mitp

Hans-Georg Schumann

Spiele programmieren mit Unity

FÜR KIDS

Inklusive DVD-ROM

Kleiner Crashkurs in C#

Hier bist du richtig, wenn du dich beim Programmieren in C# noch fühlst wie ein Nichtschwimmer im Wasser. Auf jeden Fall musst du dich auf einen zügigen Durchmarsch gefasst machen, wir springen direkt ins (kalte aber nicht zu tiefe) Wasser, um die wichtigsten C#-Grundlagen zu sammeln.

Sollte dir das alles zu schnell oder ungenügend erklärt sein, kann ich dir nur die Lektüre eines ausführlicheren Buches empfehlen – wie z. B. **Visual C# für Kids**.

In diesem Kapitel lernst du

- wie ein C#-Programm aufgebaut ist
- einige Datentypen kennen
- etwas über Kontrollstrukturen
- etwas über Klassen und Objekte
- etwas über Methoden und Parameter
- wie du eigene Klassen und Methoden vereinbarst

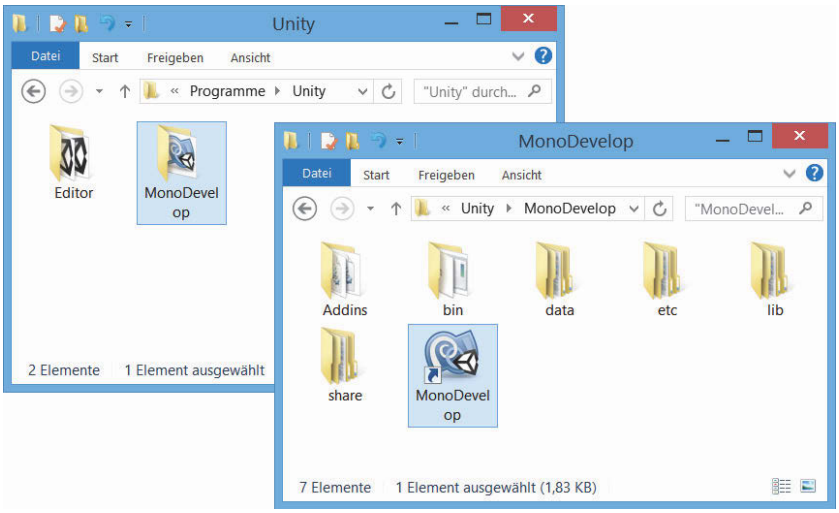
Start mit MonoDevelop

Schon in wenigen Minuten geht es los, nachdem dein C#-Entwicklungssystem gestartet ist. Das funktioniert wie bei fast jedem anderen Windows-Programm.

Weil Unity bereits eine Arbeitsumgebung für C# anbietet, benutzen wir hier auch dieses Werkzeug. Es heißt **MonoDevelop** und wird zusammen mit Unity installiert (siehe Anhang B im Buch).

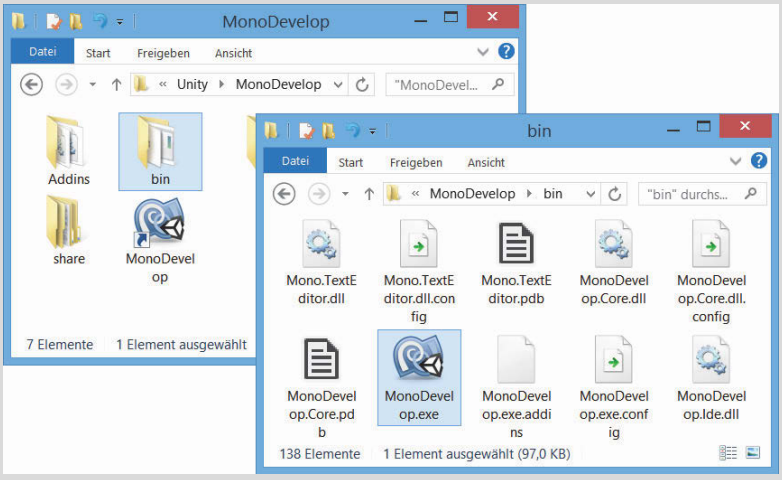
Du findest die Datei unter C:\PROGRAMME im UNITY-Ordner.

