Bernhard Wurm

Inkl. Downloads

Mit Syntax-Highlighting!

Schrödinger programmiert C#

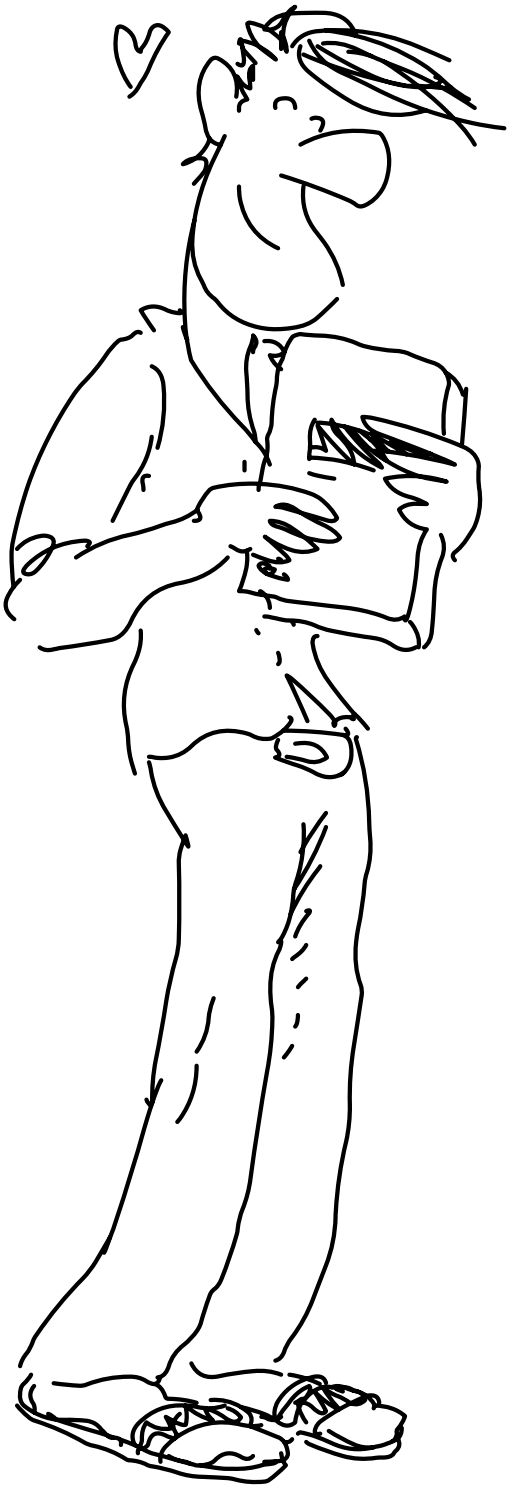
Das etwas andere Fachbuch

- Von den Sprachgrundlagen über XAML bis zur komplexen Anwendung
- Ob LINQ, Generics oder C#8: Hol dir die Juwelen aller Versionen!
- Durchblicken, mitmachen und genießen!

DRITTE AUFLAGE

Rheinwerk Computing

MIT TALENT, KATZEN-
PHOBIE 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:

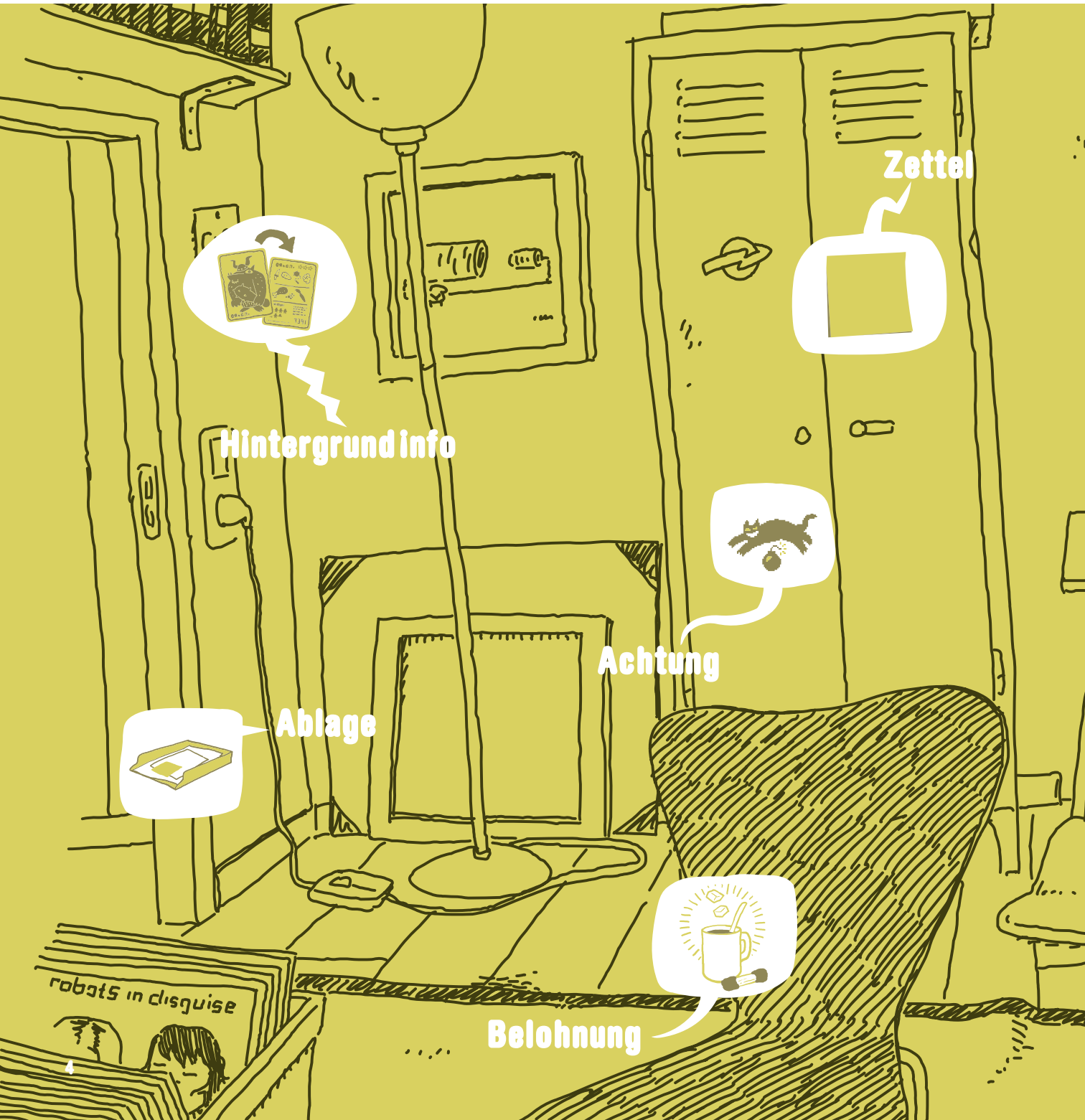
C#.

Da sind wir dabei.

Lernen müssen Sie zwar selbst, aber wir geben unser Bestes, um Sie zu unterstützen:

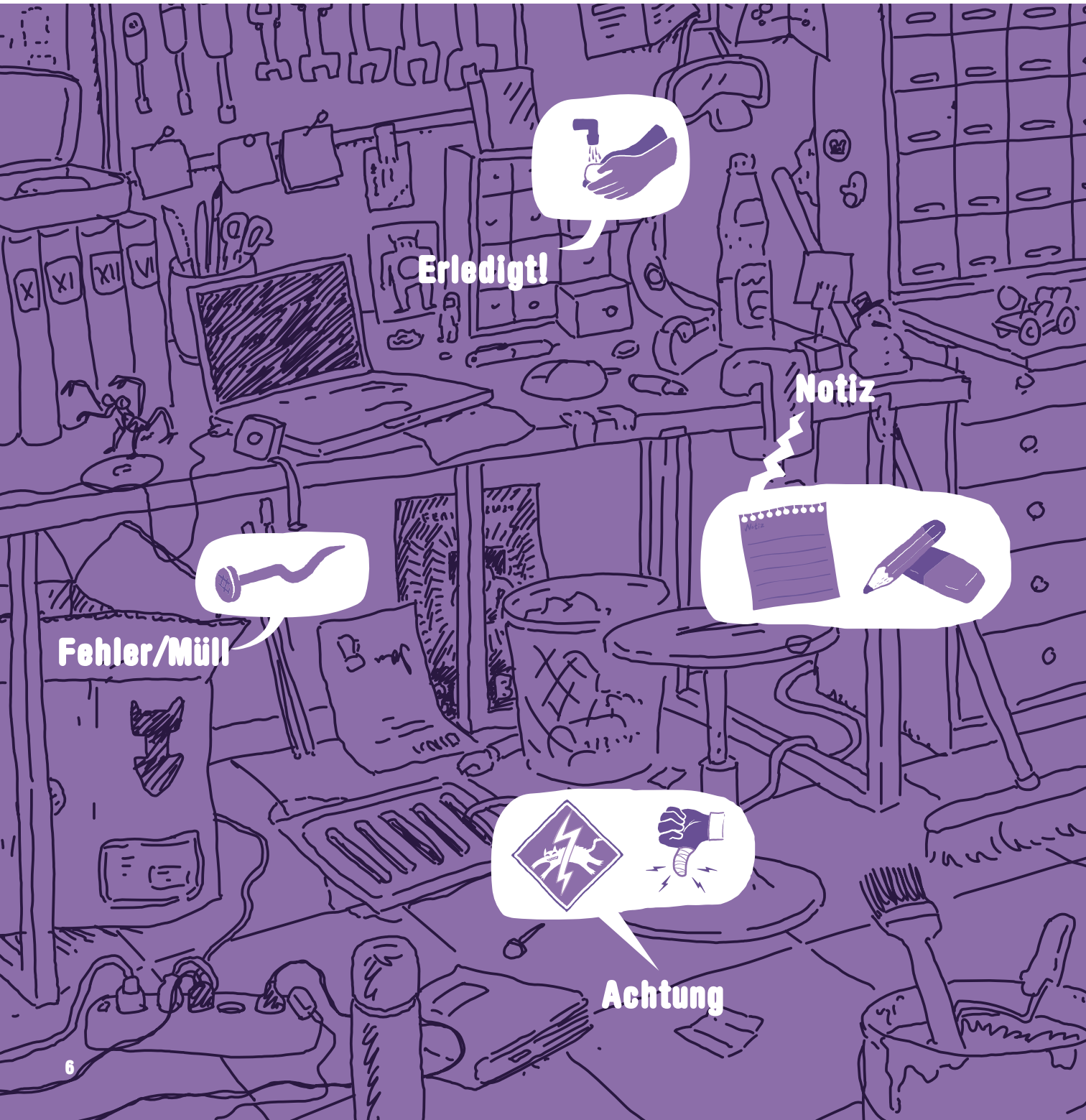
Na dann: Viel Erfolg!

