine Zählung zu aktivieren – mit dem Effekt, dass der Debugger den Code erst dann unterbricht, wenn die Bedingung das *n*-te Mal erfüllt ist.

Im Fenster *Breakpoints*, das über den Befehl *Windows/Breakpoints* im Menü *Debug* geöffnet wird, erhalten Sie einen Überblick über alle gesetzten Haltepunkte sowie deren Bedingungen (vgl. Abbildung 11.4).

Bild 11.4: Inspizieren eines Haltepunkts im Fenster *Breakpoints*

Nachdem der Haltepunkt definiert ist, können Sie das Programm über den Debugger starten. Dazu geben Sie entweder den Befehl *Start* im Menü *Debug* oder klicken auf das gleichnamige Symbol (Rechtspfeil) in der Symbolleiste. Wie nicht anders zu erwarten, unterbricht der Debugger das Programm, sobald die Ausführung den Haltepunkt erreicht hat. Nun können Sie das Programm schrittweise ausführen.

11.1.4 Schrittweise Programmausführung

Wenn der Debugger Ihre Anwendung an einem Haltepunkt angehalten hat, können Sie die Ausführung der Anwendung auf verschiedene Arten wieder aufnehmen. Die entsprechenden Befehle finden Sie in der Symbolleiste sowie im Menü des Debuggers:

- *Step Over* – der Debugger führt die nächste Anweisung en bloc aus und unterbricht danach die Ausführung wieder.
- *Step Into* – der Debugger führt die nächste Einzelanweisung aus und unterbricht danach die Ausführung wieder. Im Gegensatz zu *Step Over* lässt sich die Ausführung mit *Step Into* auch in Funktionskörper hinein verfolgen.
- *Step Out* – der Debugger setzt die Ausführung des Programms bis zum Rücksprung der aktuellen Funktion fort.
- *Run to Cursor* – der Debugger setzt die Ausführung des Programms bis zu der Anweisung fort, an der die Eingabemarke im Quelltext steht, beachtet aber auch zuvor gelegene Haltepunkte.

Probieren Sie die verschiedenen Befehle aus und beenden Sie dann den Debugger.

11.1.5 Einklinken in eine laufende Anwendung

Der SDK-Debugger kann auch die Kontrolle über Anwendungen übernehmen, die er nicht selbst gestartet hat. In diesem Fall wählen Sie die Anwendung aus der Liste der laufenden Prozesse aus. Das funktioniert beispielsweise für Anwendungen, die gerade auf eine Benutzereingabe warten oder als Dienst ausgeführt werden – wichtig ist, dass der Debugger noch die Zeit hat, sich in die Anwendung einzuklinken, bevor sie endet.

Damit Sie sehen, wie das funktioniert, finden Sie im folgenden Listing noch einmal das `do…while`-Beispiel aus Kapitel 6 abgedruckt, das den Benutzer in einer Schleife auffordert, Zahlen zur Berechnung des arithmetischen Mittels einzugeben:

Listing 11.2: Die Datei *attachto.cs* ist für den nachträglichen Aufruf des Debuggers geeignet

```
 1: using System;
 2:
 3: class ComputeAverageApp
 4: {
 5:  public static void Main()
 6:  {
 7:   ComputeAverageApp theApp = new ComputeAverageApp();
 8:   theApp.Run();
 9:  }
10:
11:  public void Run()
12:  {
13:   double dValue = 0;
14:   double dSum = 0;
15:   int nNoOfValues = 0;
16:   char chContinue = 'j';
17:   string strInput;
18:
19:   do
20:   {
21:    Console.Write("Bitte eine Zahl eingeben: ");
22:    strInput = Console.ReadLine();
23:    dValue = Double.Parse(strInput);
24:    dSum += dValue;
25:    nNoOfValues++;
26:    Console.Write("Noch eine Zahl (j/n)? ");
27:
28:    strInput = Console.ReadLine();
29:    chContinue = Char.FromString(strInput);
30:   }
31:   while ('j' == chContinue);
```

```
32:
33:    Console.WriteLine("Arithmetisches Mittel: {0}", dSum / nNoOf-
       Values);
34:  }
35: }
```

Nachdem Sie das Programm mit

```
csc /optimize- /debug+ attachto.cs
```

kompiliert haben, können Sie es ganz normal über die Kommandozeile starten. Das Programm gibt dann an der Konsole die Meldung `Bitte eine Zahl eingeben` aus und wartet auf die nächste Benutzereingabe – eine gute Gelegenheit, den Debugger zu starten und ihm die Kontrolle über das Programm zu übertragen.

Dazu öffnen Sie das Dialogfeld *Programs* über den gleichnamigen Befehl im *Debug*-Menü und markieren darin die Anwendung, deren Kontrolle der Debugger übernehmen soll (vgl. Abbildung 11.5). Beachten Sie aber, dass sich der SDK-Debugger nur in solche Anwendungen einklinken kann, die dem Typ COM+ angehören – also NGWS-Anwendungen sind.