≫ Klicke dort auf das Symbol für den Unterordner MONODEVELOP.



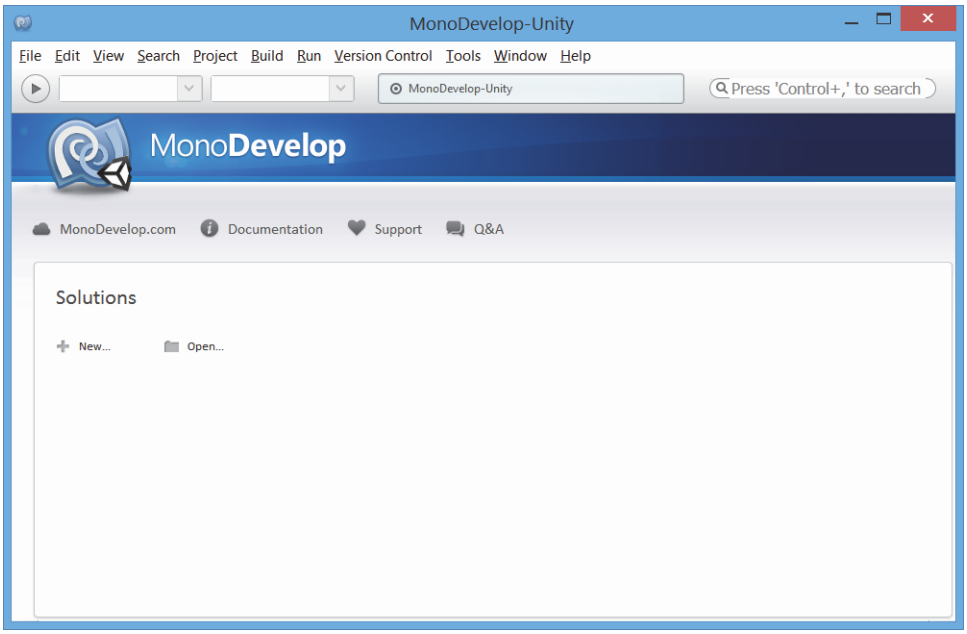
≫ Und dann auf das Verknüpfungssymbol für MONODEVELOP. Kopiere es auf den Desktop.

Falls du die Original-Datei MONODEVELOP.EXE suchst, die befindet sich im BIN-Ordner.

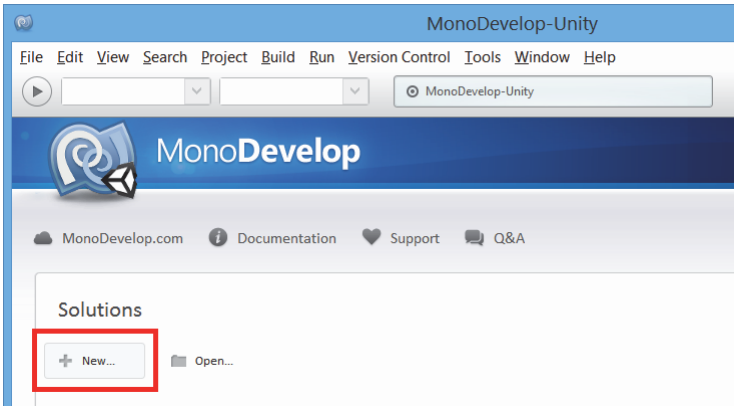


≫ Nun kannst du auf das Symbol auf dem Desktop klicken.