Schrödingers Büro



Die nötige Theorie, viele Hinweise und Tipps

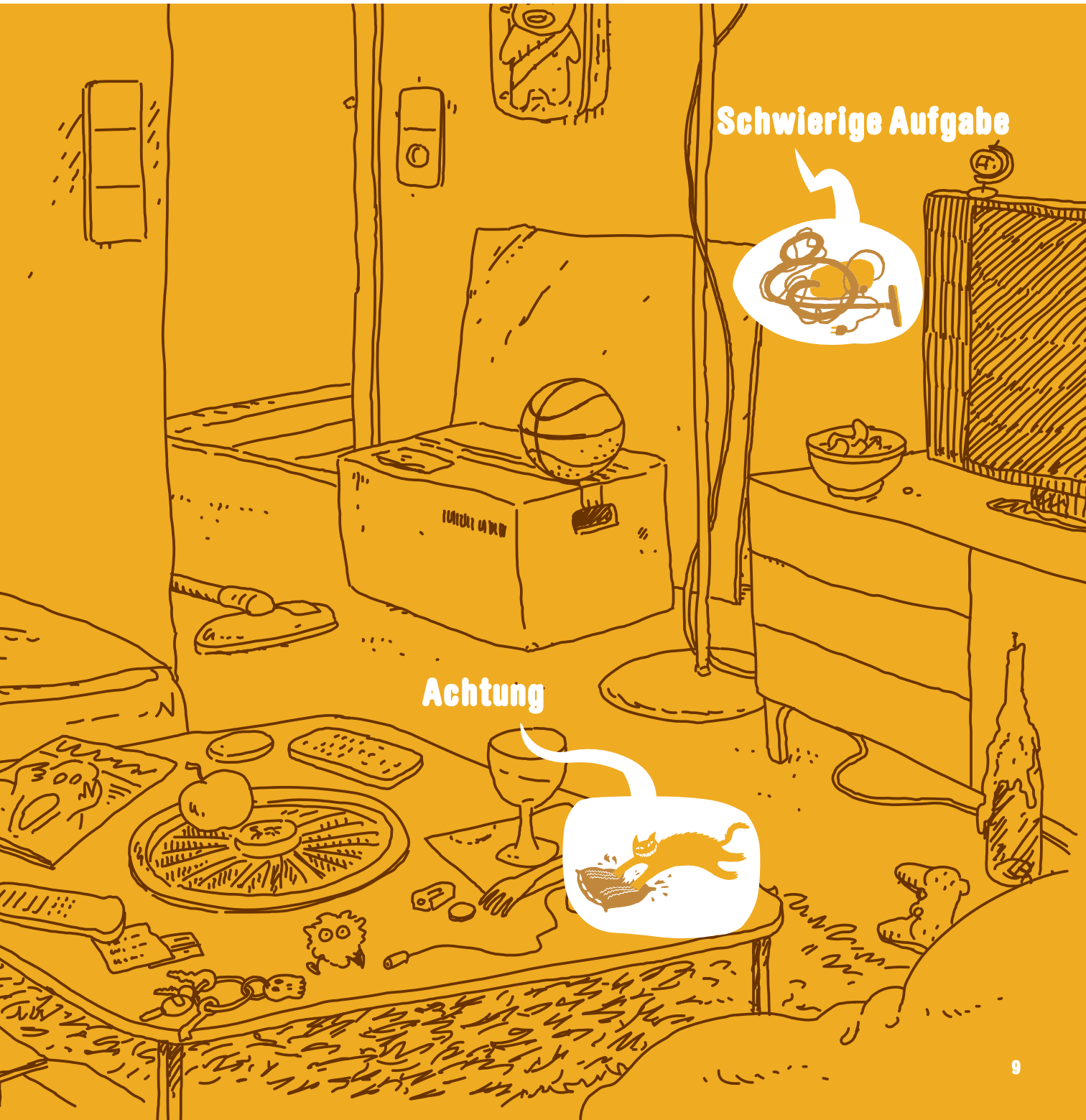
Schrödingers Werkstatt



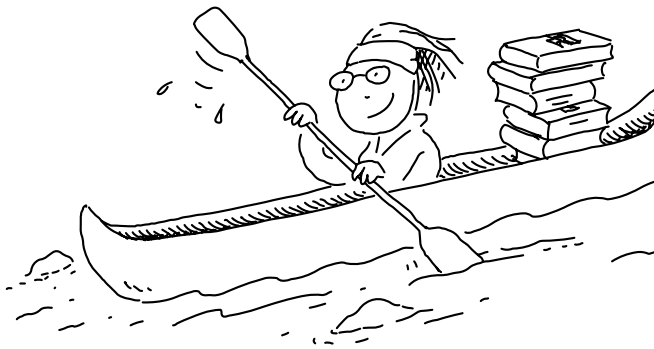
Unmengen von Code, der ergänzt, verbessert und repariert werden will

Schrödingers Wohnzimmer

Viel Kaffee, Übungen und die verdienten Pausen



Schrödinger programmiert C#

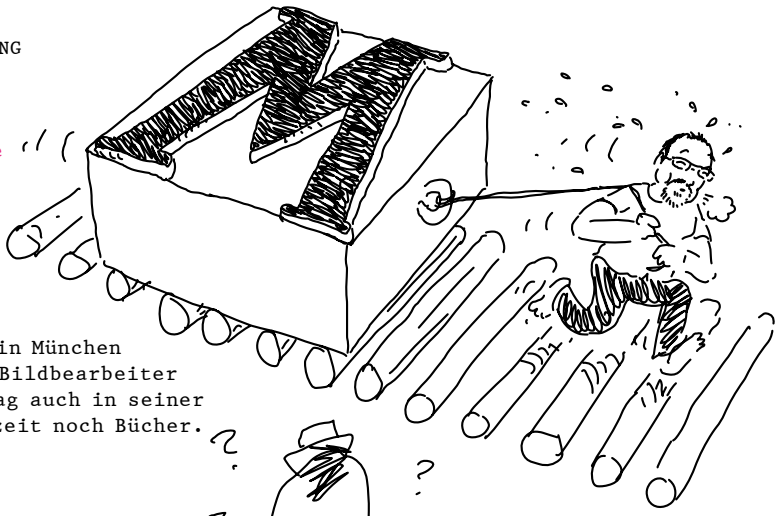


Als Kanutin umschifft Almut Stromschnellen mit notfalls nur einem Paddel. Eine Fähigkeit, die sie zur Fachbuchlektorin prädestiniert.

Almut Poll, LEKTORAT

Janina »Sherlock« Brönner. Bei der Buchherstellung sind detektivische Kombinationsgabe und Finesse gefragt. Die Kollegen haben sich allerdings das Pfeiferauchen verboten.

Janina Brönner, HERSTELLUNG



Okay, die Baumstämme hatten wir schon hingelegt, bevor Markus den Satz des Buches übernahm. Und der Rest war, wie man sieht, ein Kinderspiel, gell?

Markus Miller lebt und arbeitet in München als selbstständiger Setzer, Bildbearbeiter und Reinzeichner und mag auch in seiner Freizeit noch Bücher.

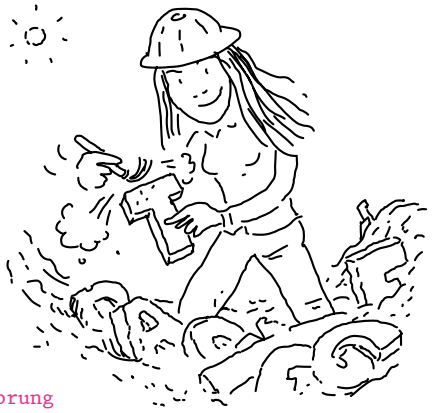
Schon zu Schulzeiten zeichnete Leo am liebsten die Bücher voll, von denen er am wenigsten kapierte.

Seit er weiß, dass man das auch gegen Bezahlung machen kann, kapiert er gar nichts mehr.

Leo Leowald lebt und arbeitet in Köln als freiberuflicher Illustrator. Er veröffentlicht unter anderem in *titanic*, *jungle world* und bei *reprodukt* und zeichnet seit 2004 den Webcomic *www.zwarwald.de*.

Andreas' zweite Leidenschaft neben der Buchgestaltung ist kochen. Wie auch immer: Hauptsache rare – VERY RARE!

Andreas Tetzlaff ist selbstständiger Buchgestalter in Köln. Er arbeitet normalerweise für Kunstbuchverlage – dass ausgerechnet ein IT-Fachbuch ihn vor künstlerische Herausforderungen stellt, hätte er sich vorher nicht träumen lassen …



Annette ist von Haus aus Archäologin,
da ist es nur ein kleiner Schritt bis zum Lektorat, und
der Vorteil ist: Bei Schrödinger und Co. findet sie immer was.

Annette Lennartz ist freiberufliche Lektorin in Bonn.
Für Schrödinger hat sie immer eine offene Tür. Privat schätzt
sie augenzwinkernde und gruselige Geschichten oder bastelt
an filigranen Schiffsmodellen.

KORRIGIERT VON: *Annette Lennartz*

Für Nicole, die »Hobby« zu »Hobbit« korrigiert
und deren Nachname alle Buchstaben des Wortes
»Nerd« enthält, ist es nur noch ein Katzensprung
aus den wilden Wäldern Neuseelands an den
Schrödinger-Schreibtisch.

Nicole Enders ist Diplom-Wirtschaftsinformatikerin,
Beraterin und Software-Entwicklerin. Sie erstellt
Kundenlösungen mit .NET, C# und SharePoint.

BEGUTACHTET VON: *Nicole Enders*, *Benjamin Kappel*

Uhrmacherei ist Präzisionshandwerk.
Wenn man sich wie Bernhard obendrein die Mühe
macht, eine Uhr zu bauen, die die Zeit in Worten
hinbuchstabiert, bleibt als nächste Herausforderung
eigentlich nur noch, ein Schrödingerbuch zu schreiben.

Bernhard Wurm führt ein Software-Unternehmen
in Österreich. Bevor er Schrödinger traf,
hat er schon einige Auszubildende in die
Kunst der C#-Programmierung eingeführt.