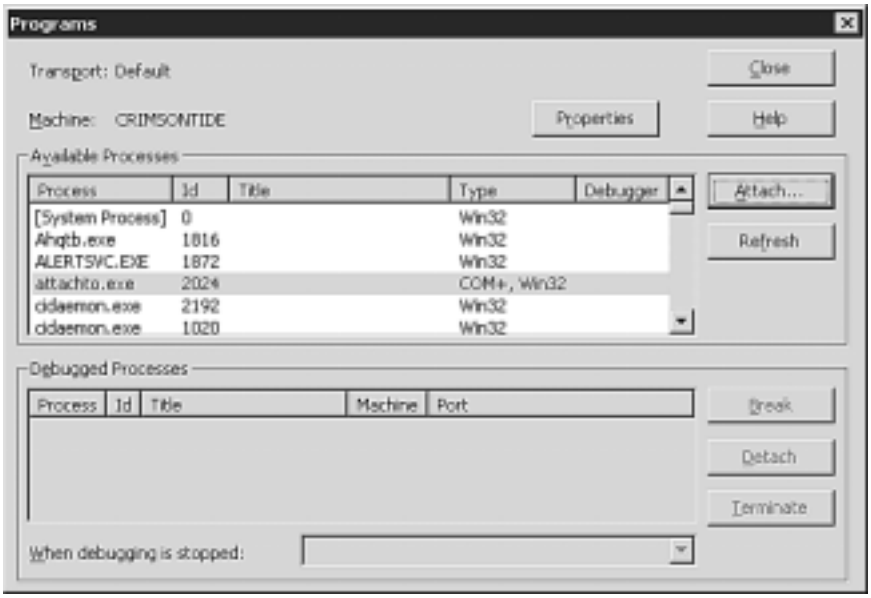


Bild 11.5: Einklinken des Debuggers in ein laufendes Programm

Wenn Sie nach einem Klick auf die Schaltfläche *Attach* das danach erscheinende Dialogfeld *Attach to Process* mit *OK* bestätigen, ändert sich die Anzeige im Dialogfeld *Programs* dahingehend, dass die gewählte Anwendung nun im Listenfeld

Debugged Processes als Eintrag erscheint. Markiert man diesen Eintrag und klickt auf die Schaltfläche *Detach*, endet die Kontrolle des Debuggers über die Anwendung (natürlich endet sie in jedem Fall auch, wenn die Anwendung oder der Debugger beendet wird).

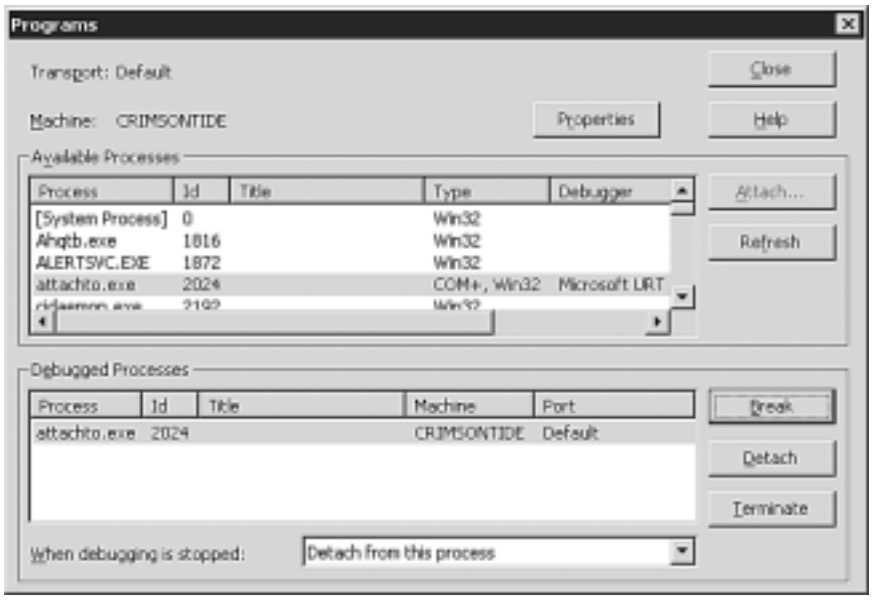


Bild 11.6: Der Debugger kann jederzeit wieder ausgeklinkt werden

Nach dem Attach-Vorgang klicken Sie auf die Schaltfläche *Break*, um die Anwendung zu unterbrechen, und schließlich auf *Close*. Sobald der Debugger die Programmausführung unterbrochen hat, öffnet er die Quellcodedatei und markiert darin die als Nächstes auszuführende Anweisung (Zeile 21). Schalten Sie zurück auf die Anwendung und geben Sie eine Zahl ein.

Der nächste Abschnitt fährt mit diesem Beispiel fort und zeigt Ihnen, wie Sie in einer Debugger-Sitzung die Werte einzelner Variablen erkunden und manipulieren.

11.1.6 Variablen inspizieren und modifizieren

Wenn Sie nun auf das Fenster des SDK-Debuggers zurückschalten, werden Sie feststellen, dass dieser immer noch in Zeile 21 verharrt. Platzieren Sie die Eingabemarke in Zeile 26 und geben Sie den Befehl *Run to Cursor*. Der Debugger liest nun die Zahl, die Sie zuvor eingegeben haben, und aktualisiert die Variablen `dSum` und `nNoOfValues`.

Um die Werte der beiden Variablen zu inspizieren, öffnen Sie das Fenster *Locals* über den Befehl *Windows/Locals* im Menü *Debug*. Wie in Abbildung 11.7 gezeigt, gibt das Fenster tabellarisch die Namen, Werte und Typen aller lokalen Variablen für die aktuelle Aufrufebene (Methode) wieder.