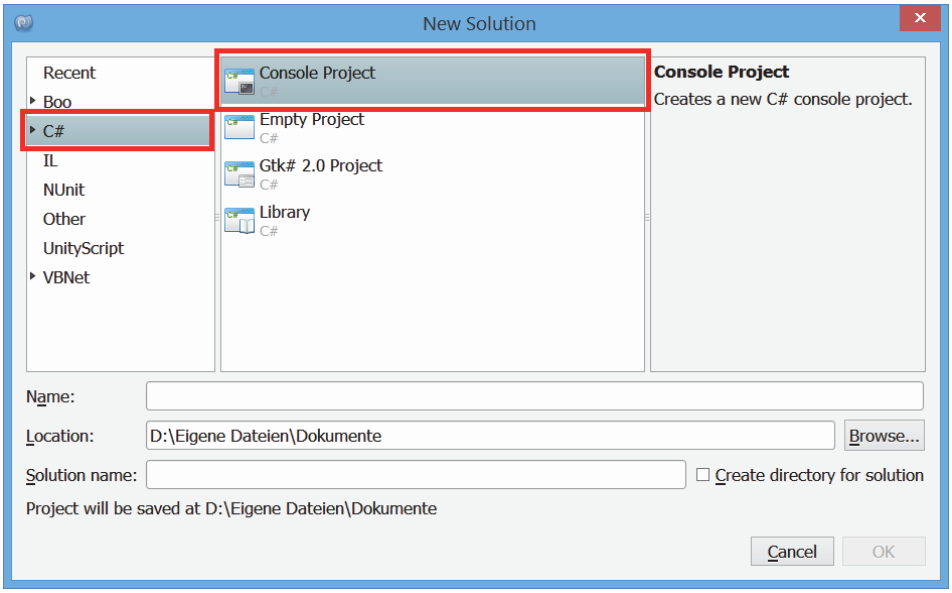
Nach einer Weile erscheint auf dem Bildschirm das Hauptarbeitsfenster von MonoDevelop:



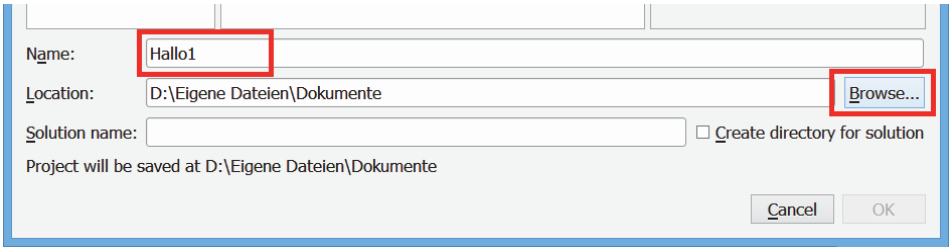
≫ Klicke nun auf die Schaltfläche NEW (oder im Menü auf FILE/NEW und SOLUTION).



≫ Links in der Liste suchst du den Eintrag C# und dann markierst du in der Mitte CONSOLE PROJECT.



≫ Gib einen Namen ein. Ich schlage `Hallo1` vor.

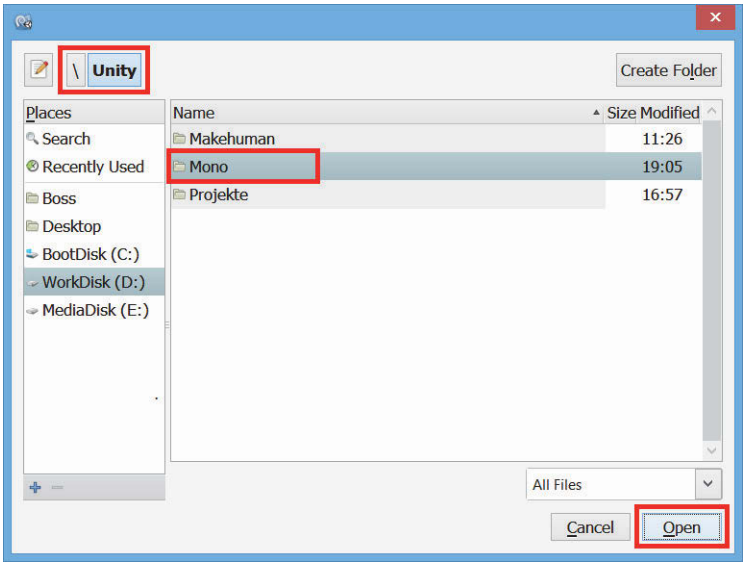


≫ Das Häkchen vor CREATE DIRECTORY FOR SOLUTION kannst du schon mal entfernen.

Wenn du willst, kannst du den Speicherort (LOCATION) stehen lassen. Oder du suchst dir einen anderen Ort aus, an dem du deine Projekte unterbringen willst. Ich habe dafür auf Laufwerk D: einen Ordner UNITY mit einem Unterordner MONO erzeugt.

