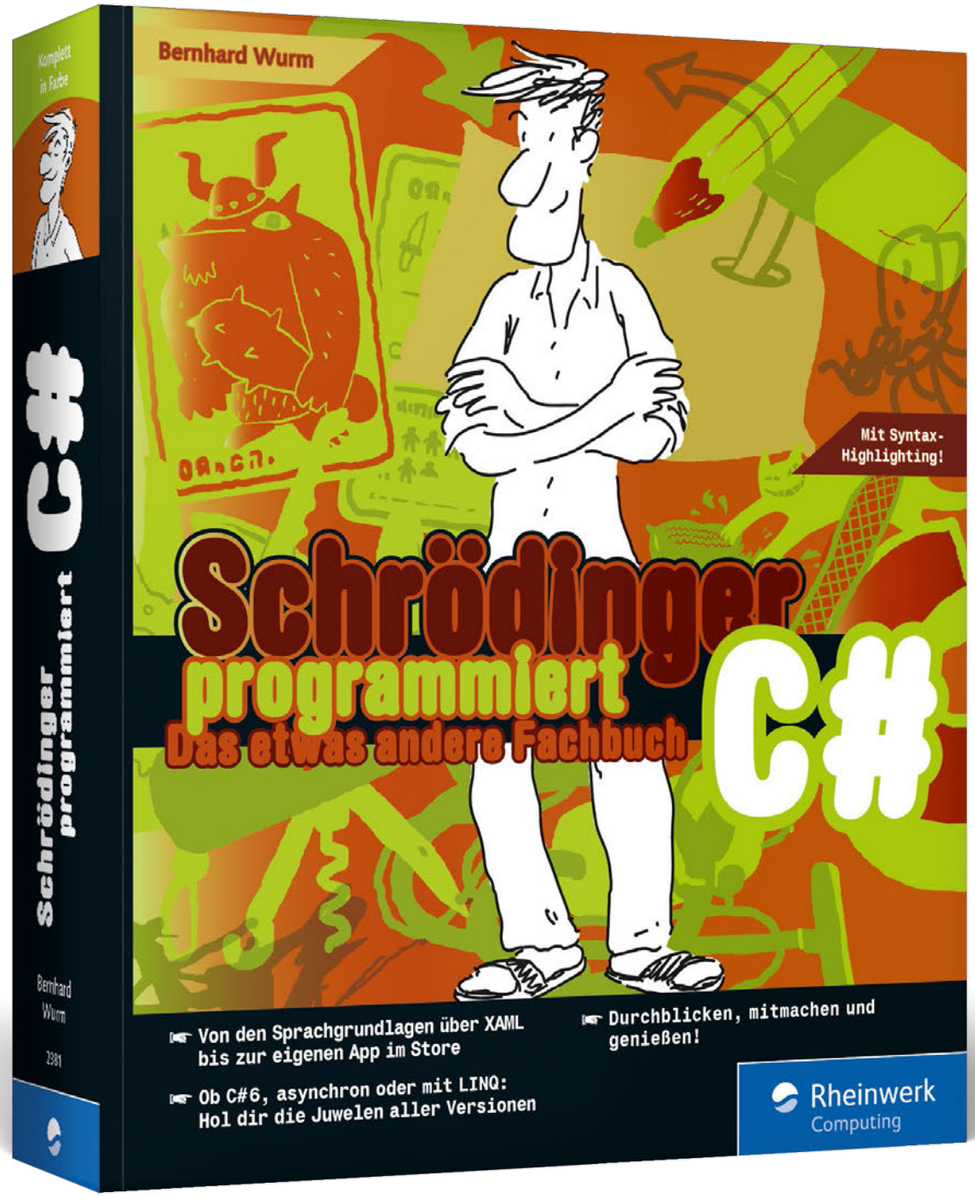


Leseprobe

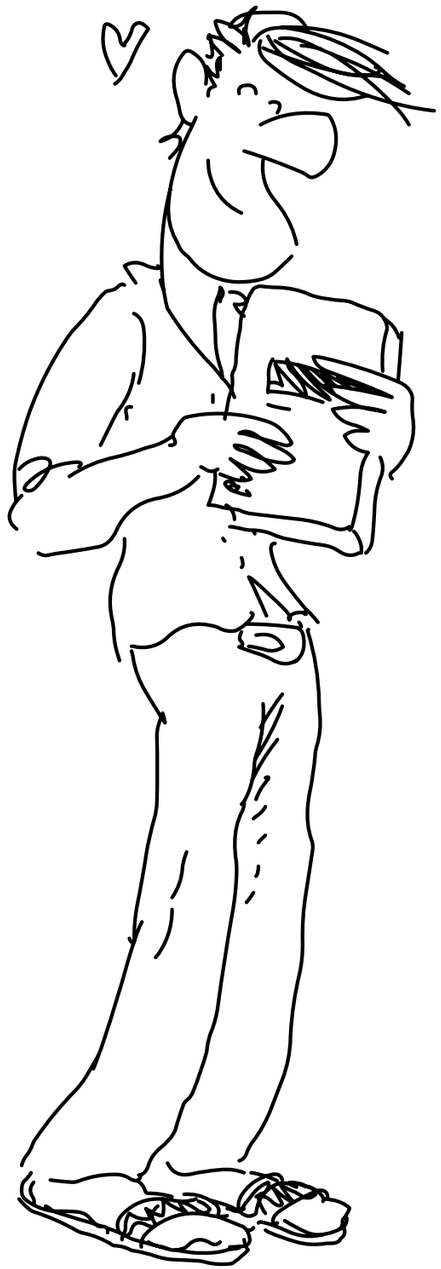
Lernen Sie mit Schrödinger C# zu programmieren. In dieser Leseprobe starten Sie in sein neues Abenteuer und lernen Compiler, Enwicklungsumgebungen und Datentypen kennen. Außerdem können Sie einen Blick in das vollständige Inhalts- und Stichwortverzeichnis des Buches werfen.

Bernhard Wurm

Schrödinger programmiert C# – Das etwas andere Fachbuch

760 Seiten, broschiert, in Farbe, März 2015
49,90 Euro, ISBN 978-3-8362-2381-2

www.rheinwerk-verlag.de/3366



MIT TALENT, KATZENPHOBIE UND LÄSSIGEM SCHUHWERK BESTACH SCHRÖDINGER DIE RHEINWERK-JURY. FÜR IHN GEHT JETZT EIN TRAUM IN ERFÜLLUNG.

Liebe Leserin, lieber Leser,

SIE HABEN DIE WAHL GETROFFEN:

C#.

Da sind wir dabei.
Lernen müssen Sie zwar selbst, aber wir geben unser Bestes, um Sie zu unterstützen:

Gut, dass Ihr das gleich klarstellt.

Wir haben einen **hervorragenden Autor** engagiert, der sich für Sie ins Zeug legt, Ihnen Sprachfeatures und Konzepte anschaulich erklärt und Ihre Entwicklung zum Profi von Anfang an im Blick hat: Best Practices gehören immer dazu, und auch bei etwas anspruchsvolleren Themen lässt er Sie nicht im Stich. Von »Hallo Welt« bis zur eigenen App im Store. Versprochen.

Wir bringen Sie mit **Schrödinger** zusammen. Nein, auch der nimmt Ihnen das Lernen nicht ab. Ehrlich gesagt, denkt er nicht einmal daran. Das wäre auch sowieso zu schade, denn Sie würden **fantastische Übungen** verpassen und womöglich gar die Illustration. Aber Spaß haben Sie mit Schrödinger bestimmt, und ein paar **schlaue Fragen** hat er obendrein parat.

Wir haben ein **Expertenteam** herbeigeholt, das den Code einfärbt, Pfeile und Hinweise anbringt, Spuren legt und gelegentlich die Lösungen auf den Kopf stellt, damit Sie nicht zu früh spinxen.

Können wir jetzt loslegen? Sonst bin ich schon mal in der Werkstatt und setze das Visual Studio auf.

Na dann: Viel Erfolg!

Schrödingers Büro

Zettel

Hintergrund info

Achtung

Ablage

Belohnung

robots in disguise

Die nötige Theorie, viele Hinweise und Tipps

15:20

Begriffsdefinition

Falscher Code

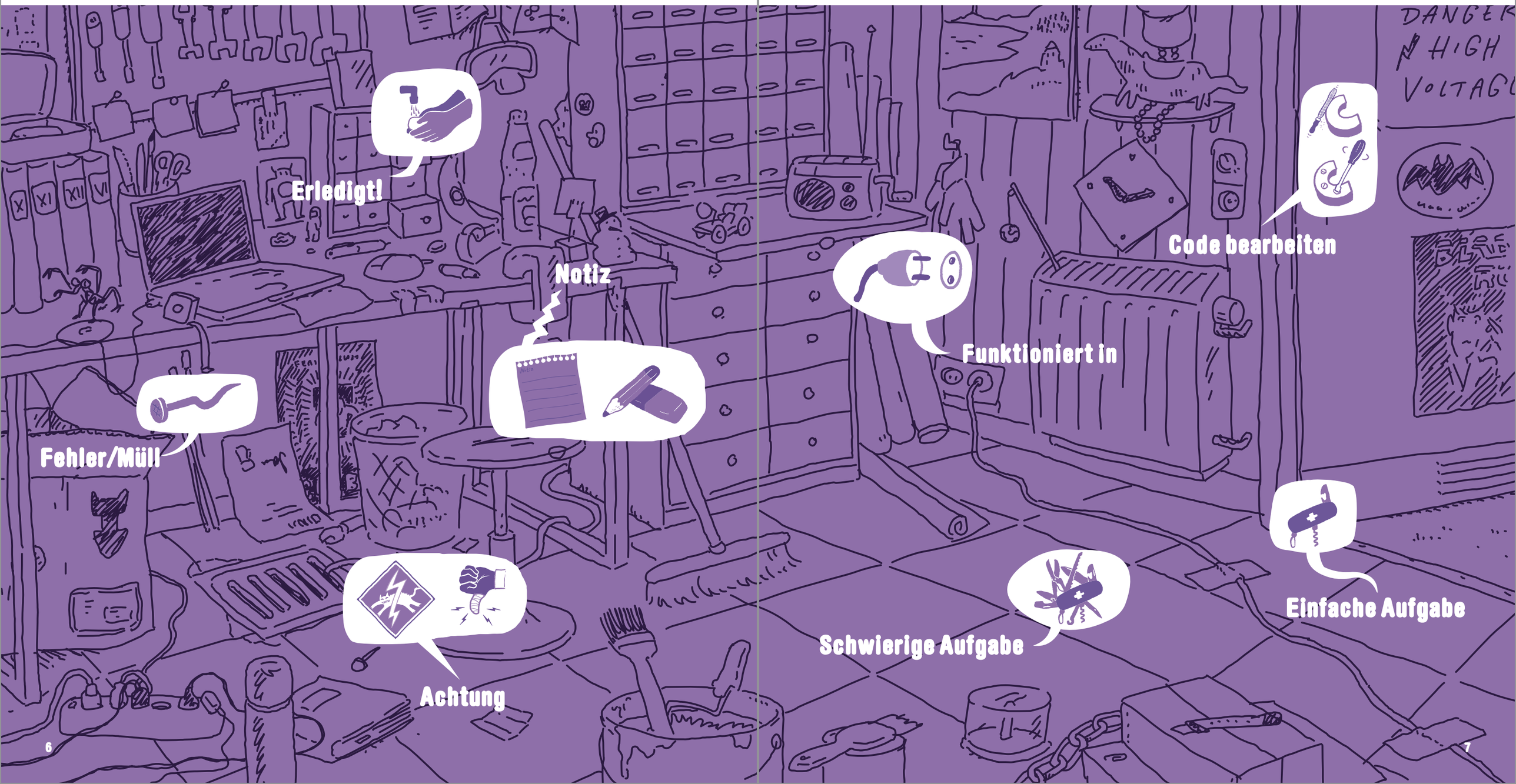
X

Einfache Aufgabe

Schwierige Aufgabe

Schrödingers Werkstatt

Unmengen von Code, der ergänzt, verbessert und repariert werden will



Schrödingers Wohnzimmer

Viel Kaffee, Übungen und die verdienten Pausen

—EINS—

Compiler und Entwicklungs-umgebungen

Ein guter Start ist der halbe Sieg

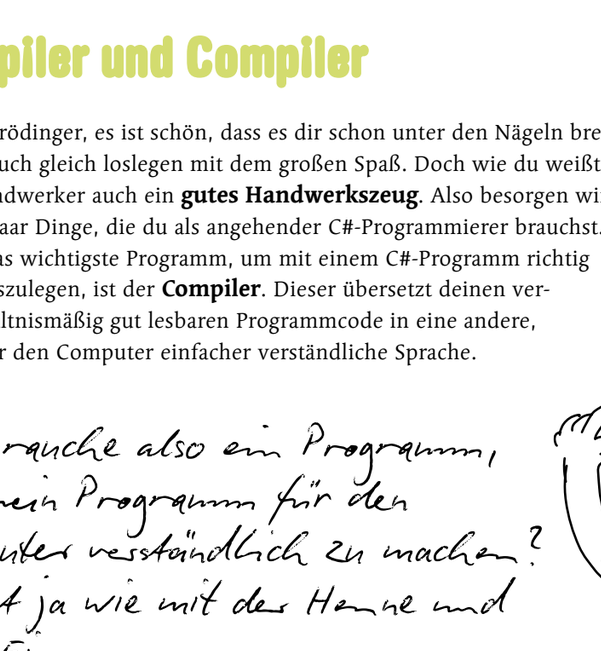
Schrödinger steht am Anfang seines neuen Jobs als C#-Entwickler. Sein Problem: Er kann noch gar kein C#. Sein erster Schritt zur Lösung: Er hat sich Hilfe geholt. Und er hat richtig Lust auf die Sache. Beste Voraussetzungen also. Jetzt ist der zweite Schritt an der Reihe: Installieren! Aber was?