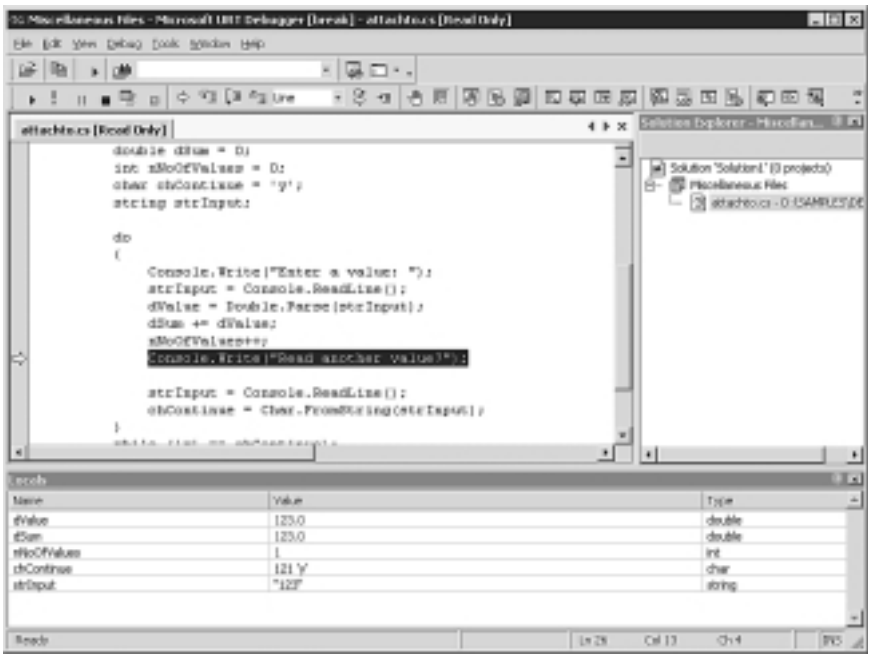


Bild 11.7: Das Fenster *Locals* listet die lokalen Variablen der aktuellen Aufrufebene auf

Wenn Sie den Wert einer Variable ändern wollen, doppelklicken Sie auf ihren Wert in der Spalte *Value* und geben dann den neuen Wert ein. Mehr ist nicht zu tun. Von nun an arbeitet das Programm mit dem neuen Wert.

Ein weiteres Mittel für die Beobachtung von Variablen und Ausdrücken ist das Fenster *Watch*. Im Gegensatz zum Fenster *Locals* ist *Watch* zu Beginn noch leer, bietet dafür aber die Möglichkeit, Variablen und Ausdrücke eigener Wahl als Einträge einzufügen. Auch bleiben die Einträge dann unabhängig von der Aufrufebene die ganze Zeit über sichtbar, selbst wenn die darin aufgeführten Variablen und Ausdrücke nicht auswertbar sind.

11.1.7 Verwaltung von Ausnahmen

Ein wirklich nützliches Feature des SDK-Debuggers ist die Möglichkeit, Regeln für den Umgang mit Ausnahmen festlegen zu können. Die diesbezügliche Konfi-

guration des Debuggers geschieht (nach Auswahl der auszuführenden Anwendung) im Dialogfeld *Exceptions*, dessen Aufruf über den Menübefehl *Debug/ Exceptions* erfolgt (vgl. Abbildung 11.8). Der Dialog gestattet es, für jede Ausnahme (im Rahmen der bestehenden Klassenhierarchie) einzeln festzulegen, wie der Debugger damit verfahren soll.

Bild 11.8: Die Reaktion des SDK-Debuggers auf verschiedene Ausnahmen festlegen

Voreinstellung für alle Ausnahmen ist die Option *Continue* im oberen Optionenblock und die Option *Break into the debugger* im unteren Optionenblock. Das heißt, dass der Debugger bei Auftreten einer Ausnahme zunächst einmal nichts unternimmt und erst in Aktion tritt, wenn die Anwendung keine Behandlung der Ausnahme vornimmt.

Obwohl Sie mit dieser Voreinstellung alle Ausnahmen mitbekommen, die Ihr Code nicht behandelt, werden Sie für den einen oder anderen Fall vielleicht eine andere Einstellung vorziehen, etwa um die Ausführung fortzusetzen, wenn eine Wertebereichsverletzung für ein Argument auftritt oder wenn Sie eine `FileIO-Exception`-Ausnahme auf jeden Fall mitbekommen möchten, selbst wenn Ihr Code eine Behandlungsroutine dafür bereitstellt.

11.1.8 Just-in-Time-Debugging

Eine unbehandelte Ausnahme ist natürlich ein hervorragender Ausgangspunkt für eine Debugger-Sitzung. Das NGWS-Laufzeitsystem konfrontiert Sie in diesem Fall mit einem Dialogfeld (vgl. Abbildung 11.9), das es Ihnen erlaubt, die Kontrolle einem Debugger Ihrer Wahl zu übergeben. Man nennt dieses Vorgehen *Just-in-time*-Debugging (JIT), da der Debugger »gerade zum richtigen Zeitpunkt« zum Einsatz kommt.