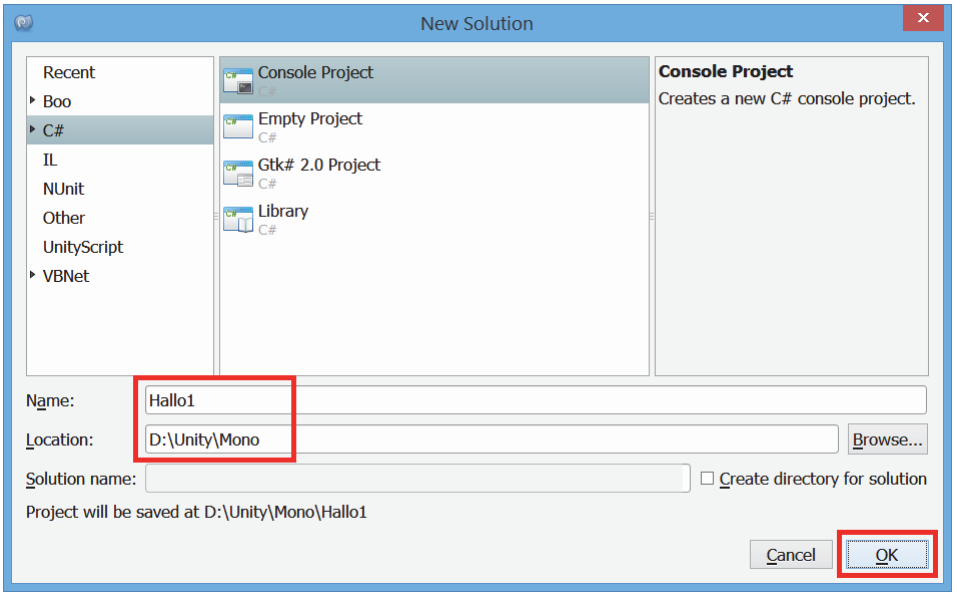
≫ Um das Zielverzeichnis (LOCATION) zu ändern, klickst du auf BROWSE.

Damit landest du in einem neuen Dialogfeld. Hier kannst du dich zum gewünschten Ordner durchklicken oder auch über CREATE FOLDER einen neuen erzeugen.

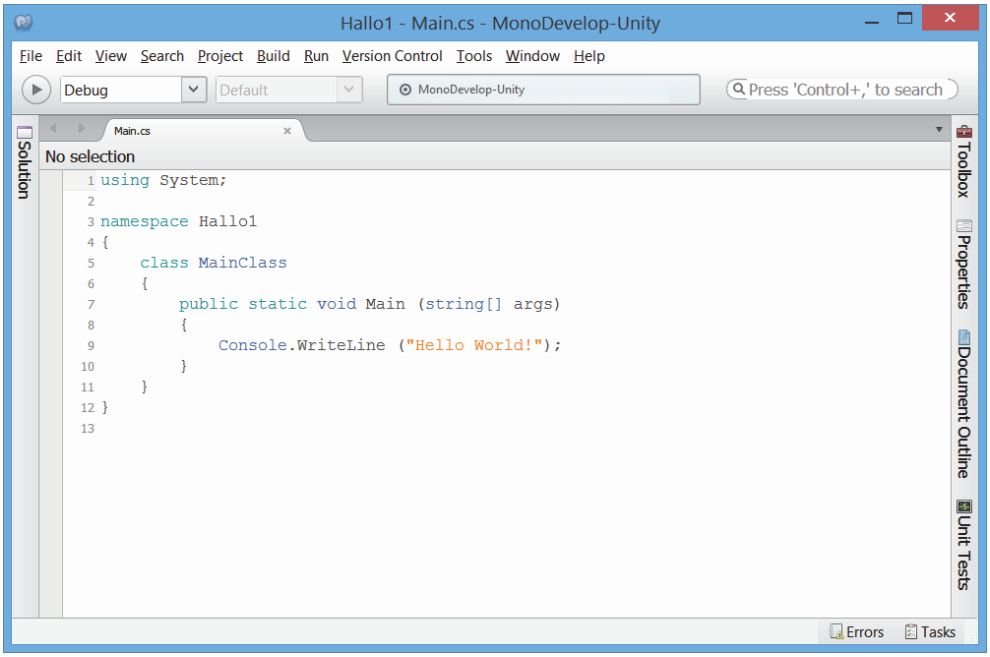


≫ Klicke abschließend auf OPEN.