GESCHRIEBEN VON: *Bernhard Wurm*

Einbandgestaltung:
Andreas Tetzlaff und Leo Leowald
Initiales Design: Andreas Tetzlaff
Ein besonderer Dank auch an die Fachlektoren
Alexander, Florian, Günther, Manfred und Patrick.

FÜR DIE, DIE ES GENAU WISSEN WOLLEN

Dieses Buch wurde gesetzt aus unzähligen Schriften (u.a. aus der WIMBY von Evert Ypma: Danke, Evert!), Tonnen an Illustrationen und anderen komischen Zeichen, die alle Beteiligten in den Wahnsinn trieben.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-8362-6970-4

3., aktualisierte Auflage 2019

INHALTSVERZEICHNIS

Kapitel 1: Ein guter Start ist der halbe Sieg

Compiler und Entwicklungsumgebungen

Kapitel 2: Ein netter Typ

Datentypen und deren Behandlung

Kapitel 3: Alles unter Kontrolle

Bedingungen, Schleifen und Arrays

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

Strings, Characters und Nullable Types

Kapitel 5: Eine endliche Geschichte

Enumerationen

Kapitel 6: Teile und herrsche

Methoden

Kapitel 7: Klassengesellschaft

Objekte, Eigenschaften und Sichtbarkeiten

Kapitel 8: Es wird Zeit für Übersicht!

Namespaces

Kapitel 9: Erben ohne Sterben

Objektorientierte Programmierung

Kapitel 10: Abstrakte Kunst

Abstrakte Klassen und Interfaces

Kapitel 11: Airbags können Leben retten

Exceptionhandling

Kapitel 12: Ein ordentliches Ablagesystem muss her

Collections und Laufzeitkomplexität

Kapitel 13: Allgemein konkrete Implementierungen

Generizität

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

LINQ

Kapitel 15: Blumen für die Dame

Delegaten und Ereignisse

Kapitel 16: Der Standard ist nicht genug

Extension-Methoden und Lambda-Expressions

Seite 395

Kapitel 17: Die Magie der Attribute

Arbeiten mit Attributen

Seite 419

Kapitel 18: Ich muss mal raus

Dateizugriff und Streams

Seite 441

Kapitel 19: Sag doch einfach, wenn du fertig bist

Asynchrone und parallele Programmierung

Kapitel 20: Nimm doch, was andere schon gemacht haben

Die Paketverwaltung NuGet

Kapitel 21: Die schönen Seiten des Lebens

Einführungin XAML

Kapitel 22: Models sind doch schön anzusehen

Das Model-View-ViewModel-Entwurfsmuster

Kapitel 23: Stereotyp Schönheit

Universal Windows Platform (UWP) Apps

Kapitel 24: Toast-Notifications

Der Einsatz von Toast-Notifications

Kapitel 25: Live is Live

Die Verwendung von Live-Kacheln

Kapitel 26: Ich will alles rausholen

Datenzugriff über die Windows API

Seite 707

Kapitel 27: Funktioniert das wirklich?

Unit-Testing

Seite 725

Für meinen Sohn

Raphael

—EINS—

Compiler und Entwicklungs-umgebungen

Ein guter Start ist der halbe Sieg

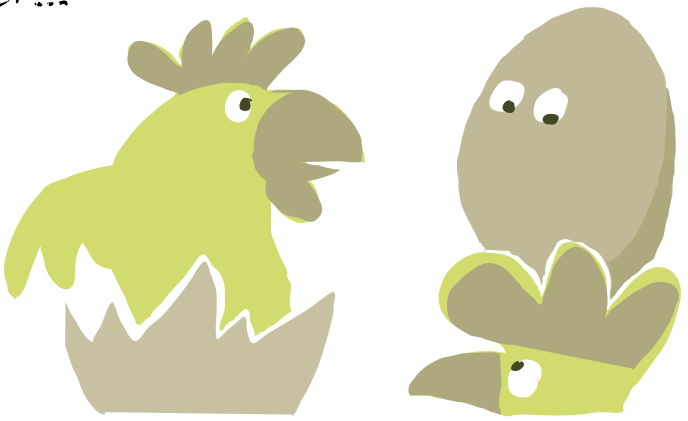
Schrödinger steht am Anfang seines neuen Jobs als C#-Entwickler. Sein Problem: Er kann noch gar kein C#. Sein erster Schritt zur Lösung: Er hat sich Hilfe geholt. Und er hat richtig Lust auf die Sache. Beste Voraussetzungen also. Jetzt ist der zweite Schritt an der Reihe: Installieren! Aber was?

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?

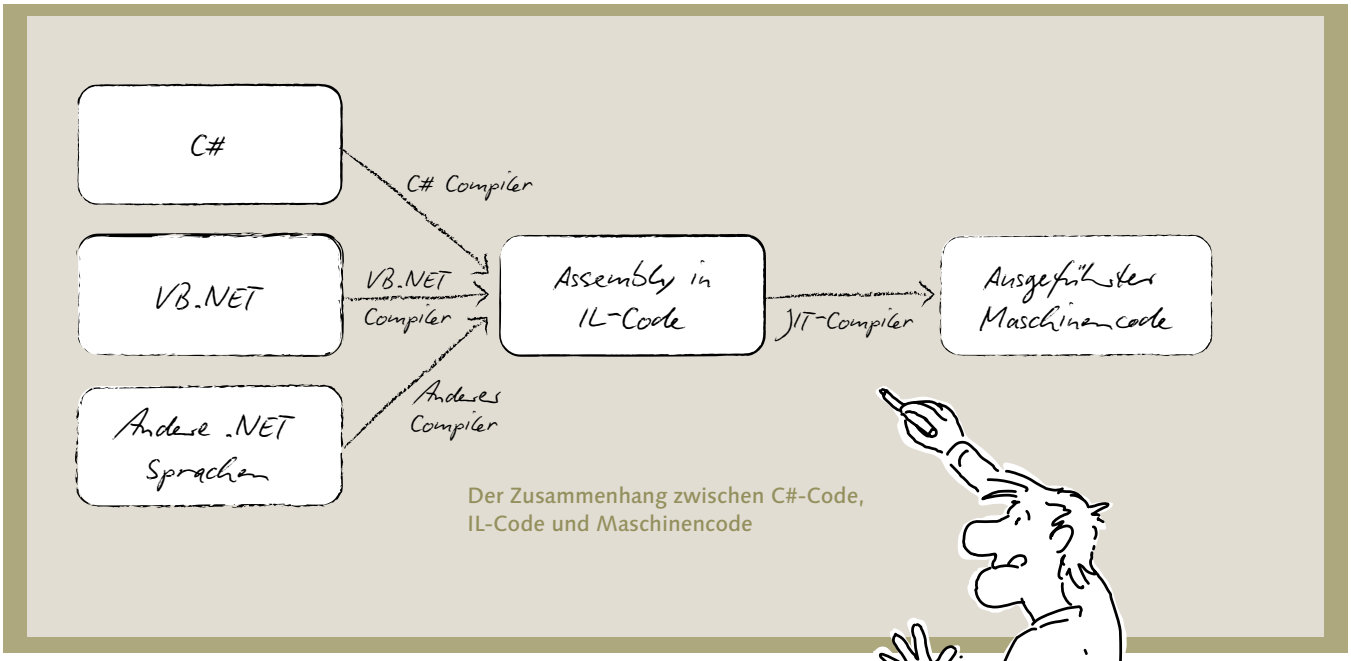
Nachdem du also mithilfe des C#-Compilers dein Programm kompiliert hast, erhältst du eine Datei, die IL-Code beinhaltet. Diese Datei nennt man **Assembly**. Ein Assembly kann, muss aber keine ausführbare EXE-Datei sein. Es kann auch eine Bibliothek – also eine DLL-Datei – sein, die verschiedene Funktionen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

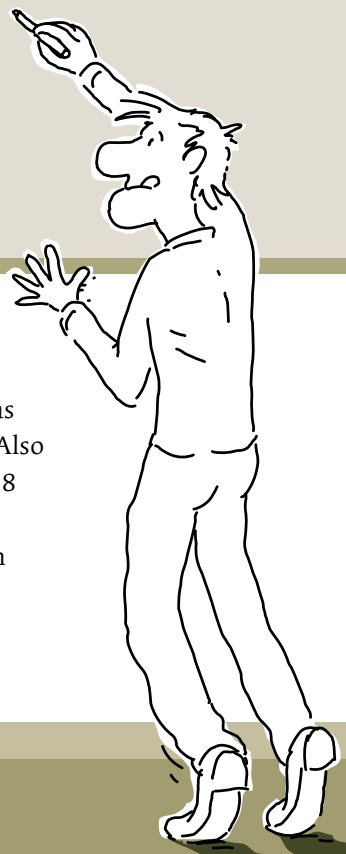
Du kannst dir das so vorstellen, dass der erste Befehl in dieser EXE-Datei ein Verweis auf die Common Language Runtime (kurz CLR) ist. Diese startet den JIT-Compiler, der den Code direkt, noch bevor dieser ausgeführt wird, in richtigen Maschinencode übersetzt und ihn dann ausführt.

[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) laufen kann, da er **plattformunabhängig** ist und **erst bei der Ausführung in den Maschinencode übersetzt** wird.