Bild 11.9: Das Dialogfeld *Just-in-Time Debugging* erscheint bei Auftreten einer unbehandelten Ausnahme

Wenn Sie das Dialogfeld über die Schaltfläche *Yes* beenden, klinkt sich der SDK-Debugger in das Programm ein, öffnet den Quellcode und markiert darin die Zeile, die für die Ausnahme verantwortlich ist. Darüber hinaus informiert Sie ein weiteres Dialogfeld (das diesmal der Debugger anzeigt) noch einmal über die Art der aufgetretenen Ausnahme (Abbildung 11.10).

Nachdem Sie das Dialogfeld weggeklickt haben, stehen Ihnen alle in diesem Kapitel vorgestellten Techniken zur Verfügung, um dem Fehler auf den Grund zu gehen.

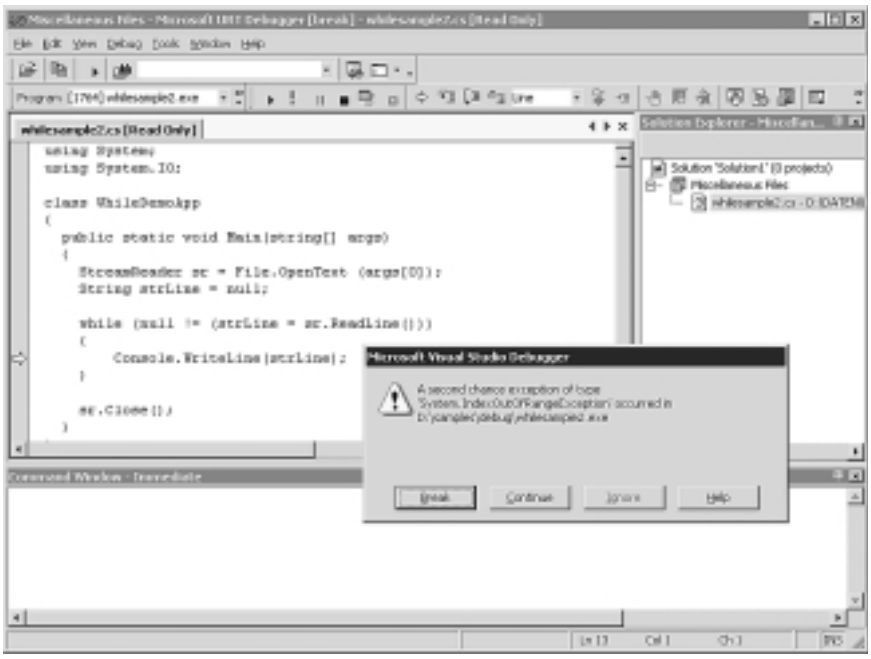


Bild 11.10: Der Debugger meldet, welche Ausnahme den Just-in-time-Aufruf bewirkt hat

11.1.9 Fehlersuche in Komponenten

Die Fehlersuche in C#-Komponenten folgt dem gleichen Prinzip wie die Fehlersuche in Komponenten, die in C++ geschrieben sind: Der Debugger wird für eine Client-Anwendung aufgerufen, die von der Komponente Gebrauch macht. Der Rest ist nicht anders als bei einer Anwendung: Sie setzen Haltepunkte in den Quellcode der Anwendung oder warten auf eine Ausnahme. Um eine Komponente zu debuggen, muss die Client-Anwendung nicht unbedingt als Debug-Version vorliegen (auch wenn es zu empfehlen ist), es reicht bereits, wenn eine Debug-Version der Komponente verfügbar ist.

Um die Debug-Version einer Bibliothek zu erstellen, geben Sie wie gehabt die Schalter `/debug+` und `/optimize-` an. Für die Beispielkomponente aus Kapitel 8, »Komponenten mit C# schreiben« würde der Compileraufruf also wie folgt lauten:

```
csc /r:System.Net.dll /t:library /out:csharp.dll /a.version:1.0.1.0
    /debug+ /optimize- whoislookup.cs requestwebpage.cs
```

Die (wie gesagt nicht unbedingt erforderliche) Debug-Version der Client-Anwendung übersetzen Sie wie gehabt:

```
csc /r:csharp.dll /out:wrq.exe /debug+ /optimize- whoisclient.cs
```

Es ist Ihnen nun freigestellt, ob Sie den Debugger sofort starten und die Client-Anwendung aus seinem Fenster heraus ausrufen, oder ob Sie ihn erst später (gegebenenfalls *just-in-time*) in die Anwendung einklinken. Wenn sowohl die Client-Anwendung also auch die Komponente als verwalteter Code und als Debug-Version vorliegen, können Sie der Ausführung mit dem Debugger schrittweise von der Client-Anwendung in die Komponente und wieder zurück folgen.

11.2 Der Intermediate Language Disassembler

Ein hübsches kleines Werkzeug aus der Werkzeugkiste des NGWS SDK ist der Intermediate Language Disassembler, kurz *IL-Disassembler* oder *ILDasm*. Trotz seiner Bestimmung, die dieser Name ohne Zweifel suggeriert, kann der IL-Disassembler auch dafür verwendet werden, wichtige Informationen über die Metadaten und Ausprägung von ausführbarem NGWS-Code darzustellen. So lohnt sich der Einsatz des Werkzeugs beispielsweise, wenn Sie einen RCW (Runtime Callable Wrapper) für eine COM-Komponente implementiert haben und ein wenig mehr über die Hüllklasse in Erfahrung bringen wollen.