Compiler und Compiler

Hallo Schrödinger, es ist schön, dass es dir schon unter den Nägeln brennt. Und wir werden auch gleich loslegen mit dem großen Spaß. Doch wie du weißt, braucht ein guter Handwerker auch ein **gutes Handwerkszeug**. Also besorgen wir dir jetzt gleich mal ein paar Dinge, die du als angehender C#-Programmierer brauchst.

Das wichtigste Programm, um mit einem C#-Programm richtig loszulegen, ist der **Compiler**. Dieser übersetzt deinen verhältnismäßig gut lesbaren Programmcode in eine andere, für den Computer einfacher verständliche Sprache.

Ich brauche also ein Programm, um mein Programm für den Computer verständlich zu machen? Das ist ja wie mit der Henne und dem Ei...



Fast noch schlimmer:
Es ist erstmal noch kein Maschinencode. Dein Programmcode wird in die **Intermediate Language** (IL-Code genannt) übersetzt. Erst **beim Ausführen des Programms** wird nun der IL-Code in **Maschinencode** übersetzt. Dies übernimmt ein weiterer Compiler – der sogenannte **Just-in-time-Compiler** (JIT-Compiler genannt).

Warte mal. Ich brauche einen Übersetzer von C# in den IL-Code und dann wieder einen von IL-Code in Maschinencode? Das ist ja umständlich!

Das wirkt vielleicht so, aber du musst wissen, **dass C# eine von mehreren Programmiersprachen ist, die auf der Entwicklungsplattform .NET laufen**. Es gibt auch andere Programmiersprachen, wie zum Beispiel Visual Basic.NET, F# usw., die ebenfalls nicht direkt in den Maschinencode übersetzt werden, und somit kannst du – sofern du so etwas willst – einfach zwischen verschiedenen Programmiersprachen wechseln und auch Bausteine von anderen Sprachen verwenden.

C#, F#, Visual Basic, dann gibt es auch noch C, C++, Java. Glaubt denn jetzt jeder, dass er eine **eigene** Programmiersprache entwickeln muss?

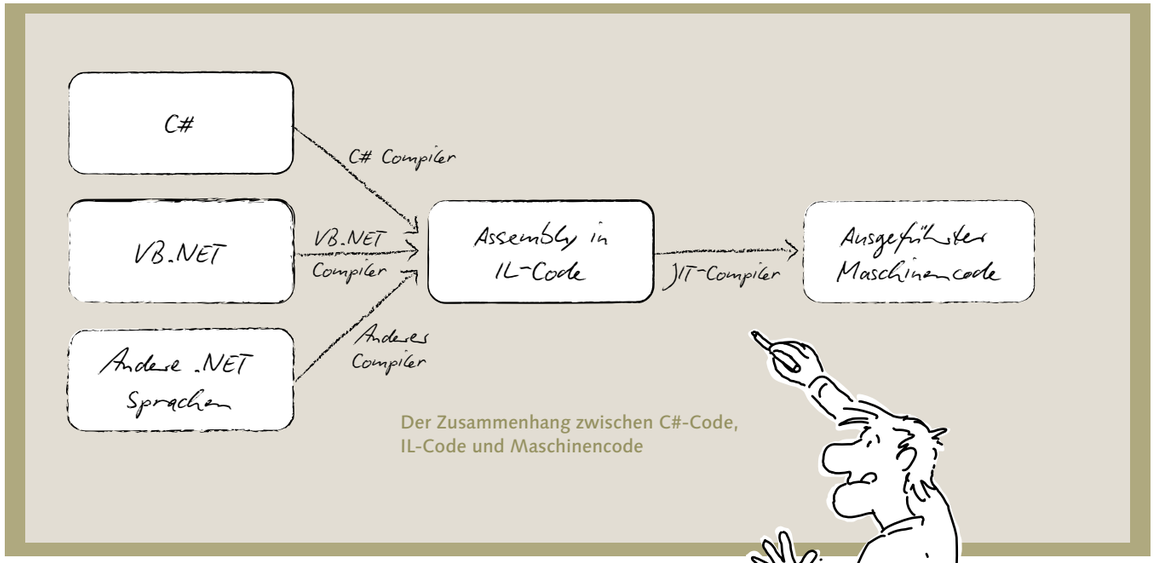
Nachdem du also mithilfe des C#-Compilers dein Programm kompiliert hast, erhältst du eine Datei, die IL-Code beinhaltet. Diese Datei nennt man **Assembly**. Ein Assembly kann, muss aber keine ausführbare EXE-Datei sein. Es kann auch eine Bibliothek – also eine DLL-Datei – sein, die verschiedene Funktionen beinhaltet, die wiederum von anderen Dateien verwendet werden können.

Aber ich dachte immer, in einer EXE-Datei steht Maschinencode? In echt steht aber IL-Code drin, und trotzdem kann sie ausgeführt werden?

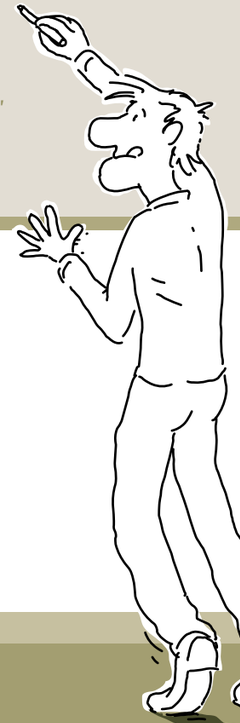
Du kannst dir das so vorstellen, dass der erste Befehl in dieser EXE-Datei ein Verweis auf die Common Language Runtime (kurz CLR) ist. Diese startet den JIT-Compiler, der den Code direkt, noch bevor dieser ausgeführt wird, in richtigen Maschinencode übersetzt und ihn dann ausführt.

[Hintergrundinfo]
Immer, wenn Code eines Assemblys ausgeführt werden soll, tritt der JIT-Compiler in Aktion. Der ist so optimiert, dass immer nur der aktuelle Codeblock kompiliert wird und nicht mehr. Das kompilierte Ergebnis wird zwischengespeichert und nur noch dieser schnelle Code ausgeführt.

Außerdem hat der IL-Code den Vorteil, dass er nicht nur in Windows funktioniert, sondern beispielsweise auch unter Linux und unterschiedlichen Rechnerarchitekturen (x86 oder ARM) funktionieren kann, da er **plattformunabhängig** ist und **erst bei der Ausführung in den Maschinencode übersetzt** wird.

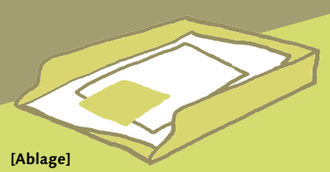


Der Zusammenhang zwischen C#-Code, IL-Code und Maschinencode

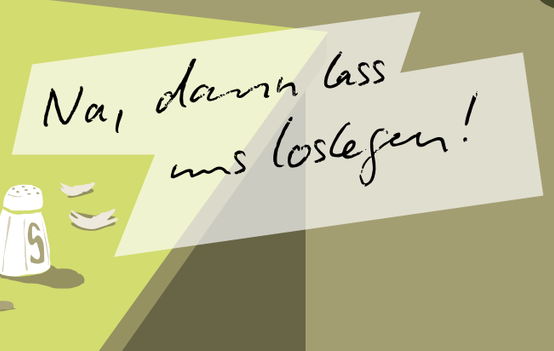


Du hast den Compiler übrigens schon auf deinem Windows-Rechner installiert. Dieser kommt nämlich automatisch mit dem .NET Framework, und das jeweils aktuelle Framework ist bei der Windows-Installation bereits dabei. Also zum Beispiel enthält Windows 7 das .NET Framework Version 4, Windows 8 das .NET Framework 4.5 und Windows 8.1 das Framework 4.5.1. Natürlich kannst du die neuen Versionen auch in ältere Windows-Versionen installieren.

Der Compiler ist die **`csc.exe`** (csc steht für C-Sharp-Compiler) und in dem Verzeichnis **`C:\Windows\Microsoft.NET\Framework64\v4.0.30319`** zu finden.



[Ablage]
Windows installiert standardmäßig eine .NET-Version auf dem Computer inklusive C#-Compiler.



Hallo Schrödinger

`Okay,` betrachte das Ganze noch einmal etwas genauer. Du kannst deinen Programmcode in jedem beliebigen Texteditor

1. schreiben,
2. abspeichern,
3. durch den Compiler anschließend zu einem Programm übersetzen und
4. laufen lassen.

`Das mach jetzt direkt mal.`
Ach, C# selbst zu **schreiben**, musst du ja noch lernen, also nimm dies:
Dieses kleine Programm gibt »Hallo Schrödinger« am Bildschirm aus, wenn du es ausführst.

`Texteditor auf und tippen:`

```
using System; *3
public class Programm { *5
  public static void Main() *1
  {
    Console.WriteLine("Hallo Schrödinger"); *2
    Console.ReadKey(); *4
  }
}
```

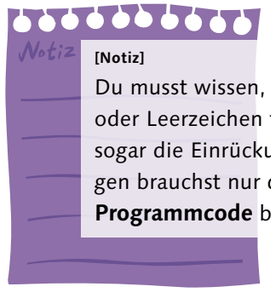
*1 Alle deine Programme starten mit der **`Main`**-Funktion. Alle Befehle innerhalb der beiden geschwungenen Klammern in dieser Funktion werden Zeile für Zeile ausgeführt.

*2 Zeigt mithilfe der Funktion **`WriteLine`** den Text **`"Hallo Schrödinger"`** in der Konsole an. Damit der Computer **`Console`** kennt, wird …

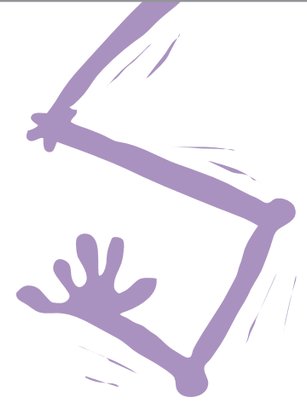
*3 … der Namespace **`System`** benötigt. **`Console`** ist ein fertiger Baustein, der innerhalb von **`System`** definiert ist. Daher musst du **`using System`** angeben, um darauf zugreifen zu können.

*4 Das Programm wartet durch die **`ReadKey`**-Funktion, bis der Benutzer eine Taste drückt, und geht erst dann zur nächsten Zeile weiter. Dadurch wird es nicht automatisch beendet, und du hast nicht das Problem, dass das Fenster nur kurz aufflackert.

*5 Hier geht es los! Alle deine Programme starten in der **`Main`**-Funktion. Diese gibt es ausnahmslos immer!



[Notiz]
Du musst wissen, dass es **dem Compiler egal** ist, ob du Tabulatoren oder Leerzeichen für die Einrückung verwendest. Dem Compiler ist sogar die Einrückung von einzelnen Zeilen selbst egal. Die Einrückungen brauchst nur du als Entwickler, damit du den **Überblick über den Programmcode** behältst und die Struktur deines Codes erkennst.



Nachdem du den angeführten Code in eine Datei **abgespeichert** hast – zum Beispiel unter dem Dateinamen **`Hallo.cs`** –, kannst du den Compiler benutzen.