≫ Zurück im vorigen Fenster kannst du dann auf OK klicken, um das Ganze abzuschließen.



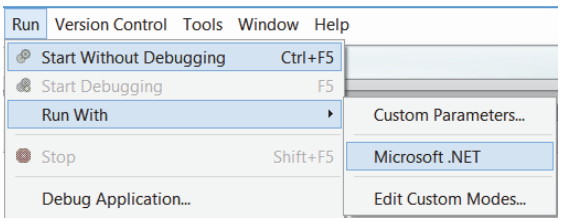
Und du bist wieder im Hauptfenster von MonoDevelop. Dort steht einiges an Text, mit dem du jetzt noch nicht allzu viel anzufangen weißt – es sei denn, du hast schon mal programmiert.



Bevor wir uns in den **Quelltext** – wie dieser Programmtext auch genannt wird – stürzen, schauen wir uns noch ein bisschen in seiner Umgebung um. Oben steht der Dateiname MAIN.CS. Das »CS« ist die Abkürzung für »C-sharp«, wie C# ausgesprochen wird. Es kennzeichnet Dateien für C#-Programme.

Die neu aufgetauchten Schaltflächen links und rechts an den Rändern des Textfensters sind für uns jetzt nicht von Belang. Du kannst aber gern mal überall draufklicken, um auszuprobieren, was sich tut. Und nicht zuletzt solltest du dieses Programm auch schon einmal starten.

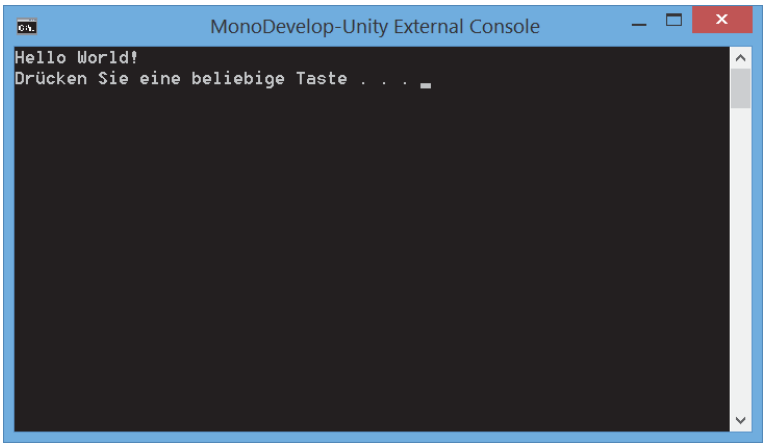
≫ Dazu klickst du im Hauptmenü auf RUN und START WITHOUT DEBUGGING. Oder du wählst die Option RUN WITH und dann im Zusatzmenü MICROSOFT.NET.



Kurz darauf öffnet sich ein kleines Extrafenster, auch Konsolenfenster genannt.

Unter der **Konsole** versteht man hier, dass alles ohne jeden Schnickschnack angezeigt wird, nämlich im Textmodus. Das heißt: Es gibt keine Grafik. Und die brauchen wir hier in diesem Crashkurs auch nicht, denn es geht allein darum, die Grundlagen von C# zu lernen. Mit Grafik bekommst du in den Buch-Kapiteln noch satt zu tun.

Dort siehst du, was das Programm bewirkt. Erst wird weiß auf schwarzem Hintergrund der Text »Hello World!« ausgegeben, darunter dann eine Zeile, die dich bittet irgendeine Taste zu drücken, um das Programm zu beenden. (Die erzeugt Windows automatisch am Programmende.)



≫ Such dir eine Taste aus und drücke sie. Damit schließt sich das Konsolenfenster wieder.