Gestartet wird der IL-Disassembler über das Untermenü *Tools* im Startmenü des Microsoft NGWS SDK. Nach Auswahl einer NGWS-Komponente über den Befehl *File/Open* präsentiert sein Fenster die Typhierarchie dieser Komponente und ermöglicht das interaktive Durchforsten ihres Namensraums (vgl. Abbildung 11.11).

Ein Doppelklick auf den Eintrag MANIFEST enthüllt, welche Bibliotheken die Komponente importiert sowie verschiedene Informationen über die vorliegende Implementation (Versionsnummer etc.) der Komponente.

Was fortgeschrittene ProgrammierInnen jedoch wirklich interessieren wird: ILDasm macht wirklich den IL-Code sichtbar, den der Compiler erzeugt hat (vgl. Abbildung 11.12). Und nachdem das Fenster zu jeder IL-Code-Sequenz auch den ursprünglichen C#-Code anzeigt (jedoch nur, wenn eine Debug-Version vorliegt), lässt sich die Arbeitsweise des IL recht einfach erlernen. Eine Dokumentation der einzelnen IL-Instruktionen finden Sie im NGWS SDK.

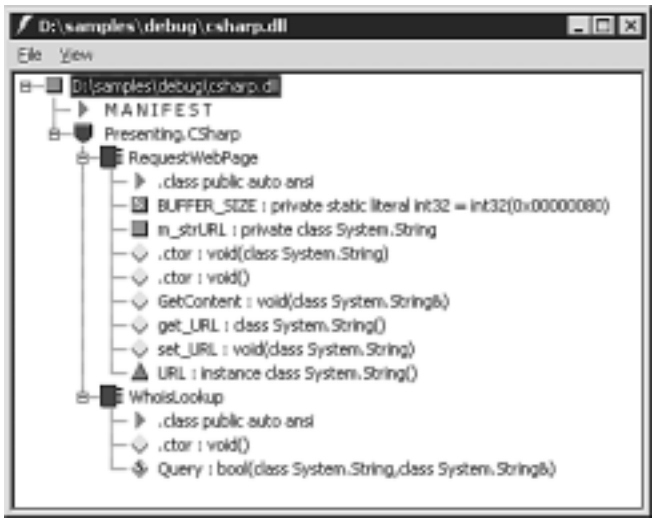


Bild 11.11: Durchforsten des Namensraums einer NGWS-Komponente mit ILDasm

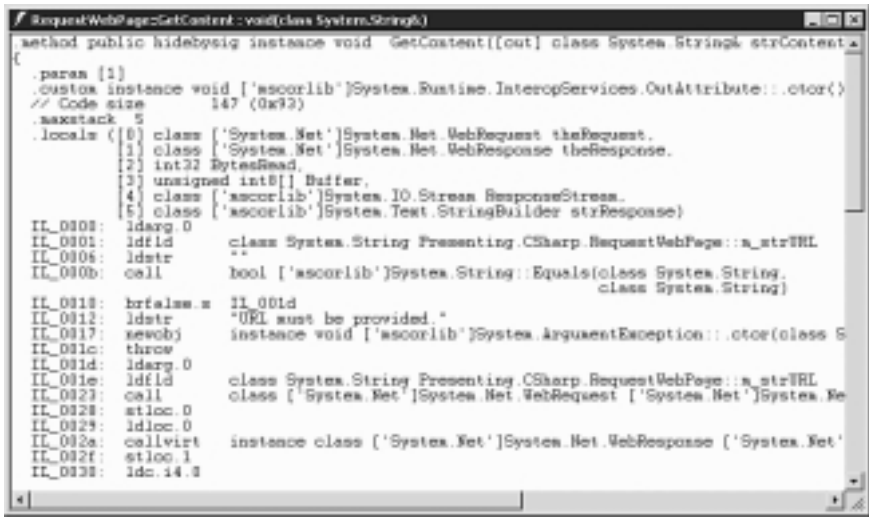


Bild 11.12: IL-Ansicht der Methode *GetContent*

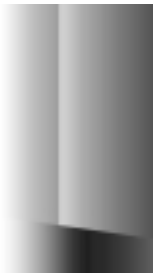
11.3 Zusammenfassung

Dieses Kapitel hat Ihnen den zum NGWS SDK gehörigen SDK-Debugger vorgestellt. Nachdem es sich bei diesem Debugger um eine leicht abgespeckte Version des zu Visual Studio gehörigen Debuggers handelt, ist der Funktionsumfang ausgesprochen reichhaltig – was die Fehlersuche natürlich sehr effizient macht.

Sie eröffnen eine Debugger-Sitzung für eine Anwendung entweder, indem Sie die Anwendung vom Fenster des Debuggers aus starten, oder, indem Sie den Debugger sich später in die bereits in Ausführung befindliche Anwendung einklinken lassen (»Attach«). Wenn der Debugger einen Haltepunkt erreicht, unterbricht er das Programm und gibt Ihnen die Möglichkeit, die Ausführung schrittweise, bis zur Position der Einfügemarke im Quelltext oder bis zum nächsten Haltepunkt fortzusetzen. An jedem Haltepunkt besteht die Möglichkeit, die Werte von Variablen nicht nur zu inspizieren, sondern gegebenenfalls auch zu manipulieren. Besondere Flexibilität bietet der Debugger auch im Umgang mit Ausnahmen: Sie können für jede Ausnahme getrennt festlegen, ob und wann der Debugger darauf reagieren soll.