[Notiz]
Für C#-Programmcode-Dateien verwendest du die Dateierweiterung **`cs`**. Die steht für »C Sharp«.

Mit einem Kommandozeilenbefehl kannst du nun den **Compiler anwerfen** und dein erstes Programm erstellen:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc /out:Hallo.exe Hallo.cs
```

Du kannst also auf jedem Computer C#-Programme schreiben und kompilieren.

[Funktioniert in]
So funktioniert es auf Windows-Rechnern, und mit einer entsprechenden Entwicklungsumgebung sogar noch einfacher. Auf der Linux-Kiste funktioniert es beispielsweise mit der Entwicklungsumgebung MonoDevelop. Und Mac willst du auch noch? Auch hier gibt es MonoDevelop für dich. Ich will jedoch beim Hersteller für C# bleiben: bei Microsoft mit **Visual Studio Community Edition**. Denn später will ich dir zeigen, wie du Windows-Store-Anwendungen entwickeln kannst, und dazu benötigst du Visual Studio.

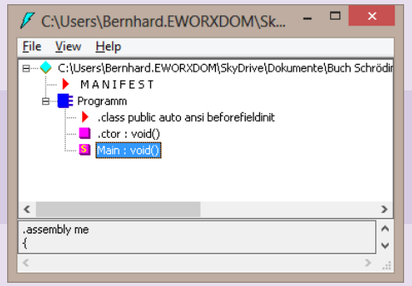
[Notiz]
Mono und C# werden übrigens auch bei Unity3D verwendet, um 3D-Spiele zu entwickeln.

Und in der EXE-Datei steht nun der IL-Code?

Ja, genau. Es gibt ein kleines Programm zum Anschauen des IL-Codes, das heißt **IL DASM – Intermediate Language Disassembly Tool**. Dieses wird automatisch mit der Entwicklungsumgebung Visual Studio von Microsoft oder auch mit dem Windows-SDK mit installiert. Da du jedoch im Allgemeinen den IL-Code bei der Softwareentwicklung nicht weiter beachtest – außer natürlich wenn du selbst einen Compiler oder Ähnliches für .NET programmieren willst – ist es auch nicht wirklich wichtig, dass du das Tool hast.

Nein, nein. Ich glaube Blizzard hat schon alle notwendigen Compiler für WoW.

Du wirst dir später Visual Studio installieren, und dann kannst du mit der Visual-Studio-Kommandozeile das Tool mit dem Befehl **`ildasm`** aufrufen. Dein Hallo-Schrödinger-Programm sieht dann so aus:

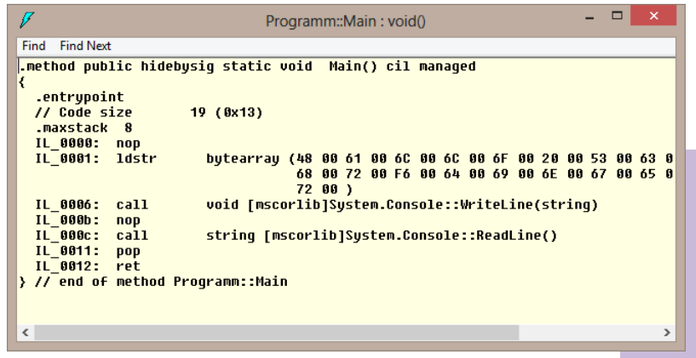


Darstellung des Hallo-Schrödinger-Programms im IL-DASM-Tool

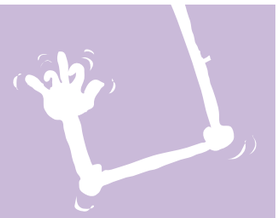
Du siehst also den Klassennamen **`Program`** und auch die Funktion **`Main`**. Außerdem wird noch ein Konstruktor dargestellt – diese Elemente wirst du später kennenlernen – und Manifest-Informationen für das Assembly.

[Notiz]
Der Konstruktor ist eine Funktion, die bei der »Geburt« von Programmelementen – sogenannten Objekten – aufgerufen wird, also dann, wenn wirklich Arbeitsspeicher dafür reserviert wird.

Du kannst in diesem Tool die **`Main`**-Funktion auswählen und dir den IL-Code der Funktion anzeigen lassen.



Der IL-Code der Main-Funktion des Hallo-Schrödinger-Programms

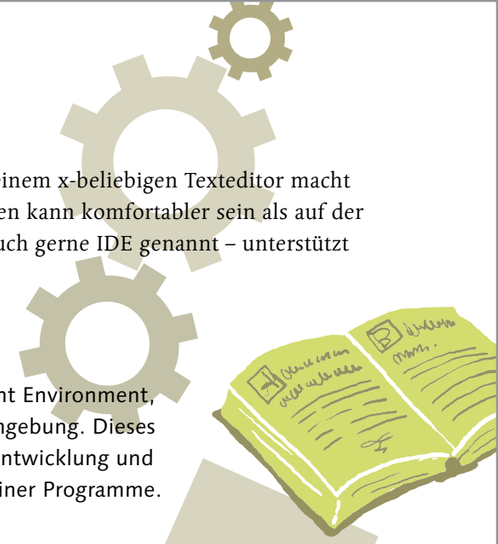


Du siehst neben verschiedenen Informationen für das Framework zur Ausführung auch wieder die Funktion **`System.Console::WriteLine`**, die in der **`mscorlib.dll`** vorhanden ist. Dieser IL-Code wird bei der Ausführung der EXE-Datei vom JIT-Compiler in richtigen Maschinencode übersetzt.

Okay, ehrlich, ich hab nichts dagegen, dass ich den IL-Code nicht ganz verstehen muss. Der sieht doch kompliziertes aus – vor allem die komischen Zahlen. Aber gut, dass ich das mal gesehen habe.

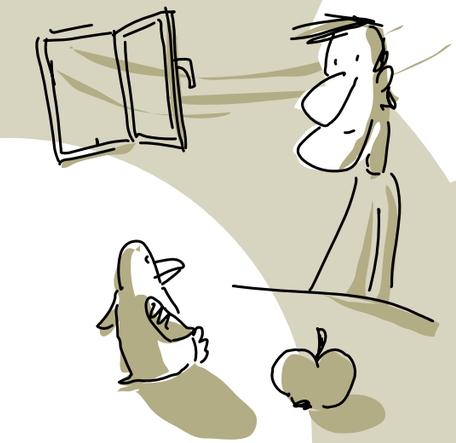
Du brauchst eine IDE!

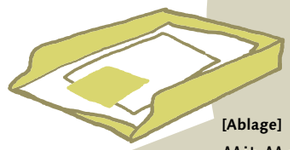
Obligatorisch ist zwar nur der Compiler, aber in einem x-beliebigen Texteditor macht programmieren keinen Spaß. Auch das Kompilieren kann komfortabler sein als auf der Kommandozeile. Die Entwicklungsumgebung – auch gerne IDE genannt – unterstützt dich bei deinen Projekten.



[Begriffsdefinition]
IDE steht für Integrated Development Environment, also eine integrierte Entwicklungsumgebung. Dieses Programm unterstützt dich bei der Entwicklung und ermöglicht auch das Kompilieren deiner Programme.

Es gibt selbstverständlich verschiedene Entwicklungsumgebungen, mit denen du C# programmieren kannst. Und du hast natürlich die Auswahl. **Selbst unter Linux** kannst du, wenn du willst, mit MonoDevelop C#-Programme entwickeln. Um Windows-Store-Anwendungen erstellen zu können, benötigst du jedoch mindestens Windows 8, und am besten eignet sich hierbei natürlich die Entwicklungsumgebung **direkt von Microsoft**. Daher schlage ich dir vor, du konzentrierst dich auf **Visual Studio**. Auch hier gibt es eine kostenlose Edition – die sogenannte **Community Edition.**

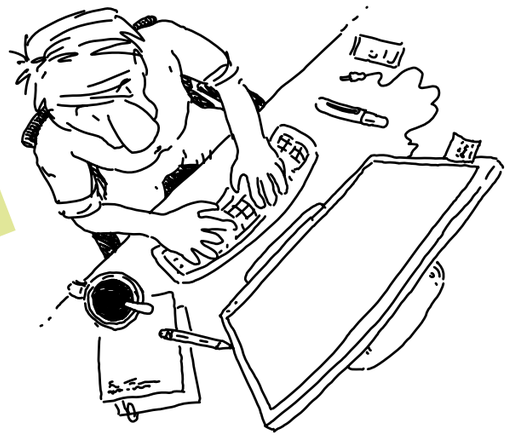




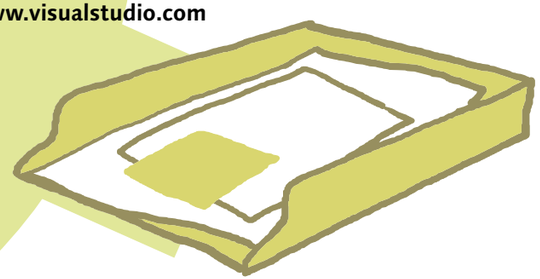
[Ablage]
Mit MonoDevelop und der .NET-Implementierung unter Linux mit dem Namen Mono kannst du auch unter Linux C#-Programme entwickeln und laufen lassen.
Auch unter Windows existieren andere IDEs als Visual Studio, zum Beispiel SharpDevelop. C# bleibt C# – die Entwicklungsumgebung macht von der Syntax her keinen Unterschied.

Visual Studio Community Edition

Visual Studio von Microsoft gibt es in verschiedenen Versionen. Die großen professionellen, jedoch kostenpflichtigen Versionen integrieren sämtliche Funktionalitäten für die Entwicklung von Desktop-, Windows 8-, Web- oder Mobile-Anwendungen, vom Einzelentwickler bis zum Riesen-Entwicklungsteam. Es steht aber auch eine sehr gute kostenlose Version zur Verfügung: Allerdings musst du je nach Anwendungstyp eine eigene Variante davon installieren. Die Visual Studio Community Edition, die den gleichen Funktionsumfang bietet wie Microsoft Visual Studio Professional.



[Ablage]
Die Entwicklungsumgebung kannst du kostenlos unter **www.visualstudio.com** herunterladen.

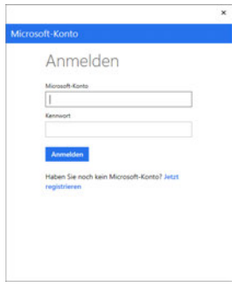


Installiere dir bitte Visual Studio, und dann geht es los.

Der Spaß geht los!

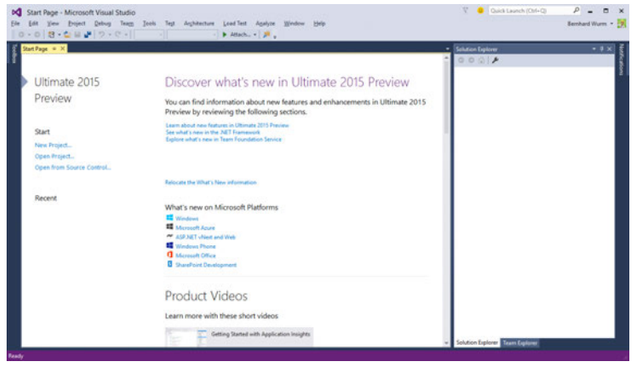
Die Software ist wie gesagt kostenlos. Du musst dich jedoch mit deinem Microsoft-Account an Visual Studio anmelden, damit es länger als 30 Tage funktioniert. Das kannst du gefahrlos machen. Einfach anmelden bzw. kostenlos registrieren, falls du noch keinen Account hast, und der Spaß kann losgehen.

Ist ja klar, dass die wieder alle meine Daten haben wollen. Aber gut – kostet ja nichts.



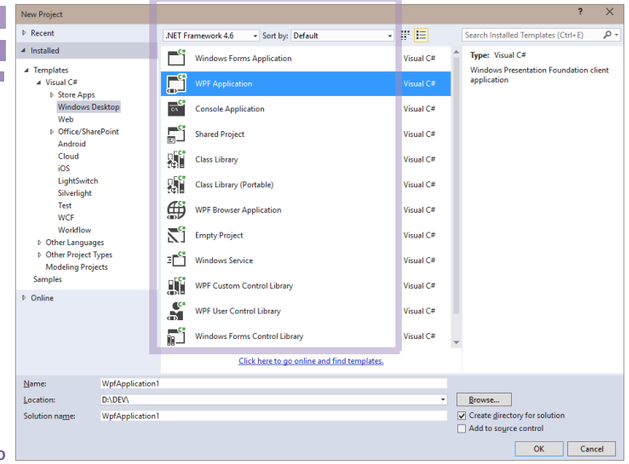
Anmelden mit dem Microsoft-Account

Nachdem du die Installation abgeschlossen hast, kannst du dein erstes Projekt starten.



Visual Studio – Startbildschirm

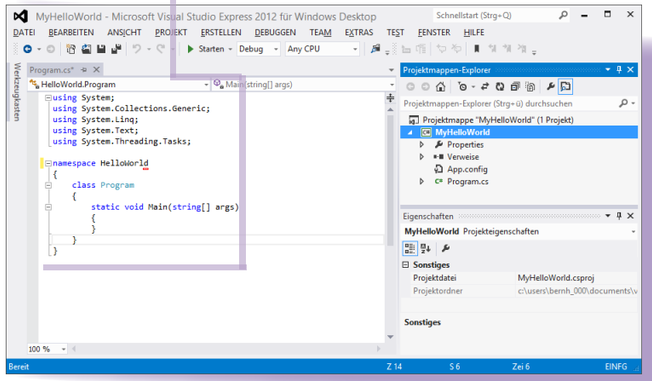
Durch einen Klick auf **`Neues Projekt`** erhältst du eine Auswahl mit den möglichen Projekttypen für diese Visual-Studio-Version. Diese Version beinhaltet alles, was du für klassische Desktopanwendungen benötigst, also ältere Windows-Forms-Anwendungen, modernere WPF-Anwendungen, Konsolenprogramme und auch Klassenbibliotheken, die dir ermöglichen, Funktionen in DLL-Dateien zu packen und projektübergreifend zu verwenden.



Auswahl der Projekttypen von Visual Studio

Am besten beginnst du zunächst mit einer **Konsolenanwendung**. Die eignet sich gut für die ersten Schritte, da du nicht von schönen bunten Oberflächen abgelenkt wirst.

Im großen Hauptfenster siehst du deinen Programmcode. Das ist dein Spielplatz, wo du an deinem Code arbeitest, ihn verfeinerst, Fehler suchst und möglicherweise manchmal fast verzweifelst. Im rechten Bereich findest du den Projektmappen-Explorer. Dieser zeigt dir alle Dateien in deiner Projektmappe an.



Grundgerüst einer Konsolenanwendung

Eine Software ist wie ein **großer Schrank**. Er sieht zunächst wie ein großes kompaktes Etwas aus, aber wenn du ihn öffnest, hast du viele verschiedene Schubladen und Fächer mit Dingen darin.

Eine Software besteht ebenfalls aus vielen einzelnen Teilen und oftmals aus **verschiedenen Projekten**. Die Projektmappe besteht also aus Projekten und ein Projekt wieder aus vielen verschiedenen Dateien und Programmcodes.
Am Ende ergibt dies zusammen deine Software.

Du erinnerst dich, dein Programm startet immer mit der **Main**-Funktion. Diese will etwas ausgefüllt werden:

```
Console.WriteLine("Bitte Namen eingeben:");
var name = Console.ReadLine(); *1
Console.WriteLine("Hallo " + name);
Console.ReadKey();
```

*1 Natürlich kannst du mit den Elementen, die der Benutzer eingibt, auch arbeiten. Dazu musst du eine Variable definieren. Die Variable ermöglicht dir, Werte im Arbeitsspeicher des Computers zu halten und damit zu arbeiten. Damit der Compiler weiß, dass es sich um eine Variable handelt, schreib einfach **var** vor den Variablennamen, den du frei wählen kannst. Später werde ich dir die verschiedenen Variablentypen zeigen, aber dazu gleich mehr. Der **Variablen** kannst du nun mittels **Console.ReadLine** einen Wert zuweisen, und zwar genau den Wert, den der Benutzer eingibt.

Wie du schon gesehen hast, kannst du mittels **Console.WriteLine** Text ausgeben. Es gibt aber auch ein **Console.ReadLine**, womit du Text einlesen kannst, den der Benutzer eingibt. Durch **Console** kannst du auf das Konsolenfenster zugreifen und entsprechend Eigenschaften setzen, wie zum Beispiel den Fenstertitel, oder auch Funktionen aufrufen, etwa zur Ein- oder Ausgabe.

Das scheint ja einfach zu sein, aber warte kurz, das will ich gleich mal ausprobieren.

Gerne. Drück einfach auf **Start**, um das Programm zu kompilieren und auszuführen.

> Bravo, Schrödinger. Du hast dein erstes Programm geschrieben.
> Und dein Programm hat dich gleich begrüßt. **Toll!**

Wenn du in Visual Studio auf den **Start**-Knopf drückst, wird automatisch der Compiler angeworfen, also die **csc.exe**, die du auch schon in der Kommandozeile aufgerufen hast. Außerdem wird das Programm gleich ausgeführt, und der JIT-Compiler springt zu deiner **Main**-Funktion, kompiliert diese und führt deinen Code nativ als Maschinencode aus.

Nativ?

[Notiz]
Nativ bedeutet, der Code wird direkt in Maschinencode von der CPU ausgeführt und nicht über weitere Umwege oder interpretierte Zwischensprachen.

Puh, der Computer treibt ganz schön viel Aufwand für eine einfache Begrüßung.

Ja, aber der Ablauf ist immer der gleiche, unabhängig davon, ob dein Programm aus 1.000.000 Zeilen Code oder lediglich drei Zeilen Code besteht.

Theorie und Praxis

[Einfache Aufgabe]
Zeig mir doch mal, ob du alles verstanden hast. Verbinde die Satzteile mit einem Bleistift.

Der C#-Compiler erzeugt …

… eine IDE.

Der JIT-Compiler erzeugt …

… gibt etwas auf der Konsole aus.

Visual Studio ist …

… liest eine Zeile von der Konsole.

Console.ReadLine …

… Maschinencode.

Console.WriteLine …

… IL-Code.

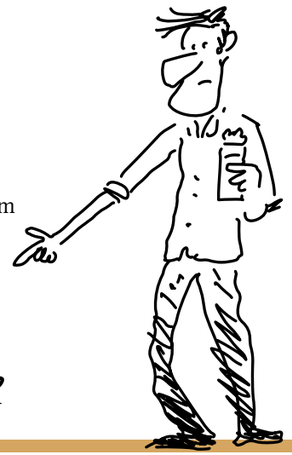
Lösung:

Der C#-Compiler erzeugt IL-Code.
Der JIT-Compiler erzeugt Maschinencode.
Visual Studio ist eine IDE.
Console.ReadLine liest eine Zeile von der Konsole.
Console.WriteLine gibt etwas auf der Konsole aus.

Wunderbar, Schrödinger. Die Theorie hast du verstanden. Und dass du die Praxis auch verstanden hast, kannst du anhand der nächsten Aufgabe beweisen:

[Schwierige Aufgabe]
Schreibe doch ein kleines Programm, das Vor- und Nachnamen nacheinander abfragt und anschließend den Namen vollständig ausgibt.

He, das Konsolenfenster öffnet sich nur ganz kurz. Was läuft da falsch?



Das Fenster schließt sich immer gleich, nachdem das Programm beendet wurde. Du hast zwei Möglichkeiten: Entweder lässt du mit `Console.ReadKey()` den Computer am Ende noch ein Zeichen einlesen, dadurch wartet das Programm bis zur nächsten Benutzereingabe und schließt somit erst anschließend, oder du startest das Programm mit **Strg + F5** bzw. über das Menü **Debuggen • Starten ohne Debugging**.

Was ist jetzt Debugging wieder?

Debuggen hilft dir bei der **Fehlersuche**. Du kannst Punkte setzen, an denen das Programm einfach stehen bleibt, damit du dir Variablenwerte ansehen kannst, usw.

Die Lösung:

```
Console.WriteLine("Vorname eingeben:");
var vorname = Console.ReadLine();
Console.WriteLine("Nachname eingeben:");
var nachname = Console.ReadLine();
Console.WriteLine(vorname + " " + nachname);
```

[Belohnung]
Wenn dein Programm funktioniert, dann kannst du jetzt gerne ein bisschen von deiner Karriere als Spieleentwickler bei Blizzard träumen.

Was gelernt!

Denkst du jetzt, du hättest noch nichts gelernt?
Dann pass mal auf, ich habe dir das Wichtigste aufgeschrieben:

- C# ist eine der vielen Programmiersprachen, die auf der .NET-Plattform basieren. Auch Visual Basic, F#, Python.NET etc. basieren auf dem .NET Framework, wobei C# nach wie vor das Flaggschiff dieser Plattform ist.
- Namespaces sind wie Schubladen. Sie helfen, die Tausenden bestehenden Programmkomponenten zu strukturieren, und das hilft auch dir, deinen Code zu strukturieren. Und so, wie du Schubladen öffnest, um Inhalte herauszunehmen, inkludierst du die Namespaces, um auf die darin vorhandenen Komponenten zuzugreifen. Der Grund-Namespace ist `System`.
- Die Intermediate Language ist die Zwischensprache, in die jede .NET-Sprache übersetzt wird. Egal, ob du C# oder Visual Basic programmierst, nach dem Kompilieren kommt IL-Code heraus.
- Visual Studio ist eine von mehreren möglichen Entwicklungsumgebungen, mit denen du C# programmieren kannst. Wenn du willst, kannst du sogar in der Kommandozeile kompilieren.
- Mit `ildasm` kannst du dir deinen Code in der Intermediate Language anschauen. Das hast du heute vermutlich zum ersten und auch gleich letzten Mal gemacht.
- Mit **Strg + F5** startest du den Debugger. Der hilft dir bei der Fehlersuche und wird noch dein Freund werden!

—ZWEI—

Datentypen und deren Behandlung

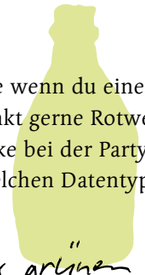
Ein netter Typ

Wie heißt es so schön, guten Freunden gibt man doch einen Kaffee – oder? Ich würde sagen, ich stelle dir ein paar freundliche Typen vor, und am Ende trinken wir eine schöne heiße Tasse Kaffee.

Das klingt gemütlich, findet Schrödinger und entspannt sich. Gemütlich? Sieben Typen allein für Zahlen? Konvertieren, kompatibel, Kommentare? Doch Schrödinger bleibt locker und lernt dabei sogar noch, mit Kamelen umzugehen.

Dieses Glas für diesen Wein

Mit den Datentypen beim Programmieren ist das ähnlich, wie wenn du eine Party schmeißt – oder Freunde zum Kaffee einlädst. Nicht jeder trinkt gerne Rotwein. Nicht jeder trinkt gerne Bier usw. So wie die verschiedenen Getränke bei der Party existieren verschiedene Datentypen, und du hast die Qual der Wahl, welchen Datentyp du deiner Variablen spendieren willst.



Danke, aber ich nehme lieber grünen Tee.

Variablen benötigst du zum Zwischenspeichern von Werten, und jede Variable hat einen bestimmten Typ, der gültige Werte festlegt. Wie bei der Party ist der Wert, den du »zwischenspeichern« willst, das Getränk an sich – zum Beispiel ein großes Bier. Damit du dieses zwischenspeichern kannst, benötigst du das Glas – deine Variable. Idealerweise ist das ein Bierglas, das groß genug ist, um einen halben Liter zu fassen. Es könnte natürlich auch eine Maß sein, allerdings wird es unschön, wenn du ein Maß-Glas für ein kleines Bier verwendest. Der Datentyp bestimmt also die Größe des Glases.

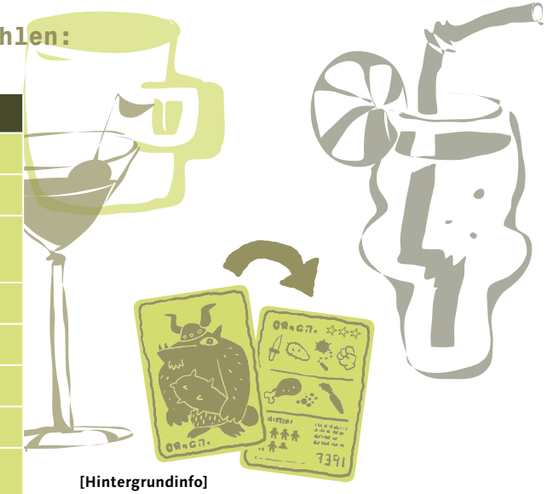
Für ganze Zahlen existieren die Datentypen **`byte`**, **`int`** und **`long`**. Wobei **`byte`** ein recht kleines Glas ist und Werte von 0 bis 255 beinhalten kann. **`int`** steht für Integer, und darin kannst du Zahlen zwischen –2.147.483.648 und 2.147.483.647 abbilden. Da geht also schon ordentlich mehr rein. Willst du größere Zahlen abbilden, nimmst du einen längeren Datentyp: **`long`**. Mit diesem kannst du dir richtig große Zahlen merken: –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807. Das ist so richtig viel!

Der **`long`**-Datentyp ist bestimmt für Bill Gates entwickelt worden, damit der seinen Kontostand abspeichern kann.

Wie gesagt sind die Datentypen wie die verschiedenen Gläser bei den Getränken. In manche Gläser kannst du mehr Inhalt hineingeben als in andere. Und da du ja Stil hast, nimmst du für Wein auch kein beliebiges Glas, sondern ein Weinglas, während du für Bier ein Bierglas verwendest usw.

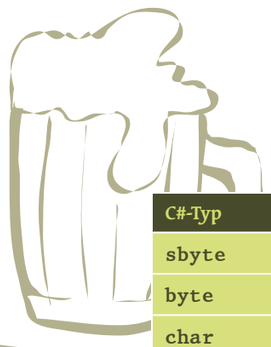
Die verschiedenen Glasgrößen für ganze Zahlen:

Typ	Größe
`sbyte`	–128 bis 127
`byte`	0 bis 255
`char`	U+0000 bis U+ffff – das ist die Zahlenrepräsentation eines Zeichens
`short`	–32.768 bis 32.767
`ushort`	0 bis 65.535
`int`	–2.147.483.648 bis 2.147.483.647
`uint`	0 bis 4.294.967.295
`long`	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`ulong`	0 bis 18.446.744.073.709.551.615



[Hintergrundinfo]

Diese Datentypen sind die Schreibweise in C#. In anderen Programmiersprachen – beispielsweise Visual Basic – schreibt man die etwas anders. Im Hintergrund gibt es aber im .NET Framework eine entsprechende gemeinsame Datenstruktur, die diesen Typ abbildet.



C#-Typ	.NET-Framework-Typ
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`char`	`System.Char`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`

Puh, das soll ich mir merken?

Keine Angst, im Prinzip ist es relativ einfach: Mit Ausnahme von **`byte`** und **`sbyte`** gibt es immer den Datentyp selbst, welcher sowohl in den negativen Bereich als auch in den positiven Bereich reicht. Mithilfe des vorangestellten **`u`** wird der Datentyp **unsigned** – also vorzeichenlos – und kann somit einen doppelt so großen Wert im positiven Bereich annehmen, da der negative ja wegfällt. Einzig **`char`** tanzt etwas aus der Reihe, da es ein Zeichen repräsentiert – also eigentlich einen Buchstaben. Um **`char`** kümmern wir uns später noch.

[Ablage]
Unsigned-Datentypen verzichten auf den negativen Wertebereich. Dadurch wird der positive Wertebereich doppelt so groß.

[Zettel]
In der Praxis werden die Unsigned-Datentypen selten verwendet.

Was du jetzt gesehen hast, sind die verschiedenen Glasgrößen für die ganzzahligen Datentypen. Benötigst du Gleitkommawerte – zum Beispiel für Preisangaben oder Ähnliches – so hast du die Auswahl zwischen **`float`** und **`double`**. Wie auch bei **`int`** und **`long`** ist das eine »Glas größer als das andere«.

Typ	Ungefährer Bereich	Genauigkeit
`float`	$-3{,}4\times10^{38}$ bis $3{,}4\times10^{38}$	7 Stellen
`double`	$5{,}0\times10^{-324}$ bis $1{,}7\times10^{308}$	15–16 Stellen
`decimal`	$\pm1{,}0\times10^{-28}$ bis $\pm7{,}9\times10^{28}$	28–29 signifikante Stellen

Die Größe des Glases bestimmt also, wie groß die Zahlenwerte sind, die du abbilden kannst. Je größer das Glas, desto mehr Platz benötigt es im Schrank. Das heißt, dass größere Datentypen mehr Speicherbedarf haben als kleinere Datentypen.

`double` benötigt doppelt so viel Speicher wie **`float`**. **`long`** benötigt doppelt so viel Speicher wie **`int`**, das doppelt so viel Speicher wie **`short`**, das wiederum doppelt so viel Speicher wie **`byte`** benötigt.

Okay, die Nachsicht ist angekommen.

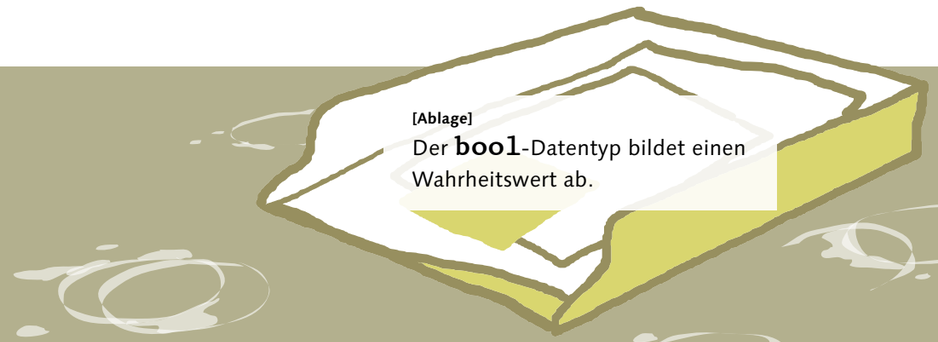
Große Datentypen sind wie große Gläser: Es passt viel hinein, aber sie brauchen auch viel Platz zum Lagern, und zwar unabhängig davon, wie viel im Glas drin ist.

[Hintergrundinfo]
Mit 1 Bit mehr im Arbeitsspeicher kann ein doppelt so großer Wertebereich abgebildet werden. Mit 2 Bit mehr sind es sogar viermal so viele mögliche Werte usw.

Damit habe ich dir die **primitiven Datentypen** vorgestellt.

[Begriffsdefinition]
Primitive Datentypen (auch als elementare oder einfache Datentypen bezeichnet) können nur Werte eines entsprechend definierten Wertebereichs aufnehmen.

`Einen habe ich noch:` Der vorerst letzte primitive Datentyp, den ich dir vorstellen möchte, ist **`bool`**. Der Boolean-Datentyp hat genau zwei gültige Werte: **`true`** und **`false`**.



[Ablage]
Der **`bool`**-Datentyp bildet einen Wahrheitswert ab.

Grundlagen im Kamelreiten

Du erinnerst dich: Die Variablen sind die Gläser für die Getränke auf deiner Party, die bestimmen, welches Getränk und in welcher Menge es eingefüllt werden kann. Ich will dir nun zeigen, wie du solche Gläser »baust«:

```
bool variable1 = true;
byte variable2 = 42;
```

Um eine Variable zu definieren, beginnst du mit dem Datentyp, anschließend mit dem gewünschten Namen der Variablen, und du kannst dieser sofort einen Wert zuweisen. Du kannst aber auch Variablen definieren, ohne diesen gleich einen Wert zuzuweisen.

Also ähnlich formal, wie sich bei den Schwiegereltern vorzustellen.

```
int variable;
```

[Achtung]
Variablen müssen vor ihrer Verwendung einen Wert zugewiesen bekommen. Andernfalls verweigert der Compiler den Code.

Ich hab das mal kurz probiert, das funktioniert aber nicht:

```
byte level in WoW = 70;
```

Das kommt daher, weil du keine Leerzeichen im Variablennamen verwenden darfst. Theoretisch könntest du aber Umlaute und ähnliche Sonderzeichen in Variablennamen benutzen, allerdings ist es gängig, darauf zu verzichten – nicht zuletzt, da nicht jeder auf der Welt die Umlaute auf der Tastatur hat. Damit du deine Variablennamen dennoch lesen kannst, ohne Leerzeichen oder Sonderzeichen zu benutzen, verwendest du **Camel Casing**.

Nein danke, ich rauche nicht.

Nein, das hat nichts mit Zigaretten zu tun. Das heißt: Wenn ein Variablenname aus mehreren Wörtern besteht, dass man alle hintereinander schreibt und jeder einzelne mit einem Großbuchstaben anfängt. Das heißt Camel-Casing, weil die großen Buchstaben wie die Höcker eines Kamels aus dem Wort ragen. Variablennamen beinhalten keine Leerzeichen oder Sonderzeichen, und der erste Buchstabe ist kleingeschrieben.

Hallo, Stell dir vor, beim Programmieren wirst du an Kamele erinnert. Du vergibst die Namen so, dass die so lustig aussehen wie Kamelhöckes...

Eine Kleinigkeit gibt es natürlich noch, auf die du aufpassen musst. Wie in jeder Programmiersprache darfst du auch hier keine Schlüsselwörter als Variablennamen verwenden. **`int`** beispielsweise ist ein Schlüsselwort, also ein von der Programmiersprache reserviertes Wort. Du darfst somit keine deiner Variablen **`int`** nennen.

[Notiz]
Schlüsselwörter werden in Visual Studio blau dargestellt.

Übungen für den Barkeeper

Da der Kaffee noch nicht ganz durch ist, kannst du ein oder zwei einfache Aufgaben erfüllen.

[Einfache Aufgabe]
Setze die Zeichen für größer als/kleiner als ein, je nachdem, welcher Datentyp einen größeren Wertebereich abbilden kann.

byte	**short**
long	**short**
float	**double**
int	**byte**
int	**ushort**

[Lösung]

byte < short
long > short
float < double
int > byte
int > ushort

Toll gemacht! Siehst du, du hast schon einen Überblick über die Datentypengrößen. Zeig doch mal, ob du die Definition von Variablen genauso gut kannst.

[Schwierige Aufgabe]
Finde mit freiem Auge die Fehler im Programmcode.

```
int myAgeIs? = 29;
bool yourAgeIs = 27;
byte size = 1870mm;
long amountOfLanguagesYouSpeak = 1.5;
```

[Lösung]
Das **?** ist nicht gültig im Variablennamen.
Der Datentyp **`bool`** darf nur **`true`** oder **`false`** als Wert besitzen.
27 ist daher kein gültiger Wert.
`byte` erlaubt lediglich Werte bis 127, und die Angabe **`mm`** ist ungültig.
Der Compiler kennt keine Einheiten.
Der Datentyp **`long`** darf nur ganze Zahlen beinhalten. Für **`1.5`** muss einer der Datentypen **`double`** oder **`float`** verwendet werden.

Rechnen mit Transvestiten

Natürlich ist die Definition und Zuweisung einer Variablen nicht alles. Du kannst mit Variablen auch arbeiten: rechnen, vergleichen, neue Werte zuweisen oder auslesen.
Du kannst gerne die Schuhgröße deiner Freundin mit Variablen berechnen. Die EU-Schuhgröße ist ca. (Fußlänge + 1,5 cm) * 1,5.