Einfach Hallo

Jetzt aber zu dem Quelltext, der mit dem Erstellen deines Projekts automatisch erzeugt wurde:

```
using System;

namespace Hallo1
{
  class MainClass
  {
    public static void Main (string[] args)
    {
```

```
      Console.WriteLine ("Hello World!");
    }
  }
}
```

Beschäftigen wir uns erst einmal kurz mit Klammern. Schon in diesen wenigen Zeilen tauchen gleich drei Sorten davon auf. C# ist bemüht, nahezu alles, was es an Zeichen auf der Tastatur gibt, zu verwenden, also auch alle Klammern – wie du in dieser Tabelle siehst:

	Bezeichnung	Tastenkombinationen
( )	runde Klammern	⇧+8 und ⇧+9
{ }	geschweifte Klammern	AltGr+7 und AltGr+0
[ ]	eckige Klammern	AltGr+8 und AltGr+9

Damit kennst du ihre Namen und weißt, wo du sie auf der Tastatur finden kannst. Auf die Bedeutung jedes Klammernpaares kommen wir im Laufe dieses Kapitels noch.

Sehr wichtig ist, dass es zu **jeder** öffnenden Klammer auch eine schließende Klammer geben muss! Oft ist es üblich, die öffnende geschweifte Klammer direkt ans Ende der ersten Zeile zu setzen. Wo genau du die Klammern positionierst, ist Geschmackssache. Der Programmteil mit den Klammern könnte also auch so aussehen:

```
namespace Hallo1 {
  class MainClass {
    public static void Main (string[] args) {
}}}
```

Aber nun widmen wir uns zeilenweise dem Quelltext unseres allerersten Projekts:

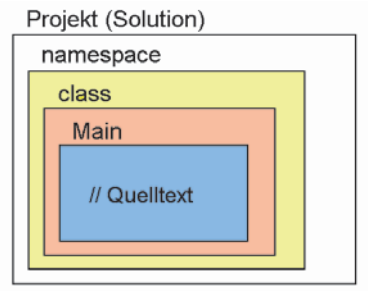
```
using System;
```

Der Standardwortschatz von C# ist eigentlich recht klein. Alles was darüber hinausgeht, steht in Extra-Bibliotheken, die mit der `using`-Anweisung in ein Programm eingebunden werden. So benutzt man nur das, was man auch wirklich braucht. `System` ist eine schon ziemlich mächtige Bibliothek, weitere wirst du im Laufe des Buches kennenlernen.

Das folgende ist dann das eigentliche Programm, das minimal so aussieht:

```
namespace Hallo1 {
  class MainClass {
    public static void Main (string[] args) {
    }
  }
}
```

Das erinnert irgendwie an ein »Zwiebelsystem«: Die Außenhaut ist das Projekt mit eigenem Ordner. Darin liegt der sogenannte Namensraum (englisch `namespace`), hier trägt er den Namen, den du deinem Projekt gegeben hast.



In diesem Paket finden wir die Daten einer Klasse (englisch: `class`). Hier sei schon einmal erwähnt, dass es natürlich mehr als eine Klasse geben kann.

Was ist eine **Klasse**? Was ein Objekt ist, weißt du sicher. Zum Beispiel ein konkretes Auto wie ein VW Golf. Oder du selbst.

Weil es mehr als ein Objekt eines Typs geben kann, wie im richtigen Leben auch, fasst man deren Eigenschaften und Verhaltens-Methoden bzw. Funktionen in einer **Klasse** zusammen. Und mit Hilfe einer Klasse lassen sich dann mehrere oder viele Objekte erzeugen.

So ließen sich aus einer (allgemeinen) Klasse Auto dann (konkrete) Objekte wie VW Golf oder Toyota Corolla ableiten. Und natürlich auch viele Golfs und viele Corollas. (Oder z. B. aus der Klasse Mensch jemand wie du.)

Innerhalb unserer Klasse mit dem Namen `MainClass` gibt es schon eine Methode namens `Main()`. Während `MainClass` hier also die Hauptklasse ist, kann man `Main()` als Hauptmethode bezeichnen.

Doch nun sollten wir uns mal anschauen, was ich oben unterschlagen habe.