Wer ein Freund von Assembler ist, dürfte im IL-Disassembler ein wertvolles Werkzeug finden, das ihm nicht nur die IL und die Umsetzung von C#-Quellcode in IL-Instruktionen nahebringt, sondern auch eine Fülle von Informationen über die Komponente selbst und ihre Metadaten zugänglich macht.

Kapitel 12



Sicherheit

Sicherheit ist ein wichtiges Thema – vor allem in einer Welt, in der Code aus einer Vielzahl von Quellen inklusive des Internet kommt. Kein Wunder also, dass diesem Punkt beim Design von NGWS erhebliche Aufmerksamkeit gewidmet wurde: Sicherheitsanforderungen werden hier sowohl auf der Codeebene als auch auf der Ebene der Benutzerrechte erfüllt.

Da sich über die vom NGWS-Subsystem gebotene Sicherheit leicht ein eigenes Buch schreiben ließe, beschränkt sich dieses Kapitel darauf, Ihnen die Konzepte und Möglichkeiten im Überblick vorzustellen. Codebeispiele fehlen auf den folgenden Seiten komplett: Da sie notwendigerweise kurz ausfallen müssten, würden sie zwangsläufig den Eindruck erwecken, Sicherheit sei bei den NGWS eine Art Nachgedanke. Und das ist sie nun wirklich nicht.

Die folgenden Seiten behandeln daher zwei Themenbereiche ausschließlich aus konzeptueller Sicht:

- Codezugriffsrechte
- Rollenbasierte Sicherheit

12.1 Codezugriffsrechte

Code gelangt heutzutage nicht mehr ausschließlich über ein vom firmeneigenen Server gestartetes Installationsprogramm auf die eigene Festplatte, sondern auch über Webseiten aus dem Internet und über E-Mails. Die Erfahrungen der jüngsten Zeit haben gezeigt, dass Code aus diesen Quellen recht gefährlich sein kann. Wie begegnet das NGWS-Subsystem dieser potenziellen Bedrohung?

Eine der von NGWS in dieser Hinsicht angebotenen Lösungen für die Sicherheit sind die *Codezugriffsrechte*: Code genießt je nach Art und Herkunft ein unterschiedlich hohes Vertrauen, und das Prädikat »uneingeschränkt vertrauenswürdig« wird nur einem ausgesprochen kleinen Teil des Codes erteilt.

Die wichtigsten Elemente dieses Sicherheitskonzepts sind:

- Administratoren können Sicherheitsrichtlinien festlegen, die Code und Codegruppen bestimmte Rechte erteilen.
- Code kann seinerseits fordern, dass ein Aufrufer bestimmte Rechte haben muss.
- Die Ausführung von Code wird durch das Laufzeitsystem eingeschränkt: Es prüft, ob die einem Aufrufer zugestandenen Rechte für die Ausführung bestimmter Operationen ausreichen.
- Code kann Rechte anfordern, die zu seiner Ausführung unbedingt notwendig sind, sowie weitere Rechte, die sinnvoll wären; darüber hinaus kann er auch explizit bekannt geben, welche Rechte er in keinem Fall benötigt.

- Für Zugriffe auf verschiedene Systemressourcen sind individuelle Rechte definiert, die sich wahlweise vergeben und entziehen lassen.
- Rechte werden beim Laden von Komponenten vergeben bzw. geprüft. Die Vergabe dieser Rechte basiert auf den Anforderungen der jeweiligen Komponente sowie den Sicherheitsrichtlinien.

Kurz und gut: Code mit geringerer Sicherheitseinstufung kann effektiv daran gehindert werden, Code mit höherer Sicherheitseinstufung aufzurufen, weil für ihn eben nur eine Teilmenge der Rechte gilt. Eine solche Einigung auf den kleinsten gemeinsamen Nenner ist vor allem bei Internetzugriffen wichtig.

Technisch gesehen, bauen Codezugriffsrechte auf zwei Schlüsselkonzepte auf: der Prüfung der Typensicherheit für verwalteten Code und der Möglichkeit, explizit bestimmte Rechte anzufordern. Woraus auch folgt: Wenn Sie die Vorteile dieses Sicherheitskonzepts genießen wollen, müssen Sie im Minimalfall typensicheren Code schreiben.

12.1.1 Prüfung der Typensicherheit

Der erste Schritt der Laufzeitumgebung bei der Durchsetzung von Zugriffsbeschränkungen besteht aus der Prüfung, ob der Code typensicher ist oder nicht. Die Typensicherheit stellt die Voraussetzung für eine zuverlässige Prüfung der Rechte eines Aufrufers dar.

Technisch gesehen, steht hinter der Überwachung von Zugriffsrechten bei Aufrufen ein schrittweises Durcharbeiten des Stacks – und das setzt wiederum voraus, dass alle dort verzeichneten Typen so gehalten sind, dass der Code auf sie nur in exakt festgelegter Weise zugreifen kann.