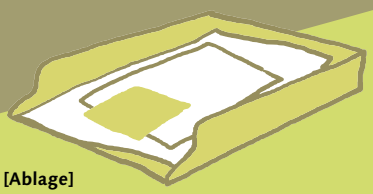


Der Zusammenhang zwischen C#-Code, IL-Code und Maschinencode

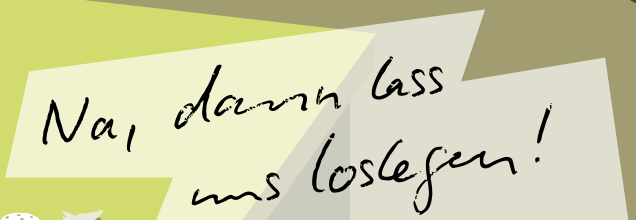


Du hast den Compiler übrigens schon auf deinem Windows-Rechner installiert. Dieser kommt nämlich automatisch mit dem .NET Framework, und das jeweils aktuelle Framework ist bei der Windows-Installation bereits dabei. Also zum Beispiel enthält Windows 7 das .NET Framework Version 4, Windows 8 das .NET Framework 4.5, Windows 8.1 die Version 4.5.1 und Windows 10 ohne Updates das .NET Framework 4.6, usw. 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
  }
}
```

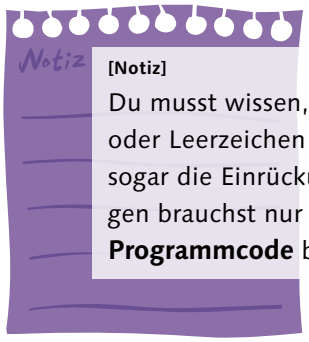
*1 Alle deine Programme starten mit der **Main**-Funktion. Alle Befehle innerhalb der beiden geschwungenen Klammern in dieser Funktion werden Zeile für Zeile ausgeführt.

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

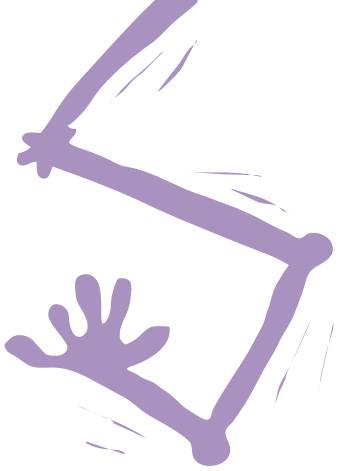
*3 … der Namespace **System** benötigt. **Console** ist ein fertiger Baustein, der innerhalb von **System** definiert ist. Daher musst du **using System** angeben, um darauf zugreifen zu können.

*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 Anwendungen für den Windows Store – sogenannte Windows Universal Apps, mit ganz offiziellem Namen **Universal Windows Platform Apps** oder UWP-Apps – entwickeln kannst. Und dazu benötigst du Visual Studio.

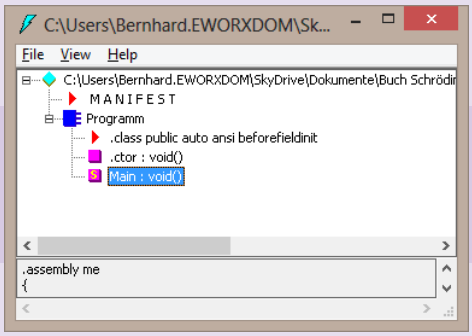
[Notiz]
Mono steckt übrigens auch in der Spieleentwicklungsplattform Unity, wo C# für die Skriptprogrammierung verwendet werden kann.

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 dieses 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

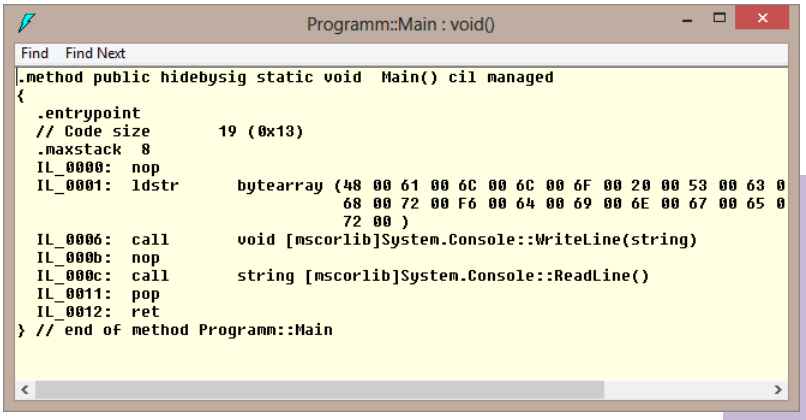
ildasm ist hier zu finden: **`C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6 Tools`**

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.

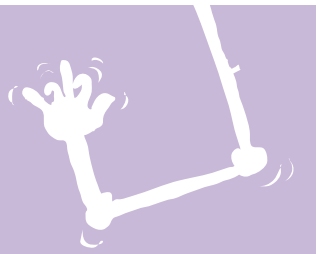
[Notiz]

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 komplizierter aus – vor allem die komischen Zahlen. Aber gut, dass ich das mal gesehen habe.

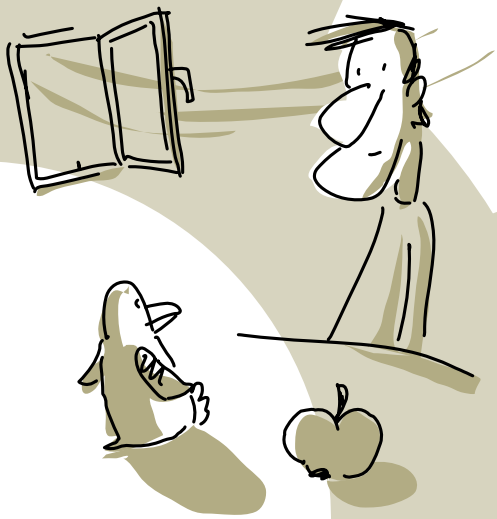
Du brauchst eine IDE!

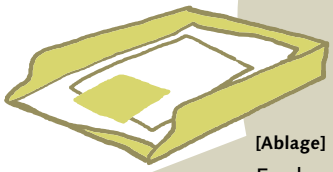
Obligatorisch ist zwar nur der Compiler, aber in einem x-beliebigen Texteditor macht programmieren keinen Spaß. Auch das Kompilieren kann komfortabler sein als auf der Kommandozeile. Die Entwicklungsumgebung – auch gerne IDE genannt – unterstützt dich bei deinen Projekten.

[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. Eine ebenfalls gute Alternative für die .NET-Entwicklung unter Windows, Mac und Linux ist **Visual Studio Code** von Microsoft. Um Anwendungen für den Windows 10-Store entwickeln zu können, benötigst du selbst Windows 10, 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]

Egal, welche Entwicklungsumgebung du verwendest, C# ist C# und bleibt C# – die Syntax ist also überall ident. Die IDEs unterscheiden sich im Allgemeinen bei Komfortfunktionen für den Entwickler.

Visual Studio Community Edition

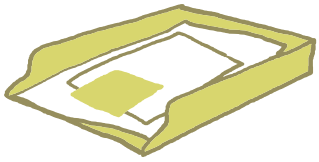
Visual Studio von Microsoft gibt es in verschiedenen Versionen. Die großen, kostenpflichtigen Enterprise-Versionen integrieren sämtliche Funktionalitäten, die das Entwicklerherz begehrt und auch einige, die du nie im Leben brauchen wirst. Es steht aber auch eine sehr gute kostenlose Versionen zur Verfügung: die **Visual Studio Community Edition**. Auch sie bietet alles was du brauchst, um **Webanwendungen, Desktopanwendungen oder Universal Windows Platform Apps** zu erstellen. Sogar wenn du für Android, Windows Phone oder für den Apfel entwickeln möchtest, ist dies mit der Community Edition möglich. Dies ist in Wahrheit eine kostenlose Professional Edition, die für **kleine Teams, Open Source Projekte und die Wissenschaft** gedacht ist.

Noch bin ich eine Ein-Mann-Armee, das ist klein genug.

[Hintergrundinfo]

Der lange Weg der Namensfindung zu **Universal Windows Platform Apps** startete mit dem Begriff Metro Apps, der nach einem Rechtsstreit fix zu **Windows Store Apps** wurde. Mit Windows 8.1 kamen die sogenannten **Universal Apps**, die eine teilweise gemeinsame Codebasis für Windows Store und Windows Phone erlaubten. **Windows 10** und die Windows Universal Platform perfektionieren dieses Ziel: Es läuft die absolut gleiche App auf dem Raspberry Pi wie auch auf einem Tablet oder Desktop – und heißt zu recht **Universal** Windows Platform App.

Das jüngste Visual Studio ist **Visual Studio Code**, welches aktuell für node.js und ASP.NET 5 Webanwendungen gedacht ist. Für uns ist es **nicht relevant**.



[Ablage]

Die Entwicklungsumgebungen kannst du unter **www.visualstudio.com** herunterladen.