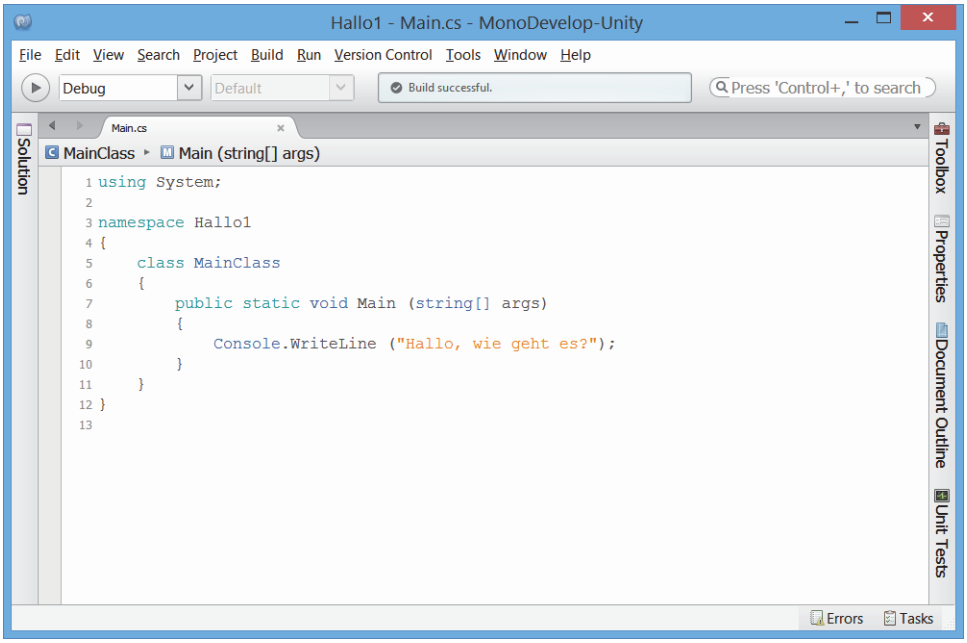
```
Console.WriteLine ("Hello World!");
```

Mit `Console` ist hier das Konsolenfenster gemeint, das nur Text anzeigen kann. Und `WriteLine()` ist eine weitere Methode. Und die dient dazu Text im Konsolenfenster auszugeben.

Wichtig ist: Jede (!) Methode bzw. Funktion hat am Ende zwei runde Klammern, die können auch leer sein, sie dürfen aber niemals fehlen. Steht etwas in den Klammern, so nennt man das **Argumente** oder **Parameter**.

So, und nun bist du dran. Du wirst jetzt das Programm ändern.

≫ Lösche den Text `"Hello World!"` Und ersetze ihn durch `"Hallo, wie geht es?"`.




```
1 using System;
2
3 namespace Hallo1
4 {
5     class MainClass
6     {
7         public static void Main (string[] args)
8         {
9             Console.WriteLine ("Hallo, wie geht es?");
10        }
11    }
12 }
13
```

≫ Starte das Programm über das RUN-Menü.

Und du wirst in einem Konsolenfenster (weiß auf schwarz) so begrüßt, wie du es vorher eingetippt hast.

Eine Variable namens Antwort

Sieht schon ganz nett aus, aber doch auch ziemlich mickrig. Zumal der Begrüßungstext eigentlich eine Antwort verlangt. Und dazu benötigen

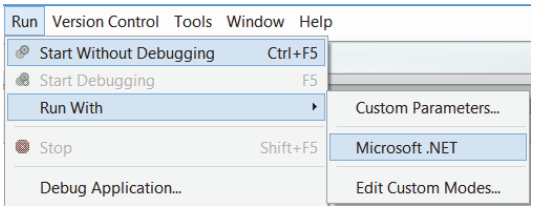
wir eine Methode, die quasi das Gegenteil von `WriteLine()` macht. Und tatsächlich gibt es die:

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

Doch wie soll sich der Computer deinen eingegebenen Text merken? Vielleicht so:

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

≫ Probiere das aus und starte das Programm über das RUN-Menü.



Und schon erntest du deine erste Fehlermeldung:

Der Name `Antwort` ist dem Programm offenbar unbekannt. Es handelt sich bei `Antwort` um eine sogenannte Variable, und das muss MonoDevelop wissen. `Antwort` muss also **vereinbart** werden.