Erfreulicherweise erfüllt C# die Forderung nach Typensicherheit von selbst – jedenfalls, solange Sie in eine Klasse nicht explizit unsicheren Code einbauen. Das Laufzeitsystem prüft übrigens nicht nur den Zwischencode, sondern auch die Metadaten, bevor es Code niedrigerer Sicherheitseinstufung den Aufruf von Code mit höherer Sicherheitseinstufung erlaubt.

12.1.2 Rechte

Der nächste Schritt besteht aus der expliziten Anforderung von Rechten. Welche Vorteile eine solche aktive Vorgehensweise bringt? Wenn Ihr Code Rechte für seine Aktionen zu einem von Ihnen festgelegten Zeitpunkt anfordert, dann wissen Sie als Verfasser, ab wann bestimmte Aktionen zulässig sind, beziehungsweise können gegebenenfalls alternative Ausführungspfade für den Fall vorsehen, dass das System dem Code nur einen Teil der gewünschten Rechte zugesteht. Außerdem lässt sich auf diese Weise klar festlegen, welche Rechte ein bestimmter Programmteil *nicht* braucht. Code, der lediglich minimale Rechte einfordert, wird

auch auf Systemen mit hohen Sicherheitsanforderungen ausgeführt. Code, der aufs Geratewohl hin erst einmal alle Rechte haben will, dürfte auf solchen Systemen scheitern.

Die Kategorien, in die Rechte in diesem Zusammenhang fallen, wurden bereits implizit erwähnt:

- notwendig – Rechte, die Ihr Code zur Ausführung unbedingt benötigt
- optional – Rechte, die nicht unbedingt notwendig sind, die zu bewältigende Aufgabe aber vereinfachen würden
- zurückgewiesen – Rechte, die Ihr Code auch dann nicht bekommen soll, wenn die Sicherheitsrichtlinien das zulassen würden. Mit einer solchen expliziten Eingrenzung können Sie die potenziellen Sicherheitslöcher klein halten.

Die interessante Frage ist natürlich, welche Rechte sich explizit anfordern lassen, welche implizit über die Einstufung des Codes und welche schließlich über die Identität des Benutzers vergeben werden. Die Antworten auf die ersten beiden Teilpunkte geben die folgenden Abschnitte:

- Standardrechte
- Code-Identität

Mit dem dritten Punkt setzt sich der Abschnitt »Rollenbasierte Sicherheit« auseinander.

Standardrechte

Vom NGWS-Laufzeitsystem zur Verfügung gestellte Ressourcen werden über Codezugriffsrechte geschützt. Mit entsprechenden Rechten dieser Art ausgestattet, erhalten Programme Zugriff auf die jeweilige Ressource und/oder die Erlaubnis, bestimmte (geschützte) Operationen mit ihr auszuführen. Ihr Code kann jedes dieser Rechte zur Laufzeit explizit anfordern, und die Laufzeitumgebung entscheidet, ob das jeweilige Recht zugestanden wird oder nicht.

Tabelle 12.1 listet die Standardrechte zusammen mit einer kurzen Beschreibung auf. Wie dort zu sehen, haben beispielsweise die Net-Klassen ein eigenes Zugriffsrecht für die Netzwerksicherheit.

Recht	Beschreibung
`EnvironmentPermission`	Diese Klasse definiert Zugriffsrechte auf Umgebungsvariablen. Zwei Arten des Zugriffs sind hier definiert: Lesen und Schreiben, wobei Schreiben auch das Anlegen und Löschen von Umgebungsvariablen abdeckt.

Tab. 12.1: Standardrechte

Recht	Beschreibung
`FileDialogPermission`	kontrolliert den Zugriff auf Dateien über die Standarddialogfelder zum Öffnen und Speichern von Dateien. Der Benutzer muss die vom Code gewünschten Zugriffe über ein Dialogfeld explizit autorisieren.
`FileIOPermission`	definiert für Dateioperationen drei verschiedene Rechte: Lesen, Schreiben und Anhängen. Leserechte beinhalten die Abfrage von Dateiinformationen, Schreibrechte das Löschen und Überschreiben von Dateien. Beim dritten Recht geht es um das Anhängen neuer Daten an eine existierende Datei – was das Lesen des bereits vorhandenen Dateiinhalts nicht einschließt.
`IsolatedStoragePermission`	kontrolliert den Zugriff auf benutzerindividuelle Daten (die ihrerseits ein mit NGWS neu eingeführtes Konzept darstellen). Unter anderem lassen sich hier Quoten vergeben und die Dauer der Speicherung festlegen.
`ReflectionPermission`	erlaubt bzw. verwehrt die Möglichkeit zur Abfrage von Typinformationen für nicht öffentliche Elemente und ist außerdem für `Reflection.Emit` zuständig.
`RegistryPermission`	kontrolliert das Lesen, Anlegen und Ändern von Schlüsseln und Werten in der Registrierung des Systems. Jedes der gewünschten Rechte muss hier separat und zusammen mit einer Liste der Schlüssel bzw. Werte angegeben werden.
`SecurityPermission`	stellt eine Sammlung einfacher Flags für das Sicherheitssystem dar. Unter anderem können Sie hier die Ausführung von Code kontrollieren, Sicherheitsprüfungen außer Kraft setzen, Aufrufe nicht verwalteten Codes kontrollieren, Rahmenbedingungen für die Serialisierung setzen usw..
`UIPermission`	Kontrolliert den Zugriff auf verschiedene Teile der Benutzeroberfläche. Unter anderem geht es hier um den Einsatz von Fenstern, die Verwendung der Zwischenablage und die Reaktion auf Ereignisse.