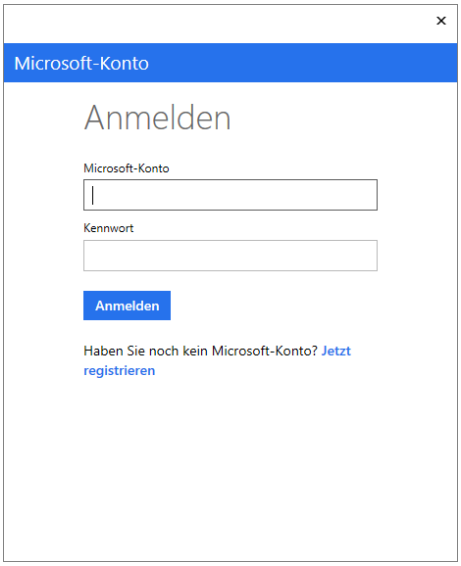
Installiere dir bitte die **Visual Studio Community Edition**, und dann geht es los.

Der Spaß geht los!

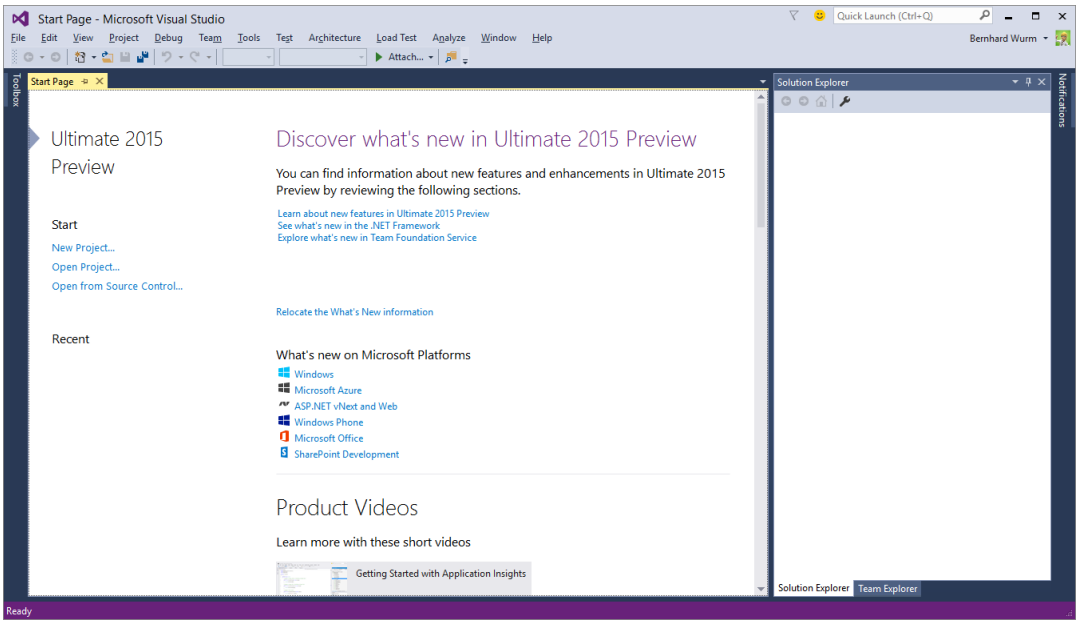
Die Software ist wie gesagt kostenlos. Du musst dich jedoch mit deinem Microsoft-Account an Visual Studio anmelden, damit es länger als 30 Tage funktioniert. Das kannst du gefahrlos machen. Einfach anmelden bzw. kostenlos registrieren, falls du noch keinen Account hast, und der Spaß kann losgehen.

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

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

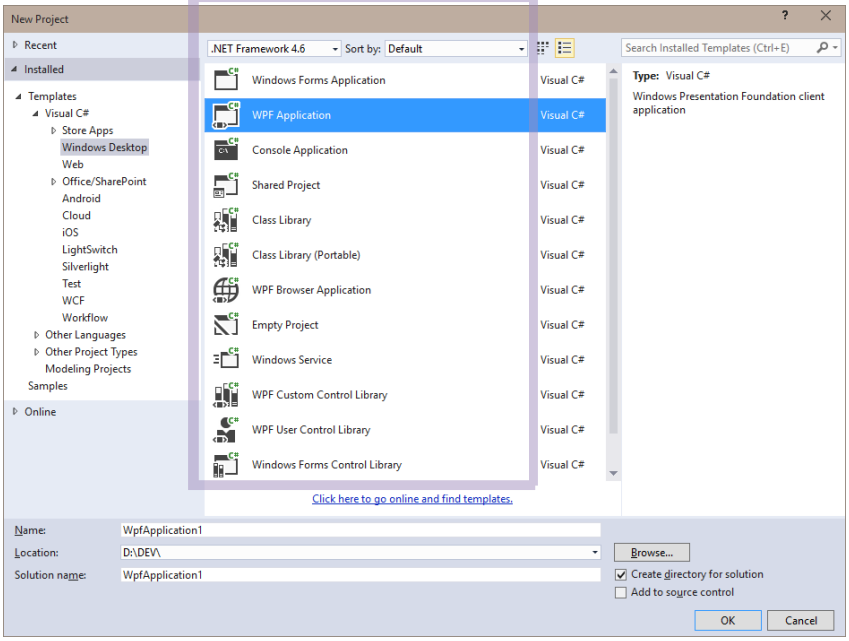


Anmelden mit dem Microsoft-Account



Visual Studio – Startbildschirm

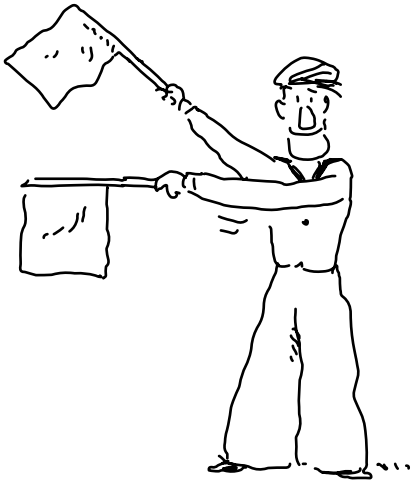
Durch einen Klick auf **`Neues Projekt`** erhältst du eine Auswahl mit den möglichen Projekttypen für diese Visual-Studio-Version. Diese Version beinhaltet alles, was du für klassische Desktopanwendungen benötigst, also ältere Windows-Forms-Anwendungen, modernere WPF-Anwendungen, Konsolenprogramme und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



Auswahl der Projekttypen von Visual Studio

.NET Core vs .NET Framework

Bevor die große Verwirrung beginnt, will ich dich gleich etwas aufklären. Du kannst oftmals zwischen zwei verschiedenen Projekten auswählen (je nach installierten Features von Visual Studio), zum Beispiel für Konsolenanwendungen. Eine basiert auf **.NET Core** und eine auf dem **.NET Framework**.



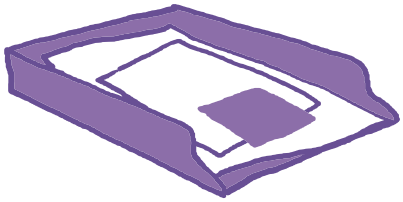
Ja, und welches ist besser?

.NET Framework ist sozusagen die alte Technologie. Nach über 15 Jahren Weiterentwicklung des .NET Frameworks haben sie nochmal neu angefangen.

[Notiz]

In 15 Jahren Softwareentwicklung tut sich einiges. .NET Core ist wieder auf dem neuesten Stand der Technik!

Das Ergebnis ist ein kleineres, schnelleres Open-Source-Framework, das auch auf Linux und Mac läuft. Obwohl das .NET Framework noch weiter gewartet wird und auch die eine oder andere Neuerung erhalten wird, geht die Zukunft der Entwicklung ganz klar in Richtung .NET Core. Microsoft hat angekündigt, künftig die Neuerungen primär für .NET Core zu entwickeln.



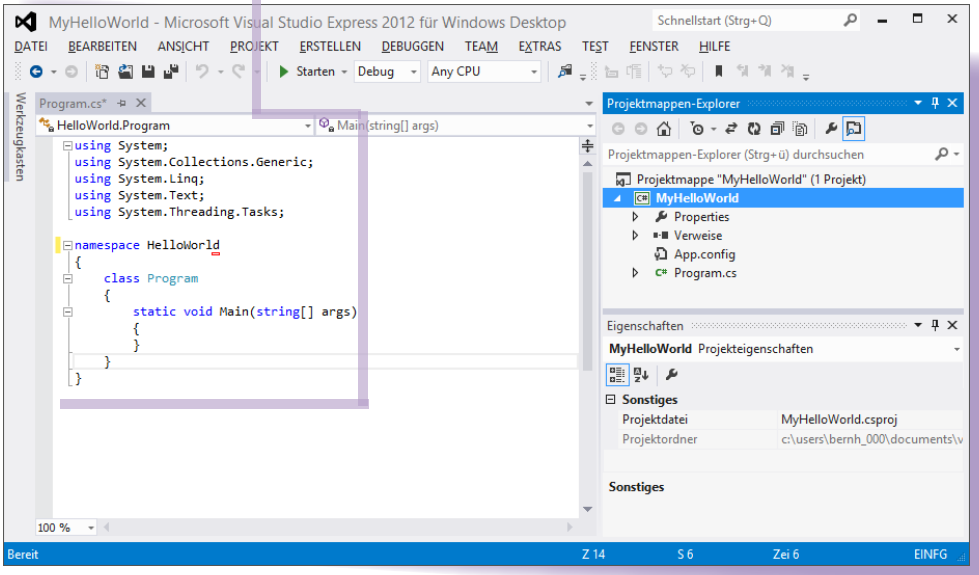
[Ablage]

Meine Empfehlung lautet: Wenn du ein neues Projekt beginnst und die Wahl zwischen dem alten .NET Framework und dem neuen .NET Core hast, verwende **.NET Core**! Unsere Konsolenanwendungen werden ebenfalls alle .NET Core nutzen.

Dein erstes Projekt

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.

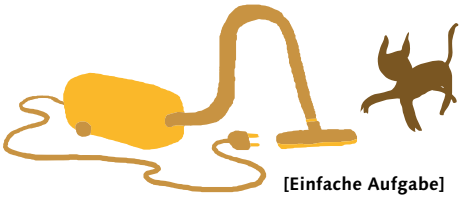
Nativ?

[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 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.

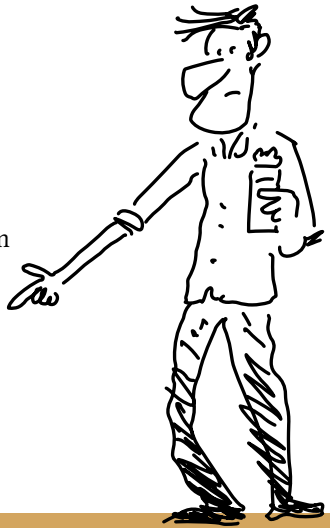
Lösung:

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.

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



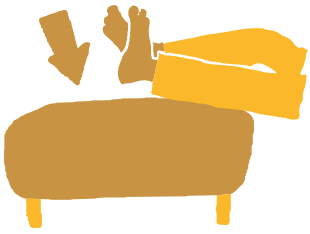
Das Fenster schließt sich immer gleich, nachdem das Programm beendet wurde. Du hast zwei Möglichkeiten: Entweder lässt du mit `Console.ReadKey()` den Computer am Ende noch ein Zeichen einlesen, dadurch wartet das Programm bis zur nächsten Benutzereingabe und schließt somit erst anschließend, oder du 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:

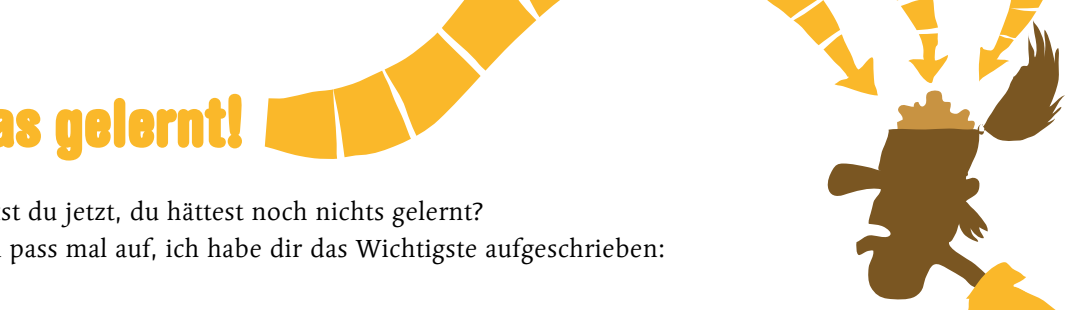
```
Console.WriteLine("Vorname eingeben:");
var vorname = Console.ReadLine();
Console.WriteLine("Nachname eingeben:");
var nachname = Console.ReadLine();
Console.WriteLine(vorname + " " + nachname);
```



[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.

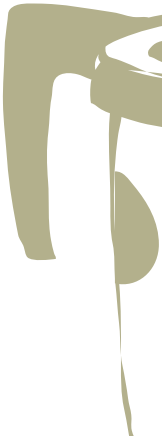
Dieses Glas für diesen Wein

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!

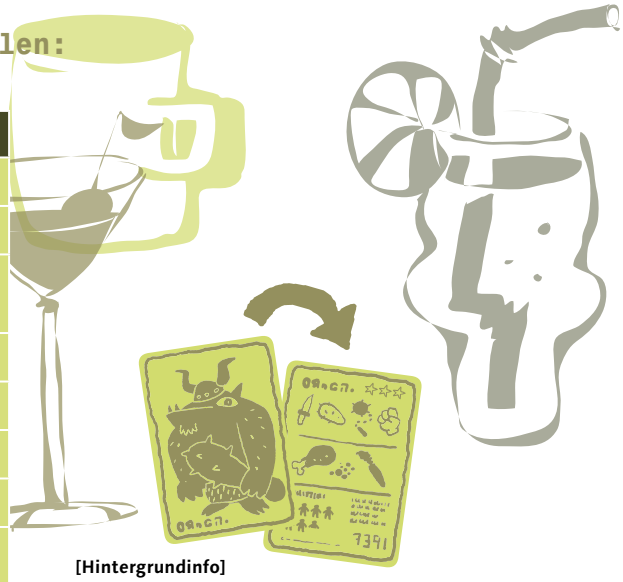


Der **`long`**-Datentyp ist bestimmt für Bill Gates entwickelt worden, damit der seinen Kontostand abspeichern kann.

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.

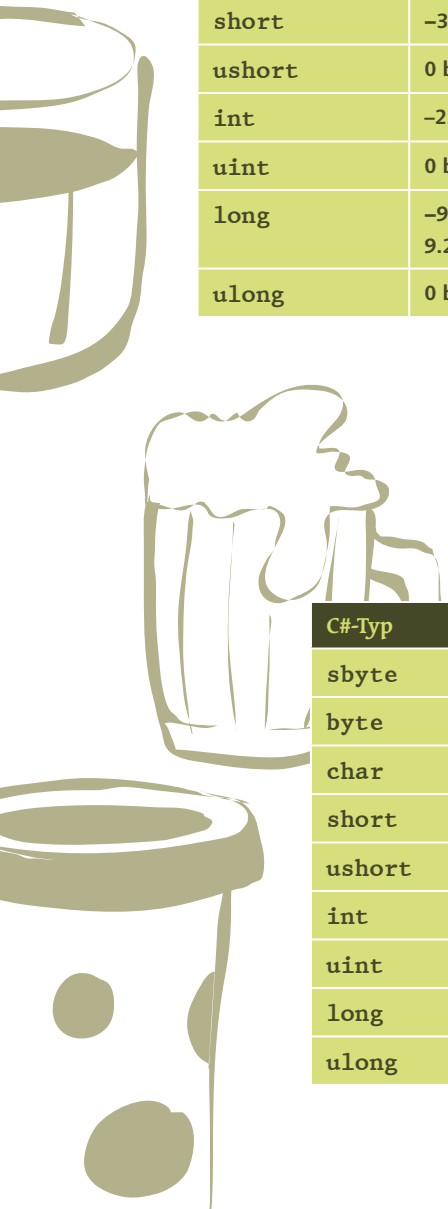
Die verschiedenen Glasgrößen für ganze Zahlen:

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



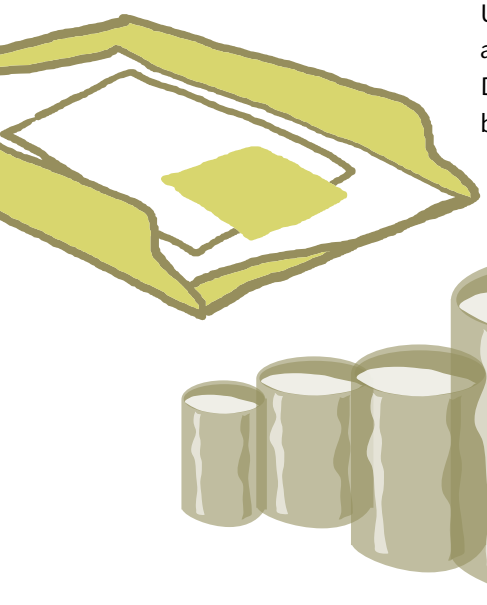
[Hintergrundinfo]

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

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 Nachricht ist angekommen.

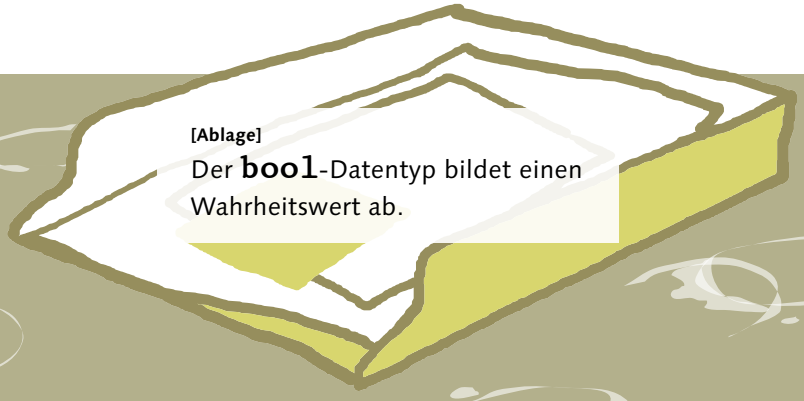
Große Datentypen sind wie große Gläser: Es passt viel hinein, aber sie brauchen auch viel Platz zum Lagern, und zwar unabhängig davon, wie viel im Glas drin ist.

[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 vergißt die Namen so, dass die so lustig aussehen wie Kamelhöcker...

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 = fussLaenge + 1.5;
double schuhgroesse = mitAbstand * 1.5;
```