```
float fussLaenge = 24;
double mitAbstand = fuszLaenge + 1.5;
double schuhgroesse = mitAbstand * 1.5;
```

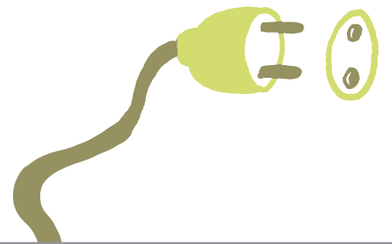
Wie du siehst, kannst du die Variablen einfach wie Zahlen verwenden. Und die klassischen Rechenoperationen sind sehr naheliegend. Erst wird der Teil rechts vom Gleichheitssymbol ausgerechnet, und dann wird das Ergebnis der Variablen auf der linken Seite zugewiesen.

Ich kann also auch ganze Zahlen Gleitkomma-Variablen zuordnen?

Ja genau. Diese werden automatisch konvertiert, genauso wie verschiedene Datentypen.

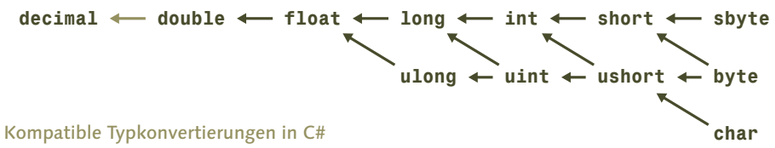
[Achtung]
Die automatische Konvertierung funktioniert immer nur von kleineren Datentypen nach größeren oder allgemeineren Datentypen.

Die Konvertierung wird wie gesagt oftmals automatisch durchgeführt, und zwar, wenn es aufgrund der Anweisung notwendig ist.



[Funktioniert in]
Werden zwei Werte verschiedener Datentypen miteinander verglichen oder damit gerechnet, so wird eine automatische Konvertierung der Variablen des kleinen Datentyps in den größeren Datentyp durchgeführt.

von	nach	nach	nach	nach	nach
byte	short	int	long	float	double
short	int	long	float	double	
int	long	float	double		
long	float	double			
float	double				



Kompatible Typkonvertierungen in C#

Du kannst aber auch explizit konvertieren. Hierzu existiert eine **`Convert`**-Klasse mit entsprechenden Funktionen, die dir die Konvertierung abnehmen.

[Hintergrundinfo]
Streng genommen ist C# rein objektorientiert, wodurch keine Funktionen, sondern lediglich Methoden existieren. Du kannst die Begriffe jedoch synonym verstehen. Da dir der Begriff Funktion aus der Mathematik geläufig und somit verständlicher sein wird, werde auch ich oft von Funktionen sprechen, wobei es wie gesagt Methoden sind.

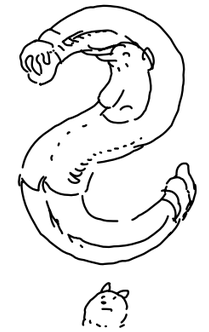
```
double pi = 3.14159265;
byte firstNumOfPi = Convert.ToByte(pi);
```

Und was passiert mit den Nachkommastellen?

Diese werden in diesem Fall abgeschnitten. Die Variable beinhaltet lediglich die Zahl **3**. Es wird auch nicht gerundet, sondern einfach abgeschnitten.

[Zettel]
Bei Konvertierungen wird niemals gerundet. Kommastellen werden kompromisslos abgeschnitten.

Und wenn der Wert in der **`double`**-Variablen größer ist als 127?



Dann wird das Programm abstürzen. Wenn du eine Konvertierung durchführst, musst du also darauf achten, dass der Wert wirklich Platz in der neuen Variablen hat. Das ist wie bei Biergläsern. Versuchst du eine Maß in ein Seiterl-Glas oder einer Kölsch-Stange zu schütten, hast du ebenfalls eine Sauerei angerichtet – statt des Programms stürzt hier das Bier ab.

[Zettel]
Sicher ist sicher! Mithilfe von **`[Datentyp].MaxValue`** kann für jeden Datentyp abgefragt werden, wie groß der maximal gültige Wert für diesen Typ ist, zum Beispiel **`int.MaxValue`**, **`double.MaxValue`** etc.

```
double max = double.MaxValue;
bool doubleIsLargerThanInt = (max > int.MaxValue);
```

[Hintergrundinfo]
Da es sich bei **`int`** und **`double`** sowie auch bei den anderen primitiven Datentypen im Hintergrund um die .NET-Datentypenstrukturen **`Int32`**, **`Int64`** usw. handelt, werden in Wahrheit die Eigenschaften der .NET-Framework-Typen verwendet. Aber das hast du dir bestimmt schon gedacht, dass dir hier C# das Leben nur einfacher machen will, damit du dich nicht mit den Framework-Typen herumschlagen musst.

Aber zurück zu den Rechnungen. Du kannst den Wert einzelner Variablen natürlich ebenfalls erhöhen:

```
int i = 0;
i = i + 1;*1
```

*1 Hier wird der aktuelle Wert der Variablen **`i`** ausgelesen, um 1 erhöht und das Ergebnis wieder als Wert der Variablen **`i`** zugewiesen. Also wird **`i`** um 1 erhöht.

Rans - erhöhen - rein. Okay.

Da dies sehr häufig vorkommt, gibt es hierzu Kurzschreibweisen.

Kurzform, um eine Variable um den Wert 1 zu erhöhen

```
int i = 0;
i  +=  1;
```

Es geht jedoch noch kürzer.

Die allerkürzeste Form, um eine Variable um den Wert 1 zu erhöhen

```
int i = 0;
i++;
```

[Achtung]
Es lassen sich natürlich nur Zahlen erhöhen. Wohin soll ein **`bool`**-Wert auch erhöht werden, wo dieser doch nur **`true`** und **`false`** als Werte kennt? Wahrer als wahr gibt es nicht.

[Begriffsdefinition]
Die Rechenoperationen nennt man auch **arithmetische Operationen**.
Eine Operation verarbeitet mehrere Operatoren zu einem neuen Wert – oder auch nur einen einzigen Operanden, dann nennt man ihn einen **unären Operator**.
Die arithmetischen Operationen in der Tabelle verarbeiten jeweils **zwei Operanden**.

Ähnliches geht natürlich auch mit Subtrahieren, Multiplizieren oder Dividieren. Hier ist ein kleiner Überblick für dich:

Rechenoperation	Anweisung	Alternative 1	Alternative 2
Addieren	`i = i + 1`	`i += 1`	`i++;`
Subtrahieren	`i = i – 1`	`i -= 1`	`i--;`
Multiplizieren	`i = i * 2`	`i *= 2`	
Dividieren	`i = i / 2`	`i /= 2`	
Divisionsrest finden (Modulo)	`i = i % 2`	`i %= 2`	

Schau dir den Modulo-Operator genau an:

```
int rest = 10 % 3;
```

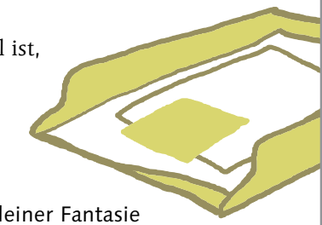
Welchen Wert hat `rest`?

Eins! Denkst du, ich kann gar nicht rechnen? 3 mal 3 ist 9, 7 ist der Rest.

Damit kannst du einen Ausdruck schreiben, der **`true`** ergibt, falls **`i`** eine gerade Zahl ist, und **`false`**, falls **`i`** eine ungerade Zahl ist:

```
(i % 2 == 0)
```

Ausdruck mit Vergleichsoperator und Modulo-Operator



[Ablage]
Und schon sind deiner Fantasie keine Grenzen mehr gesetzt. Von der Primzahlenberechnung bis hin zur … Weltherrschaft … lässt sich der Modulo-Operator verwenden.

Was ist ein Opa-Ratio?

[Begriffsdefinition]
Mithilfe von Vergleichsoperatoren vergleichst du Werte. Ihr Ergebnis ist ein boolescher Wert, **`true`** oder **`false`**. Ideal also, um boolesche Ausdrücke zu erstellen.

`==` nennt sich **Vergleichsoperator** und prüft, ob rechts und links das Gleiche steht. Das Ganze ist ein boolescher Ausdruck, weil er einen der Werte **`true`** oder **`false`** annimmt. Du kannst ihn einer booleschen Variablen zuweisen:

```
int i = 5;
bool istGerade = (i % 2==0);
Console.WriteLine("Ist 5 eine gerade Zahl?" + istGerade);
```

Und die Antwort…

…ist natürlich `false`!

Noch mehr Vergleichsoperatoren für dich:

`==`	gleich
`>`	größer
`>=`	größer oder gleich
`<=`	kleiner oder gleich
`<`	kleiner
`!=`	nicht gleich (ungleich)

Ja oder nein?

Du kannst auch boolesche Ausdrücke kombinieren und so neue Werte erstellen:

Essen = Dinkelsalat oder Schnitzel und Pommes.

Für UND-Verknüpfungen verwendest du **`&&`**.
Für ODER-Verknüpfungen verwendest du **`||`**.

Mithilfe des Nicht-Symbols **`!`** kannst du einen Wert verneinen oder invertieren:

```
bool wochentag = true;
bool wochenEnde = !wochentag;
```

Wochenende (und nicht Sonntag), das muss Samstag sein.

Der Operator **`|=`** verbindet eine ODER-Verknüpfung mit einer Zuweisung:

```
bool feiertag |= sonntag
```

Damit erhält **`feiertag`** den Wert **`true`**, falls **`feiertag`** bereits diesen Wert hat oder **`sonntag`** den Wert **`true`** hat. Dies ist die Kurzform von:

```
bool feiertag = feiertag | sonntag
```

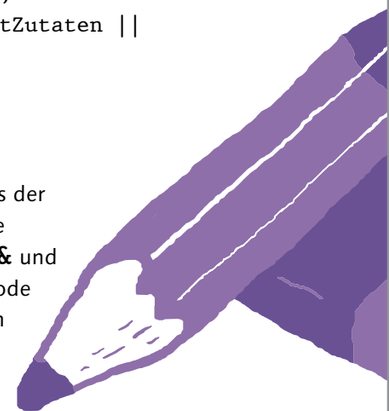
[Achtung]
Es existieren auch einfache Operatoren **`&`** und **`|`**. Allerdings handelt es sich dabei um Binär-Operatoren und **nicht um Logik-Operatoren**. Hiermit werden die **einzelnen Bits** miteinander verknüpft. Nicht verwechseln! Der Operator `^` ist das bitweise exklusive Oder.

Was gibt's zu essen?

Ein Blick in den Kühlschrank offenbart dir, was es zu essen gibt. Wenn die Zutaten für einen Dinkelsalat vorzufinden sind, dann gibt es heute Dinkelsalat. Sollte dies nicht der Fall sein, sind jedoch die Zutaten für Schnitzel und Pommes vorhanden, so hast du Glück. Fehlt beides, heißt es, erstmal einkaufen zu gehen oder die Freundin zum Essen auszuführen.

```
bool schnitzel = true;
bool pommes = false;
bool salatZutaten = true;
bool essenZuHause = salatZutaten ||
(schnitzel && pommes);
```

[Notiz]
Der **`&&`**-Operator hat eine höhere Priorität als der **`||`**-Operator. Werden diese Operatoren ohne Klammerung verwendet, wird also erst der **`&&`** und erst später der **`||`** ausgewertet. Damit der Code einfacher zu lesen ist, empfehle ich dir jedoch ohnehin, Klammern zu verwenden.



In diesem Fall musst du nicht einkaufen gehen oder deine Freundin zum Essen ausführen. Immerhin sind die Zutaten für den Salat vorhanden.

Oh... ja... leckes Salat...

.NET ist hier **sogar so effizient, dass es gar nicht mehr prüft, ob Schnitzel und Pommes vorhanden sind**, weil dadurch, dass die Dinkelsalatzutaten vorhanden sind, das Ergebnis ohnehin nur noch **`true`** sein kann. Das nennt sich **Kurzschlussauswertung**. Sobald das Ergebnis einer booleschen Abfrage unausweichlich bekannt ist, wird der Rest nicht mehr ausgewertet.

So gesehen hat mein Schnitzel kaum eine Chance, weil immer zuerst auf den Salat geprüft wird. Och no.

Wie geht es eigentlich unserem Kaffee? Ist der inzwischen durch oder können wir noch kurz üben?

Das ständige Hin und Her zwischen ja und nein

[Schwierige Aufgabe]
Was ist das Ergebnis dieser Auswertung?

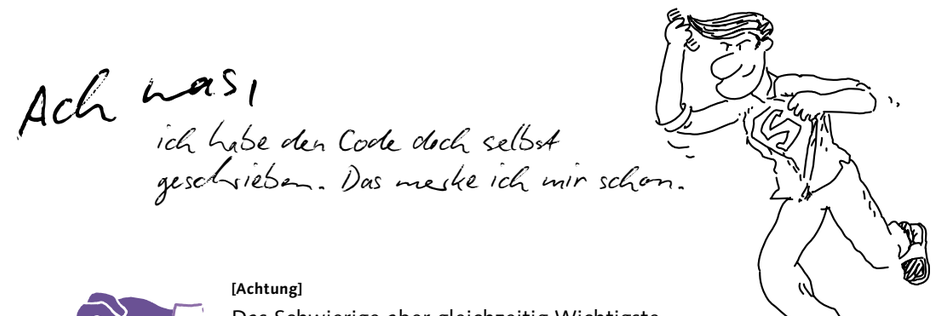
```
bool ergebnis = false || true;
ergebnis |= false;
ergebnis = !ergebnis;
ergebnis |= !true;
```

[Lösung]
Nach der ersten Zeile **`true`**
Nach der zweiten Zeile ebenfalls **`true`**
Nach der dritten Zeile **`false`**
Nach der vierten Zeile **`false`**

Gut kommentieren!

Bevor der Kaffee endgültig fertig ist, habe ich noch etwas für dich: Verwende Kommentare in deinem Programmcode! In dem Moment wo du das Programm schreibst, ist dir zwar klar, warum du es so schreibst, wie du es schreibst, aber glaube mir, nach wenigen Stunden oder Tagen weißt du es häufig nicht mehr und du wünschst dir, du hättest deinen Code besser kommentiert.

Verwende einzeilige Kommentare mithilfe von **`//`**. Oder mehrzeilige Kommentare mit **`/* Das ist ein Mehrzeiliger Kommentar */`**.



[Achtung]
Das Schwierige aber gleichzeitig Wichtigste ist, nicht das Offensichtliche zu kommentieren, sondern warum der Code so ist, wie er ist. Kommentiere also nicht, WAS da steht, sondern WARUM es da steht.

Eine kleine Hilfestellung für dich ist das Schlüsselwort **`var`**. Dieses ersetzt jeden beliebigen Datentyp. Der Compiler versucht für dich herauszufinden, welcher Datentyp der richtige ist und verwendet automatisch diesen für dich. Wenn der Compiler nicht herausfinden kann, welcher Datentyp korrekt ist, funktioniert das Schlüsselwort nicht, und du bekommst einen Compilerfehler.

[Achtung]
Beim Schlüsselwort **`var`** gilt: Lieber sparsam einsetzen. Es zeugt von gutem Stil, wenn man den Datentyp angibt und dieser somit ganz klar erkennbar ist.

Kommentare im Einsatz

So verwendest du Kommentare an beliebigen Stellen im Programmcode. Ob du gerne einzeilige oder mehrzeilige verwendest, ist dir überlassen und häufig eine reine Geschmacksfrage.