Tab. 12.1: Standardrechte (Forts.)

Code-Identität

Mit der Identität des Codes verbundene Rechte können nicht explizit angefordert werden – sie ergeben sich implizit. Eine entsprechende Signatur des Codes gibt dem NGWS-Laufzeitsystem die Möglichkeit, unter anderem den Ursprung (wie etwa eine Website) sowie den Hersteller herauszufinden.

Tabelle 12.2 gibt eine kurze Beschreibung der in diesem Zusammenhang verwendeten Rechte.

Recht	Beschreibung
`PublisherIdentityPermission`	die Signatur einer NGWS-Komponente. Sie stellt sozusagen die Unterschrift des Herstellers dar.
`StrongNameIdentityPermission`	gibt den den stark verschlüsselten Namen der Komponente an. Die Identität des Codes setzt sich aus dem stark verschlüsselten Namen und dem regulären Namen der Komponente zusammen.
`ZoneIdentityPermission`	gibt die Zone an, aus der der Code stammt. (Jeder URL kann zu jedem Zeitpunkt nur einer Zone angehören.)
`SiteIdentityPermission`	gibt die Website an, von der der Code stammt
`URLIdentityPermission`	gibt den URL an, von dem der Code stammt

Tab. 12.2: Mit der Identität des Codes verbundene Rechte

12.2 Rollenbasierte Sicherheit

Auch wenn Ihnen Sicherheit auf der Basis individueller Benutzerrechte von COM her ein Begriff sein wird und sich das Sicherheitsmodell von NGWS in diesem Punkt kaum unterscheidet, sollten Sie diesen Abschnitt nicht überspringen: Einige (wenige) Dinge gibt es nämlich schon, die Ihre Beachtung verdienen.

Die rollenbasierte Sicherheit von NGWS baut auf einen sogenannten *Prinzipal* auf, der entweder einen Benutzer oder einen im Namen des Benutzers handelnden Agenten repräsentiert. NGWS-Anwendungen führen Sicherheitsprüfungen entweder auf Basis der Identität des Prinzipals oder seiner Rollenzugehörigkeit aus.

Was unter einer Rolle zu verstehen ist, lässt sich am einfachsten anhand eines Beispiels erklären. Nehmen wir also eine Bank mit ihren Angestellten, einigen Prokuristen und dem Filialleiter. Ein Angestellter kann ein Formular für einen Kreditantrag vorbereiten – unterschreiben muss es aber ein Prokurist. Welcher der Prokuristen diese Unterschrift leistet (oder ob es gleich der Filialleiter tut), ist egal: Es kommt darauf an, dass irgendjemand aus der Gruppe der Prokuristen zum Stift greift.

In einem etwas mehr technischen Kontext ist eine *Rolle* eine benannte Gruppe von Benutzern, die identische Rechte haben. Ein Prinzipal kann Mitglied mehrerer Rollen (also Benutzergruppen) sein. Ob bestimmte Aktionen in seinem Namen ausführbar sind oder nicht, ergibt sich aus der Rollenzugehörigkeit.

Wie zuvor schon einmal kurz angedeutet, muss es sich bei einem Prinzipal nicht unbedingt um einen Benutzer handeln – er kann auch ein Agent sein. Tatsächlich gibt es insgesamt drei Arten von Prinzipalen:

- Generische Prinzipale – repräsentieren Benutzer ohne Authentifizierung und die für sie erlaubten Rollen.
- Windows-Prinzipale – bilden Windows-Benutzer und deren Gruppen (Rollen) ab. Identitätswechsel (also der Zugriff auf Ressourcen im Namen eines anderen Benutzers) wird unterstützt.
- Selbstdefinierte Prinzipale – werden von Anwendungen angelegt und können den Identitätsbegriiff sowie die angenommenen Rollen bei Bedarf erweitern. Das setzt allerdings voraus, dass Ihre Anwendung für eine Authentifizierung sorgt und die Typen bereitstellt, die den Prinzipal implementieren.

Wie das in der Praxis aussieht? Nun, das NGWS-Subsystem definiert eine Klasse `PrincipalPermission`, die für die Konsistenz mit Codezugriffsrechten sorgt. Diese Klasse macht es dem Laufzeitsystem möglich, Autorisierungen in ähnlicher Weise wie Codezugriffsrechte zu prüfen und stellt Ihrem Code darüber hinaus Informationen zur Identität des Prinzipals und seiner Rollen zur Verfügung.

12.3 Zusammenfassung

Dieses letzte Kapitel hat die Sicherheitskonzepte vorgestellt, die im NGWS-Subsystem zu finden sind. Codezugriffsrechte lassen sich in aktiv angeforderte Standardrechte und Sicherheitseinstufungen aufgrund der Code-Identität (Quelle, Hersteller, etc.) unterteilen. Die rollenbasierte Sicherheit sorgt dafür, dass ein Benutzer bzw. ein ihn vertretender Agent auf die Operationen beschränkt ist, die dieser Benutzer auf dem jeweiligen System ausführen kann.