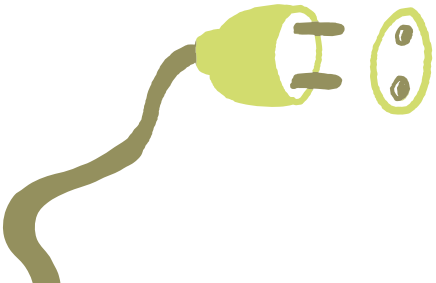
Wie du siehst, kannst du die Variablen einfach wie Zahlen verwenden. Und die klassischen Rechenoperationen sind sehr naheliegend. Erst wird der Teil rechts vom Gleichheitssymbol ausgerechnet, und dann wird das Ergebnis der Variablen auf der linken Seite zugewiesen.

Ich kann also auch ganze Zahlen Gleitkomma-Variablen zuordnen?

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

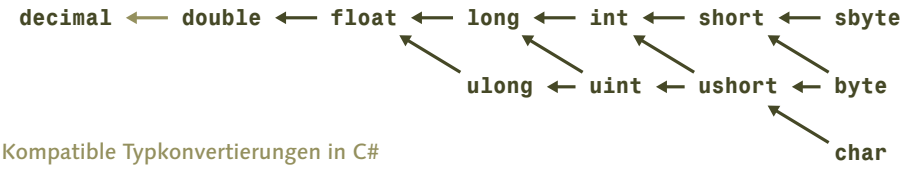
[Achtung]
Die automatische Konvertierung funktioniert immer nur von kleineren Datentypen nach größeren oder allgemeineren Datentypen.

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);
```

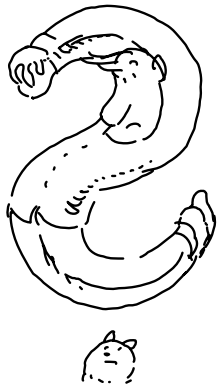
Und was passiert mit den Nachkommastellen?

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 255?



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

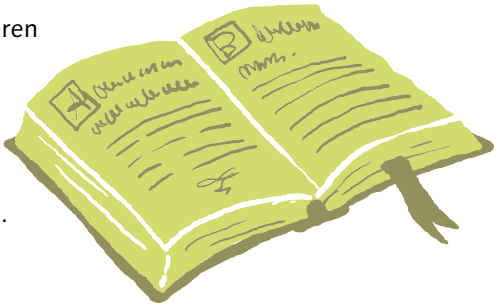
```
int i = 0;
i++;
```

[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;
```

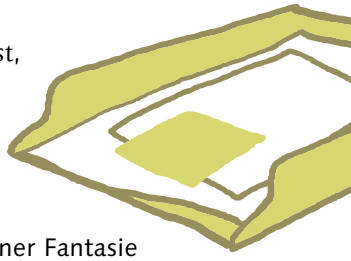
Welchen Wert hat `rest`?

Eins! Denkst du, ich kann gar nicht rechnen? 3 mal 3 ist 9, 1 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:

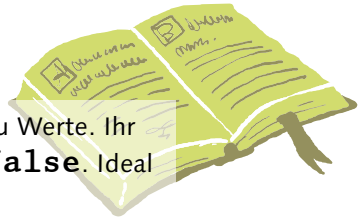
```
(i % 2 == 0)
```

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-Ratgeber?



[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)

Alles nur Klone!

Immer, wenn du einer Variablen den Wert einer anderen zuweist, wird der Wert (bei den Datentypen, die du bis jetzt kennengelernt hast) **kopiert**. (Zumindest bei den Datentypen, die wir bis jetzt besprochen haben, ist das so.) Dann kannst du die beiden Variablen unabhängig voneinander verändern.

```
int variable1 = 3;
int variable2 = variable1;
variable1++;    // Variable2 bleibt unverändert beim Wert 3.
```

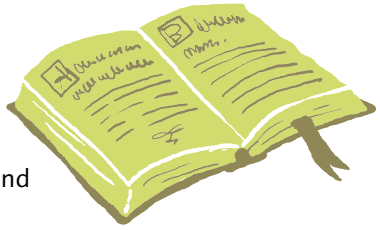
[Funktioniert in]

Seit C# 7.2 gibt es die Möglichkeit, Variablen anzulegen, die nur auf eine andere Variable verweisen.

Die beiden Variablen teilen sich also den Wert?

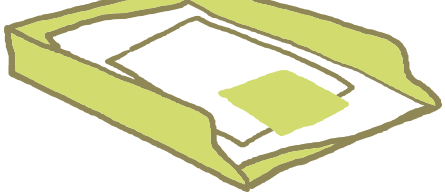
Genau. Egal, welche Variable du veränderst, beide haben immer den gleichen Wert.

So, als würden wir beide aus einer Tasse Tee trinken? Egal, wer trinkt oder nachfüllt, wir haben nur eine gemeinsame Tasse.



[Begriffsdefinition]

Verweis heißt auf Englisch **Reference** und wird in C# immer mit `ref` abgekürzt.



[Ablage]
Du kannst dir Verweisvariablen wie Verknüpfungen bei deinem Computer vorstellen. Das sind auch nur Verweise auf die eigentliche Datei.

```
int gemeinsameVariable = 3;
ref int verweisAufVariable = ref gemeinsameVariable;*1
```

*1 Um eine Verweisvariable (eine Variable, die sich den Wert mit einer anderen Variablen teilt) anzulegen, brauchst du bei der Definition und der Wertzuweisung das Schlüsselwort **ref**.