```
/* Viele boolesche Verknüpfungen
    Was ist am Ende das Ergebnis? */
bool ergebnis = false || true;  // Ergibt true
ergebnis |= false;              // Bleibt true
ergebnis = !ergebnis;           // Wird false
ergebnis |= !true;              // Bleibt false
```

Andere für sich denken lassen

Wie gesagt kannst du die Wahl des Datentyps gerne dem Compiler mit dem Schlüsselwort **`var`** überlassen:

```
var boolVariable = true;
var zahl = 3;
var doubleVariable = 3.4;
var floatVariable = 3.4f;
var longVariable = 42L;
```

Damit eine konstante Zahl als **`float`** erkannt wird, muss der Zahl ein großes **`F`** oder ein kleines **`f`** nachgestellt werden. Ansonsten wird die Zahl immer als **`double`** erkannt und behandelt.

Wenn mir der Compiler Arbeit abnimmt, hab ich da überhaupt nichts dagegen.

[Achtung]
Sobald der Datentyp festgelegt ist, kann dieser nicht geändert werden, auch mit **`var`** nicht.

Compiler-Spiele

Schrödinger, ich möchte kurz überprüfen, ob du das jetzt alles verstanden hast.

[Einfache Aufgabe]
Spiele doch einmal Compiler, und sag mir, welche Datentypen hier angenommen werden.

```
bool x = true;
bool y = false;
var wert1 = x || y;
var wert2 = 42;
var wert3 = 3.14159265;
```

[Lösung]
`bool`, **`int`**, **`double`**

Sehr gut gemacht.

[Schwierige Aufgabe]
Das ist jetzt noch etwas interessanter. Wie sieht es hier aus?