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 **||**.

[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.

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
```

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 deine 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.

Soll das heißen, mein Schnitzel hat kaum eine Chance, weil immer zuerst auf den Salat geprüft wird?! Och no.

Mithilfe von geteilten Variablenwerten hast du jedoch eine Chance, essen zu gehen.

```
ref var bleibeZuHause = ref essenZuHause;
bleibeZuHause = false;
```

[Notiz]
`bleibeZuHause` und **`essenZuHause`** teilen sich den gleichen Wert. Sobald du also **`bleibeZuHause`** auf **`false`** setzt, ist es egal, ob du die Zutaten für den Salat hast oder nicht. Auch der Wert von **`essenZuHause`** wird **`false`**.

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?

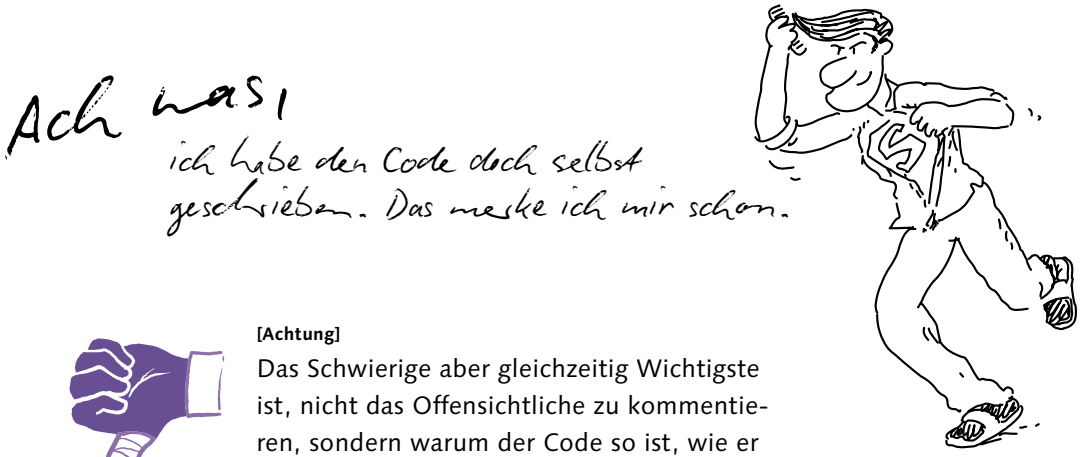
```
bool ergebnis = false || true;
ergebnis |= false;
ergebnis = !ergebnis;
ergebnis |= !true;
```

[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.

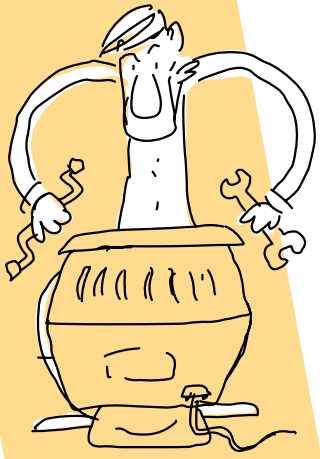
[Achtung]
Sobald der Datentyp festgelegt ist, kann dieser nicht geändert werden, auch mit **`var`** nicht.

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!

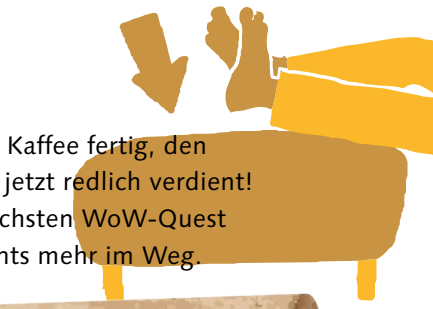
Aber welches Datentyp wird für **`wert5`** angenommen, wenn **`wert2`** ein **`int`** und **`wert3`** ein **`double`** ist?

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 Cliquen 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 auf 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.
- Werden Variablen mit ref definiert, so teilen diesesie sich den Wert mit anderen Variablen. Ohne **`ref`** wird bei Zuweisungen der Wert kopiert.
- 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.

—DREI—

Bedingungen, Schleifen und Arrays

Alles unter Kontrolle

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.

Bedingungen

Variablen ein- und auszulesen ist ja ganz nett, aber ich muss doch Unterscheidungen treffen, Vorgänge wiederholen usw.

Nur keine Hektik! Bald hast du auch das drauf! Als Erstes zeige ich dir Unterscheidungen. Bei diesen **`if`**-Anweisungen benötigst du einen **booleschen Ausdruck** – also etwas, das wahr oder falsch sein kann.

Eine **`bool`**-*Variable also.*

Die funktioniert natürlich. Aber es muss keine Variable sein, es muss nur etwas sein, das am Ende **`true`** oder **`false`** ergibt.

*1 Der Ausdruck in den runden Klammern wird ausgewertet, und je nach Ergebnis wird der **`if`**- oder der **`else`**-Zweig ausgeführt.

```
bool freundinDa = true;
if (freundinDa == true) { *1
    // Fernsehabend mit Schnulzenfilm *2
}
else {
    // WoW zocken mit Pizza und Chips *3
}
// Am Ende auf jeden Fall schlafen gehen *4
```

*2 Der Bereich innerhalb der geschwungenen Klammern wird nur ausgeführt, wenn die Bedingung erfüllt ist.

*3 Der Bereich innerhalb der geschwungenen Klammern nach dem **`else`**-Zweig wird nur dann ausgeführt, wenn die Bedingung, die beim **`if`** angegeben wurde, nicht erfüllt ist.

*4 Am Ende geht's nach den Klammern wieder weiter.

Ich kann aber prinzipiell schon frei entscheiden, was ich tun möchte! Na ja, es stimmt schon, dass ich kaum einen WoW-Abend habe, wenn meine Freundin da ist, wenn ich's mir recht überlege…

Das hier

```
bool nochmal = true;
if (nochmal == true) { … }
```

ist das Gleiche wie

```
if (nochmal*1) { … }
```

*1 Gibst du direkt eine **bool**-Variable an, so wird diese auf den Wert **true** geprüft.

und natürlich das Gleiche wie

```
if (nochmal != false) { … }
```

Willst du das Gegenteil einer **bool**-Variablen, also den **false**-Wert, abfragen, funktioniert das natürlich **so:**

```
if (nochmals == false) { … }
```

Oder auch so:

```
if (nochmals != true) { … }
```

Oder in Kurzform auch so:

```
if (!nochmals*1) { … }
```

*1 Durch den **!**-Operator wird der Wert verneint. Das **!** ist der Nicht-Operator, auf Fachchinesisch Negationsoperator.

[Hintergrundinfo]

Operatoren werden auf sogenannte **Operanden** angewendet. Es gibt unäre Operatoren wie das **!**, das einen **bool**-Wert invertiert und eben einen Operanden benötigt (das bedeutet »unär«). Die meisten Operatoren, wie die Vergleichsoperatoren, sind **binäre Operatoren** und benötigen **zwei Operanden**. Es gibt auch – genau einen – ternären Operator, nämlich die Kurzform von wenn-dann-sonst: **? :**

Also wenn **gutesWetter** falsch ist, ist **!gutesWetter** wahr.
Nachvollziehbar.

Um Tipparbeit zu sparen, kannst du die geschwungenen Klammern sowohl im **`if`**- als auch im **`else`**-Bereich weglassen, wenn du lediglich eine Zeile im Zweig hast.

[Zettel]
Die Einrückung innerhalb des **`if`**- und **`else`**-Bereichs ist nur für dich wichtig. Dem Compiler ist die Einrückung egal. Dieser orientiert sich nur an den geschwungenen Klammern.

[Achtung]
Verlier jedoch nicht die Übersicht. Unerfahrene Programmierer übersehen leicht die Zusammenhänge, wenn Sie die geschwungenen Klammern weglassen.
Es kann sogar sein, dass die **Programmierrichtlinien von beispielsweise Blizzard** das verbieten, um potenzielle Fehler bereits im Vorfeld zu vermeiden.

Ich empfehle dir ganz klar, zu Beginn deiner Programmiererlaufbahn immer die geschwungenen Klammern zu verwenden!

Kennst du noch die Kurzschlussauswertung? Die wird auch hier verwendet. Um beim Beispiel vom letzten Kapitel zu bleiben: Sobald die Salatzutaten vorhanden sind, wird nicht mehr geprüft, ob du Schnitzel mit Pommes bekommst. Dieser Ausdruck ergibt dann sofort **`true`**:

```
salatZutaten || (schnitzel && pommes);
```

Sobald also das Ergebnis bekannt ist, wird der Rest nicht mehr ausgewertet, sondern gleich der **`if`**- oder der **`else`**-Zweig aufgerufen. Ach ja, natürlich kannst du den **`else`**-Zweig auch weglassen.

```
bool hatHundHunger = false;
bool hatKatzeHunger = true;
if (hatHundHunger || hatKatzeHunger) {
    // Fütterungszeit
}
```

In der Kürze liegt die Würze

Kurz

Wie schon kurz erwähnt, kannst du die geschwungenen Klammern sowohl im `if`- als auch im `else`-Zweig **weglassen**, wenn dieser nur aus einer Zeile besteht.

Noch kürzer

Noch kürzer geht es, wenn du einer Variable »je nachdem« den einen oder den anderen Wert zuweisen willst.

Schrödinger, was schaust du dir heute Abend im Fernsehen an?

Kurz

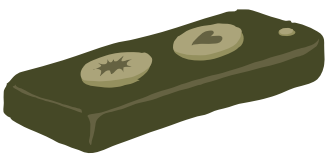
Einen guten Actionfilm natürlich – außer, meine Freundin ist da, dann sehen wir uns wohl einen Liebesfilm an.