```
var wert4;
var wert7 = 32 + 17;
var wert5 = wert2 + wert3;
var wert6 = 1.2F * wert3;
```

*Keine Ahnung, was für einen Datentyp **wert4** bekommt. Der hat ja noch keinen Wert.*

Genau. Und daher ist das auch nicht gültig.
Hier bekommst du einen Compilerfehler. Richtig!

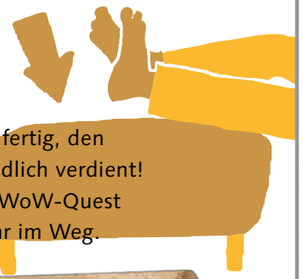
*Aber welcher Datentyp wird für **wert5** angenommen, wenn **wert2** ein **int** und **wert3** ein **double** ist?*

Ich helfe dir etwas weiter. Erinnere dich an die automatischen Typkonvertierungen. Der Compiler wendet diese hier an und ermittelt so das Ergebnis.

[Lösung]
`int` – beides sind **`int`**-Werte,
`double - int` + **`double`** ergibt eine automatische Konvertierung zu **`double`**,
`double - float` * **`double`** ergibt eine automatische Konvertierung zu **`double`**.

Viele neue Freunde

Du hast jetzt wirklich viele neue Typen kennengelernt und weißt, wer mit wem in welchen Cliques abhängt.



[Belohnung]
Endlich ist der Kaffee fertig, den haben wir uns jetzt redlich verdient! Und deiner nächsten WoW-Quest steht auch nichts mehr im Weg.

- Die Datentypen, die du in C# durch die Schlüsselwörter verwendest, werden beim Kompilieren in die sprachunabhängigen .NET-Framework-Datentypen übersetzt.
- **`byte`**, **`short`**, **`int`** und **`long`** sind die Datentypen, mit denen du ganze Zahlen abbildest, und zwar sowohl im positiven als auch im negativen Wertebereich. Die Reihenfolge, die ich dir hier gezeigt habe, reicht vom kleinen Wert zum großen Wert. Und jeder Datentyp benötigt doppelt so viel Speicherplatz – wovon du bestimmt genug besitzt.
- **`float`**, **`double`** und **`decimal`** sind die Datentypen für die Gleitkommazahlen. Wenn du also nicht nur mit ganzen Euros, sondern auf den Cent genau arbeiten möchtest.
- Zu den Zahlendatentypen gibt es jeweils Unsigned-Varianten. Bei diesen wird der negative Wertebereich einfach weggelassen, wodurch sich der positive Wertebereich um das Doppelte erhöht, wenn du es ganz genau nimmst um das Doppelte + 1.
- Bei Bedarf werden die Datentypen automatisch in die größeren konvertiert, wodurch du beispielsweise **`long`**-Variablen mit **`int`**-Variablen addieren kannst. Hierzu wird der Wert der **`int`**-Variablen kurzerhand in ein **`long`** übernommen und anschließend addiert. Das passiert aber im Hintergrund für dich.
- Der **`bool`**-Datentyp enthält einen der Wahrheitswerte **`true`** oder **`false`**.
- Mit den Zahlenwerten kannst du rechnen, addieren, subtrahieren und was dir sonst noch einfällt. Sogar die Modulo-Rechnung für den Rest ist möglich. Selbstverständlich ist es für dich ein Leichtes, Werte miteinander zu vergleichen (**`==`**) oder einer Variablen den Wert einer anderen zuzuweisen (**`=`**).
- Für das Rechnen und Zuweisen in einem Schritt gibt es kürzere Schreibweisen – Programmierer sind irgendwie schreibfaul –, also **`+=`**, **`-=`**, **`/=`** und **`%=`**, aber auch **`++`** und **`--`**.
- Und zu guter Letzt – ich befürchte du wirst diesen Rat ignorieren – habe ich dir gezeigt, wie du deinen Code mit Kommentaren versehen kannst (**`//`** und **`/* */`**), was du ausreichend tun sollst. Denn die Kommentare sollen dir auch nach Monaten noch ermöglichen, zu verstehen, was du dir damals beim Programmieren gedacht hast.

INHALTSVERZEICHNIS

Kapitel 1: Ein guter Start ist der halbe Sieg

Compiler und Entwicklungsumgebungen

Schrödinger steht am Anfang seines neuen Jobs als C#-Entwickler. Sein Problem: Er kann noch gar kein C#. Sein erster Schritt zur Lösung: Er hat sich Hilfe geholt. Und er hat richtig Lust auf die Sache. Beste Voraussetzungen also. Jetzt ist der zweite Schritt an der Reihe: Installieren! Aber was?

Kapitel 2: Ein netter Typ

Datentypen und deren Behandlung

Wie heißt es so schön, guten Freunden gibt man doch einen Kaffee – oder? Ich würde sagen, ich stelle dir ein paar freundliche Typen vor, und am Ende trinken wir eine schöne heiße Tasse Kaffee. Das klingt gemütlich, findet Schrödinger und entspannt sich. Gemütlich? Sieben Typen allein für Zahlen? Konvertieren, kompatibel, Kommentare? Doch Schrödinger bleibt locker und lernt dabei sogar noch, mit Kamelen umzugehen.

Kapitel 3: Alles unter Kontrolle

Bedingungen und Schleifen

Schrödinger findet zwar Datentypen ganz in Ordnung, jedoch dämmert ihm, dass er mit ein paar Variablen und der Ausgabe auf der Konsole wohl noch keine Spieleentwicklerkarriere starten kann. Beim Spielen gibt es so viele Entscheidungen zu treffen. Angreifen oder abhauen? Stehen oder gehen? Diese Entscheidungen müssen doch auch mit C# programmiert werden können! In diesem Kapitel gewinnt Schrödinger die Kontrolle über alle Abläufe im Code.

Kapitel 4: Sexy Unterwäsche – von kleinen Teilen bis gar nichts

Strings, Characters und Nullable Types

Selbst das kleinste Programm arbeitet mit Zeichenketten. Es wird Zeit, dass Schrödinger den Datentyp dafür kennenlernt: string. Aber wer den Buchstaben nicht ehrt, ist des Strings nicht wert. Schrödinger schaut sich auch einzelne Zeichen ganz genau an. Was er nicht gedacht hätte: Es gibt einen engen Zusammenhang zu Zahlen. Jetzt schaut er noch genauer hin und lernt sogar, mit nichts umzugehen, rein gar nichts – mit Variablen, die rein gar keinen Wert haben.

Kapitel 5: Eine endliche Geschichte

Enumerationen

Allzu oft kommt es vor, dass Schrödinger nicht nur mit Zahlen und Texten arbeiten möchte, sondern mit bestimmten Auswahlmöglichkeiten, mit einer von vielen. Wie bei einer Ampel, die genau Grün, Gelb und Rot zeigt, oder wie die Völker bei World of Warcraft, wobei die ja immer wieder mal erweitert werden. Wie auch immer, ein Konstrukt muss her, das Auswahlmöglichkeiten erlaubt, ohne sie unbedingt in Zahlen abzubilden. Da gibt es doch etwas … Enumerationen.

Kapitel 6: Teile und herrsche

Methoden

Schrödingers Code wird immer länger, und für ihn wird es auch immer schwieriger, den Überblick zu behalten. Ganz zu schweigen davon, immer gute Variablennamen zu finden, die er selbst nicht schon vergeben hat. Schrödinger wird nun also lernen, wie er seinen Code mithilfe von Funktionen – oder formal etwas richtiger: Methoden – besser strukturieren und Teile davon sogar wiederverwenden kann. Copy & Paste von Codeteilen gehört ab jetzt der Vergangenheit an.

Kapitel 7: Klassengesellschaft

Objekte, Eigenschaften und Sichtbarkeiten

Schrödinger kann zwar Methoden schreiben und doppelten Code vermeiden, doch so wirklich will sich keine Übersichtlichkeit einstellen. Es muss doch möglich sein, Programmcode auf mehrere Dateien aufzuteilen und Variablen nicht immer »global« definieren zu müssen, egal ob sie nur in bestimmten Bereichen benutzt werden. Das schreit nach Objekten! In Objekten kommt zusammen, was zusammengehört: Eigenschaften und Methoden, gekapselt in einem Objekt und mit ein bisschen Geheimniskrämerei.

Kapitel 8: Es wird Zeit für Übersicht!

Namespaces

Es gibt Tausende Klassen im .NET-Framework, und auch in Softwareprojekten sind es gerne Hunderte. Daher schadet es nicht, dass Schrödinger das Konzept der Namespaces kennenlernt. Nicht nur um die eigenen Klassen zu strukturieren, sondern auch um zu verstehen, wo er welche Funktionalitäten aus dem Framework suchen muss.

Kapitel 9: Erben ohne Sterben

Objektorientierte Programmierung

Seite 179

Dass er doppelten Code vermeiden soll, ist für Schrödinger schon lange nichts Neues mehr. Er kennt aber längst noch nicht alle Tricks dazu. Jetzt nimmt er sich Vererbung und Polymorphismus vor – wichtige Konzepte der Objektorientierung. Damit kann man doppelten Code vermeiden (ach, was!) und viele Funktionen einfach geschenkt bekommen.

Kapitel 10: Abstrakte Kunst

Abstrakte Klassen und Interfaces

Seite 201

Jetzt hat Schrödinger Vererbung und Polymorphismus verstanden und darf sich immer noch nicht »Meister der objektorientierten Programmierung« nennen. Warum nicht? Sein Kumpel kommt immer wieder auf das Thema Copy & Paste zu sprechen. Schrödinger soll doppelten Code noch konsequenter vermeiden: Gemeinsamkeiten zusammenfassen und durch Interfaces und abstrakte Klassen abbilden. Erst dann darf er sagen, dass er die Konzepte der objektorientierten Programmierung kennt.

Kapitel 11: Airbags können Leben retten

Exceptionhandling

Seite 225

Wenn bloß Schrödingers Programme nicht immer gleich abstürzen würden, sobald der Benutzer etwas eingibt, das er nicht sollte! Das wäre schon eine feine Sache. Beim Programmieren gibt es immer Ausnahmezustände, mit denen man rechnen muss. Deshalb heißt es: Mit Fehlern umgehen lernen. Erstens, die Konzepte kennenlernen, die C# dafür zu bieten hat. Zweitens, dann natürlich dran denken. Also, liebe Leser, lieber Schrödinger: Fehlerbehandlung nicht vergessen!

Kapitel 12: Ein ordentliches Ablagesystem muss her

Collections und Laufzeitkomplexität

Seite 249

Arrays sind zwar nett, aber oftmals auch sehr unflexibel. Vor allem die fixe Größe gefällt Schrödinger überhaupt nicht und treibt ihn regelmäßig an den Rand des Wahnsinns. Eine Alternative zu Arrays muss her! Eine, die das Leben vereinfacht, in der Schrödinger eine beliebige, bei der Definition noch unbekannte Anzahl von Elementen verwalten kann. Und schnell soll sie sein, diese Alternative. Sind Schrödingers Anforderungen der Wunsch nach einer Eier legenden Wollmilchsau?

Kapitel 13: Allgemein konkrete Implementierungen

Generizität

Schrödinger hat bereits bei den Collections ein bisschen mit generischen Typen gearbeitet. Doch noch weiß er nicht wirklich, was sich dahinter verbirgt und dass er selbst generische Typen und generische Methoden programmieren kann. Was ihm eine ganz neue und spannende Welt eröffnet.

Kapitel 14: Linke Typen, auf die man sich verlassen kann

LINQ

Algorithmen sind für Schrödinger manchmal ganz schön aufwendig, und wenn es knifflig wird, neigt er zur Ungeduld. Gäbe es doch etwas, das das Suchen von Elementen, das Abgleichen von Listen oder Umwandeln einer Liste von Elementen in andere Typen einfacher macht. Das wäre wirklich vorteilhaft! Und tatsächlich gibt es so etwas. Es heißt LINQ – Language-Integrated Query.

Kapitel 15: Blumen für die Dame

Delegaten und Ereignisse

Manchmal wünschen wir uns alle, bestimmte Arbeiten einfach delegieren zu können. Ginge es im Leben doch auch nur so einfach wie beim Programmieren. Schrödinger wird bestimmt bald versuchen, die Delegaten, die er nun in C# kennenlernen wird, ins echte Leben zu übertragen und viel Arbeit an seine Freundin abzugeben. Hoffentlich kommen dann nicht ungeahnte Ereignisse auf ihn zu.

Kapitel 16: Der Standard ist nicht genug

Extension-Methoden und Lambda-Expressions

Schrödinger hat Gefallen gefunden an LINQ und den anderen Konzepten, die er bis jetzt kennengelernt hat. Aber sein Ausbilder hat ihn besser durchschaut als er sich selbst: Er hätte es gerne kürzer. Noch weniger zu tippen, ja, gar nicht zu tippen, wäre eigentlich am schönsten. Ob es da etwas gibt?

Kapitel 17: Die Magie der Attribute

Arbeiten mit Attributen

Seite 391

»Attribute sind kleine, nette, zusätzliche Elemente, die an Datentypen, Methoden oder Eigenschaften hängen. Sie leben in Symbiose mit deinen Klassen und deren Elementen und wie Fabelwesen sind sie nicht direkt sichtbar, sondern nur unter bestimmten Umständen. Nämlich genau dann, wenn du einen Typ (eine Klasse) selbst ganz genau betrachtest.« Und dann soll es auch noch um Psychologie gehen, und um Magie. Schrödinger ist skeptisch. Ob der Bernhard das alles ernst meint?

Kapitel 18: Ich muss mal raus

Dateizugriff und Streams

Seite 413

Schrödinger denkt bei jedem Programm an WoW und daran, was ihm an Fähigkeiten noch fehlt, um bei Blizzard punkten zu können. Da fällt ihm auf, dass er noch gar keine Spielstände speichern könnte, da er nicht weiß, wie er auf Dateien zugreift, Daten auf der Festplatte speichert oder auch eine Datei aus dem Internet herunterladen kann. Das sollte doch ganz einfach funktionieren. Nur wie?

Kapitel 19: Sag doch einfach, wenn du fertig bist

Asynchrone und parallele Programmierung

Seite 463

Schrödinger hat bereits gemerkt, dass Dinge, die mit dem Download von Dateien zu tun haben, lange dauern. Aber auch Algorithmen können ihre Zeit brauchen. Gleichzeitig hat sein PC mehrere Kerne, die sich meistens langweilen. Das muss sich doch zusammenbringen lassen? Er hat gehört, dass parallele und asynchrone Programmierung nicht so einfach sind, doch mit den richtigen Tricks ist es plötzlich gar nicht mehr schwer.

Kapitel 20: Nimm doch, was andere schon gemacht haben

Die Paketverwaltung NuGet

Seite 513

Beim Programmieren ist es doch so, dass die meisten Probleme schon von anderen Entwicklern gelöst wurden. Oftmals gießen diese die Lösung dann in fertige Bibliotheken, die nur darauf warten, genutzt zu werden. Das wird Schrödinger bestimmt gefallen.

Kapitel 21: Die schönen Seiten des Lebens

Einführung in XAML

Seite 521

Auf die Dauer werden Konsolenanwendungen öde. Auch wenn Schrödinger geduldig die Programmierkonzepte von C# gelernt hat, wird es Zeit, sich der Oberflächenprogrammierung zu widmen. Endlich Fenster und Schaltflächen für seine Programme! Die XAML-Technologie scheint da genau das Richtige zu sein! Wird sie doch sowohl für Silverlight, Windows, Windows Phone und Windows-Store-Anwendungen verwendet. Doch wie in diese große neue Welt am besten eintauchen?

Kapitel 22: Models sind doch schön anzusehen

Das Model-View-ViewModel-Entwurfsmuster

Seite 569

Die Trennung von Code und Design ist ein wichtiges Konzept, so viel ist klar. Dies kann bereits mit den Code-Behind-Dateien erreicht werden. Aber für große Anwendungen mit WPF und XAML gibt es etwas Besseres: Das Entwurfsmuster Model-View-ViewModel, MVVM. Sieht schon mal schön symmetrisch aus. View steht bestimmt für Design. Das braucht Schrödinger also unbedingt, sonst kann er sich nicht Profi nennen.

Kapitel 23: Stereotyp Schönheit

Windows-Store-Apps

Seite 621

Microsoft lässt nicht jede beliebige App in den Windows Store. Wie schon zu Zeiten von Windows 95 sollen auch bei Windows-Store-Apps die Anwendungen alle ähnlich zu bedienen sein. Nur haben sich die Regeln für die Designerstellung signifikant verändert. Hierzu gibt es Designrichtlinien, an die sich Schrödinger zu halten hat. Doch Microsoft sei Dank gibt es verschiedene Layout-Templates, die vieles davon bereits vorgeben.

Kapitel 24: Charmante Möglichkeiten

Charms für Windows-Store-Apps

Seite 647

Windows erlaubt es, die Suchfunktion der eigenen App in die Windows-Suche zu integrieren. Außerdem erlaubt Windows das Teilen von Informationen zwischen verschiedenen Windows-Store-Apps über den Share Charm, wenn die App diese Windows-Funktionen implementiert. Das hört sich für Schrödinger alles sehr spannend an, doch wie geht das?

Kapitel 25: Live is Live

Die Verwendung von Live-Kacheln

Seite 689

Viele Windows-Store-Anwendungen besitzen nicht nur Kacheln, sondern sogar Live-Kacheln. Diese zeigen für den Benutzer relevante Informationen an und führen dazu, dass er die Anwendung häufiger benutzt. Hört sich an, als würde sich der Aufwand auch für Schrödinger lohnen, Live-Kacheln einzubauen.

Kapitel 26: Ich will alles rausholen

Datenzugriff über die Windows API

Seite 705

Prinzipiell weiß Schrödinger bereits, wie er auf Dateien zugreifen kann. Aber Windows-Store-Anwendungen sind ja sehr abgeschottet, Sicherheit wird ganz großgeschrieben. Deshalb wird es dort bestimmt nicht ganz so einfach. Alles neu lernen für die Store-Apps muss er aber nicht. Nur ein bisschen was dazulernen. Aber muss man das nicht sowieso immer?

Kapitel 27: Funktioniert das wirklich?

Unit-Testing

Seite 723

Es ist immer wieder das Gleiche mit Fehlern. Kaum hat man einen gefunden, wurden zwei neue produziert, oder es tauchen Fehler an einer ganz anderen Stelle auf. Dann heißt es, alles noch einmal testen und immer wieder von vorn, langweilig und lästig. Wäre das nicht etwas für einen Computer? Könnte Schrödinger nicht ein Programm schreiben, das seine Programme testet?

Kapitel 28: Auf ins Kaufhaus!

Das Publizieren im Windows Store

Seite 741

Das Wichtigste ist, dass die App auch in den Windows Store geladen wird. Denn ansonsten steht sie der Welt nicht zur Verfügung. Und jetzt ist es so weit, sich darum zu kümmern. Und doch gibt es auch hier ein paar Fallstricke, die Schrödinger berücksichtigen muss, damit die Annahme seiner App im Store nicht verweigert wird.

Index

D

E

F

G

Q

R

S

T

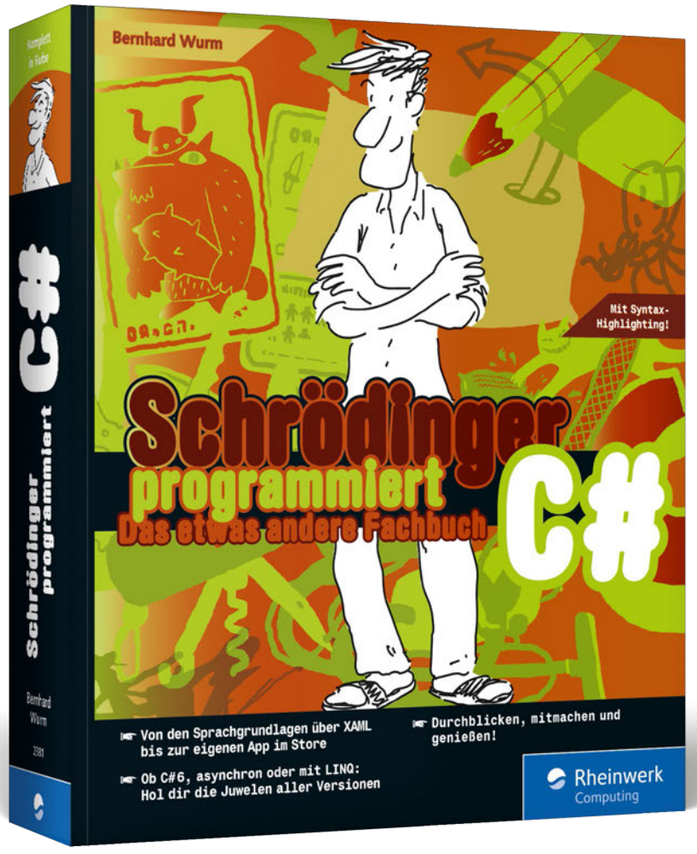
U

V

W

X

Z



Bernhard Wurm ist Softwareentwickler aus Leidenschaft und kann sich noch sehr gut an seine ersten Programmierversuche erinnern. Er weiß, dass gute Nerven und Ausdauer genauso gefragt sind wie Neugierde und Spaß am logischen Denken. Auf jeden Fall hat ihn noch kein Programm in die Knie gezwungen; er studierte prompt Information Engineering und Management sowie Software Engineering. Inzwischen leitet er die Software-Entwicklung in einem österreichischen Unternehmen. Seine Interessen sind breit gestreut, aber immer computer-affin: Sie reichen von Distributed Computing über die digitale Bilderverbeitung und Software Architekturen bis hin zu den neuesten Technologien.

Bernhard Wurm

Schrödinger programmiert C# – Das etwas andere Fachbuch

760 Seiten, broschiert, in Farbe, März 2015
49,90 Euro, ISBN 978-3-8362-2381-2

www.rheinwerk-verlag.de/3366