```
int actionFilm = 1;
int liebesFilm = 2;
int tvKanal;
if (freundinDa)
    tvKanal = liebesFilm;
else
    tvKanal = actionFilm;
```

Pass auf, eine Sache habe ich noch für dich: Du verwendest die `bool`-Abfrage als Bedingung. Danach kommt ein Fragezeichen. Der Wert der Variablen nach dem Fragezeichen wird verwendet, wenn die `bool`-Abfrage `true` ergibt. Dann folgt ein Doppelpunkt, und ansonsten wird der Wert nach dem Doppelpunkt verwendet. Das Ding mit dem `?` `:` heißt **bedingter Operator.**

Noch kürzer

```
int actionFilm = 1;
int liebesFilm = 2;
int tvKanal = freundinDa ? liebesFilm : actionFilm;
```



Das gefällt mir. Die Filmwahl wird dadurch jedoch nicht besser.

Durch Variationen bleibt es interessant

Ich habe dir einiges an Operatoren gezeigt. Du kennst Vergleiche (**<**, **>**, **>=**, **<=**, **!=**), logische Verknüpfungen (**&&**, **||**), Verneinungen (**!**) und auch den bedingten ternären Operator (**? :**). Das üben wir ein bisschen:

```
bool einFilmFürDich = actionFilm || thriller || horrorFilm;
bool mädchenFilm = !einFilmFürDich;
```

Du kannst dir hier auch die Variable sparen, indem du den gesamten Ausdruck verneinst:

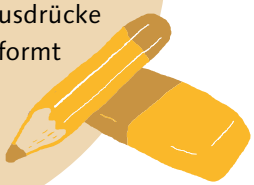
```
bool mädchenFilm = !(actionFilm || thriller || horror);
```

Oder es ist natürlich auch erlaubt, auf die Klammern zu verzichten:

```
bool mädchenFilm = !actionFilm && !thriller && !horror;
```

Und so wird aus einem Oder auch schon ein Und.
Aus und wird oder, aus ja wird nein.
Was für eine verkehrte Welt.

[Notiz]
Logische (= boolesche) Ausdrücke können ineinander umgeformt werden.



[Achtung]
Bei der Umwandlung von logischen Ausdrücken ändern sich häufig das Und und das Oder, auch die Negierung kommt häufig hinzu oder weg, vor allem beim Entfernen oder Hinzufügen von Klammern. Achte darauf, dass der Ausdruck wirklich das Gleiche bedeutet!

[Notieren/Üben]

Um sicherzustellen, dass zwei boolesche Ausdrücke das Gleiche ergeben, kannst du dir eine Wahrheitstabelle aufzeichnen. Jede boolesche Variable bekommt eine Spalte für die möglichen Wertkombinationen. In einer eigenen Spalte schreibst du das Ergebnis des Ausdrucks mit dieser Belegung.

Ein Beispiel:

```
Film = Fernbedienung && Programm
Film1 = Fernbedienung || Programm
```

Fernbedienung	Programm	Film	Film1
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Die beiden Ausdrücke sind nicht gleich, da die Ergebnis-Werte von **`Film`** und **`Film1`** nicht gleich sind.

[Einfache Aufgabe]

Schau Dir diese Ausdrücke genau an.
Welche bedeuten wirklich das Gleiche?

1. Gehen wir heute aus? Überprüfe, ob diese Ausdrücke das Gleiche bedeuten:

```
fernsehen && chips && !(kino && popcorn)
fernsehen || chips || !kino || !popcorn
fernsehen && chips && !(kino || !popcorn)
```

[Lösung]

Der erste und der dritte bedeutet das Gleiche. Der zweite ergibt sofort **`true`**, sobald nur ein einziges der Teile **`true`** ergibt. Also sobald Chips zu Hause sind, oder ein Fernseher vorhanden ist, ergibt dies bereits **`true`**.

2. Bleibt die Bahn noch in der Werkstatt?

```
bahnKaputt || (startsignalGesetzt && !bahnFrei)
(bahnKaputt || startsignalGesetzt) && (!bahnFrei || bahnKaputt)
```

[Lösung]

Das ist das Gleiche.

3. Lecker Essen!

```
((!suess && lecker) || mitKuemmel) && !gekocht
((!suess && lecker && !gekocht) || mitKuemmel && !gekocht)
!((!suess && lecker) || mitKuemmel) && !gekocht)
((suess || !lecker) && !mitKuemmel) || gekocht
```

[Achtung/Vorsicht]

Achtung: Das Und ist stärker als das Oder.

[Lösung]

Variante 1 und Variante 2 von Süß & Lecker sind gleich. Die letzten beiden Varianten ergeben genau das Gegenteil der ersten beiden.

Der Herr der Fernbedienung

EIN FILM, SIE ZU UNTERHALTEN, SIE VOR DEN FERNSEHER ZU TREIBEN, UM DORT EWIG ZU BLEIBEN.

Die Theorie ist nicht alles! Jetzt geht's an die Praxis, Schrödinger.

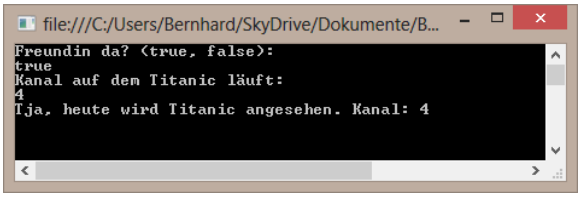
Das hört sich nach Arbeit an.

Ich möchte das Programm für die Abendgestaltung kurz festhalten.

```
using System;

namespace Codes {
  class Program {
    static void Main(string[] args) {
      Console.WriteLine("Freundin da? (true, false):");
      bool freundinDa = bool.Parse(Console.ReadLine());
      if (freundinDa == true) {
        Console.WriteLine("Kanal auf dem Titanic läuft:");
        int kanal = int.Parse(Console.ReadLine());
        Console.WriteLine("Tja, heute wird Titanic angesehen. Kanal: " + kanal);
      }
      else {
        Console.WriteLine("Horror (true) oder Action (false)?");
        bool horror = bool.Parse(Console.ReadLine());
        if (horror)
          Console.WriteLine("Gute Wahl heute wirds gruselig.");
        else
          Console.WriteLine("Ein guter Actionfilm ist immer toll.");
      }
      Console.ReadKey();
    }
  }
}
```

Jetzt bist du dran. Bring das mal ans Laufen.

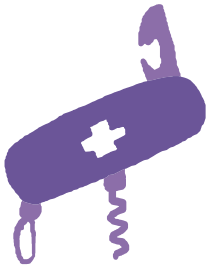


So sieht das Programm bei der Ausführung aus.
Und ganz bestimmt ist deine Freundin heute da.

Du siehst, dass du **`if`**-Anweisungen natürlich auch beliebig verschachteln kannst. Je mehr du diese ineinander verschachtelst, desto schwieriger wird es, in deinem Code den Überblick zu behalten.

Ja, das habe ich mir schon gedacht.

Und jetzt schreibst du selbst weiter:



[Einfache Aufgabe]
Schreibe den Code doch etwas um, so dass du gefragt wirst, ob du Horror sehen willst oder, falls nicht, Action oder, falls auch das nicht, einen Thriller.

Wird gemacht!

So geht's:

```
// … das ist nur der geänderte Codeteil
Console.WriteLine("Wie wäre es mit Horror?");
bool horror = bool.Parse(Console.ReadLine());
if (horror)
     Console.WriteLine("Gute Wahl heute wirds gruselig.");
else {
     Console.WriteLine("Doch ein Actionfilm?");
     bool action = bool.Parse(Console.ReadLine());
     if(action)
          Console.WriteLine("Ein guter Actionfilm ist immer toll.");
      else {
          Console.WriteLine("Thriller?");
          bool thriller = bool.Parse(Console.ReadLine());
          if(thriller)
               Console.WriteLine("Oh es wird spannend. Ein Thriller steht an.");
           else {
               Console.WriteLine("Dann gibt es heute kein Fernsehen.");
          }
     }
}
```

Siehst du, schon wird es etwas unübersichtlich.

Ist noch Bier da?

Wenn das mit einem anständigen Film schon nichts wird, ist hoffentlich noch etwas Bier im Kühlschrank. Also nach dem Motto: Guck mal, ob Bier im Kühlschrank ist. Wenn ja – ach lass mal. Und sonst geh welches einkaufen:

```
if (istBierImKühlschrank){
    // lass mal
}
else {
    GeheEinkaufen();
}
```

Gibt es auch eine Möglichkeit, dass ich den **if**-Zweig weglasse, weil ich nur den **else**-Zweig benötige, oder muss ich diesen dann leer lassen?

Nein, das gibt es nicht. Es ist aber auch nicht notwendig. Du musst in diesem Fall die Abfrage im **if** so umdrehen, dass du den **else**-Zweig nicht benötigst. Immerhin kannst du jeden Vergleich oder jede **bool**-Abfrage auf **true** oder auf **false** abfragen oder invertieren, dadurch hast du immer einen **if**-, aber möglicherweise keinen **else**-Zweig.

Es schaut echt komisch aus, wenn du ein **if** mit leerem Bereich baust –
Schrödinger, so etwas macht man beim Programmieren nicht.

[Erledigt!]
Leere **if**-Zweige sind kein guter Programmierstil. Jeder boolesche Ausdruck kann invertiert werden. Dadurch sind leere **if**-Zweige nicht erforderlich.

Umdrehen heißt in diesem Fall: »Guck mal, ob das Bier im Kühlschrank alle ist. Dann musst du nämlich welches einkaufen gehen.«

Oder im Code:

```
if (!istBierImKühlschrank)
    GeheEinkaufen();
```

Einer von vielen

Oftmals hast du bei Abfragen nicht nur einen einzelnen Wert, sondern viele **if-else-if-else-if-else**-Varianten, also viele Kombinationen, zu prüfen. Hierzu existiert ein Sprachkonstrukt mit der Bezeichnung **switch**.

Switch, das erinnert mich an Sandwich.
Oh Mann, mir knurrt vielleicht der Magen.

Geduld, Geduld. Bei der **switch**-Anweisung hast du verschiedene Zweige, die du betreten kannst, und einen Standardzweig am Ende:

*1 Beim **switch** wird direkt die Variable angegeben, die die verschiedenen Werte beinhalten kann.

*2 Ein **case**-Zweig beschreibt einen möglichen Wert. Dieser Wert wird im jeweiligen Zweig behandelt.

```
Console.WriteLine("Bitte Monat eingeben:");
int month = int.Parse(Console.ReadLine());
switch (month) { *1
    case 1: *2
        Console.WriteLine("Der erste Monat dieses Jahres.");
        break; *3
    case 2:
        Console.WriteLine("Valentinstag nicht vergessen!");
        break;
    case 3: *4
    case 4:
        Console.WriteLine("Es ist März oder April!");
        break;
    //....
    case 12:
        Console.WriteLine("Es ist Dezember - Weihnachten, yippie.");
        break;
    default: *5
        Console.WriteLine("Hey, so viele Monate hat das Jahr doch gar nicht.");
        break;
}
```

*3 Ein **case**-Zweig endet immer mit **break** – oder einem Fehler oder einem Return, aber im Allgemeinem mit einem **break**. Das ist guter Programmierstil.

*4 Ein Zweig kann durch die Angabe mehrerer **case**s auch mehrere Werte behandeln.

*5 Der **default**-Zweig wird aufgerufen, wenn keiner der anderen Zweige zutrifft.