Den Begriff **Variable** hast du sicher schon gehört. So genau erinnerst du dich aber nicht mehr, was das eigentlich ist? Aus dem Matheunterricht kennst du wahrscheinlich den Begriff **Platzhalter**. Die werden meist mit Buchstaben wie **x** oder **y** bezeichnet. Und weil Platzhalter in jeder Aufgabe einen anderen Wert annehmen können, also keinen von vornherein festgelegten Wert haben, nennt man so etwas Variablen (das Fremdwort »variabel« heißt auf Deutsch so viel wie »veränderlich«).

Im Gegensatz dazu gibt es natürlich in C# auch **Konstanten**. Die haben dann einen festgelegten Wert, der sich während des Programmlaufs nicht verändert. Und auch bei jedem neuen Programmstart behält eine Konstante ihren Wert.

Ein Beispiel ist der Text »Hallo, wer bist du?«. Aber auch Zahlen wie z. B. 0, 1, –1, 3.14 lassen sich als Konstanten einsetzen (wie du noch sehen wirst).

Der Name einer Variablen darf übrigens nicht mit einer Ziffer beginnen. (Probiere einfach aus, was geht!)

Und so wird der neue Name vereinbart:

```
String Antwort;
```

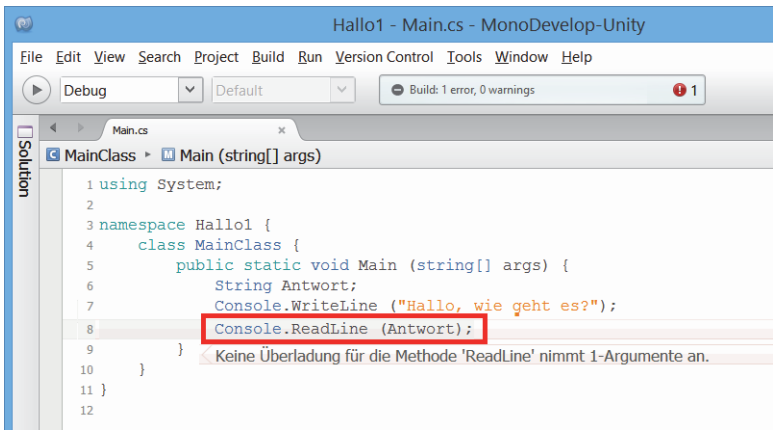
Unter einem `String` versteht man eine **Zeichenkette**. Und eine eingetippte Antwort ist ja eine Kette von Zeichen. Ebenso wie übrigens `"Hallo, wie geht es?"` ein String ist.

Damit hat die `Main`-Methode dieses Aussehen (→ HALLO1):

```
public static void Main (string[] args) {
  String Antwort;
  Console.WriteLine ("Hallo, wie geht es?");
  Console.ReadLine (Antwort);
}
```

≫ Ergänze den Quelltext entsprechend und starte das Programm erneut.

Und prompt wird der nächste Fehler gemeldet:



Was ist denn nun? Etwas, das du nicht wissen kannst. Hier ist die Lösung:

```
Antwort = Console.ReadLine ();
```

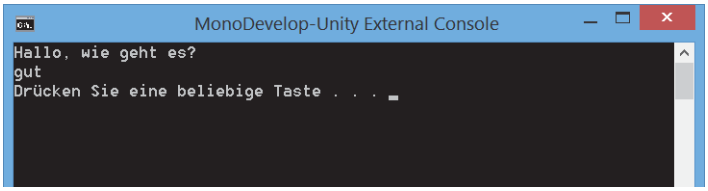
Die `ReadLine`-Methode hat keinen Parameter, die Klammern bleiben also leer. Aber hier passiert etwas, das man **Zuweisung** nennt: Erst liest die Methode den eingegebenen Text ein, dann legt sie ihn in der Variablen `Antwort` ab.

Bei einer Zuweisung steht links immer eine Variable, dann folgt ein Gleichheitszeichen (=), das man auch Zuweisungsoperator nennt.

Auf der rechten Seite kann entweder ein Wert stehen wie z.B. `Antwort = "gut";` oder etwas Komplizierteres wie z.B. die `ReadLine`-Methode.

≫ Korrigiere den Quelltext und lass das Ganze nochmal laufen.

Und endlich tut sich etwas. Und erst, wenn du eine Antwort eintippst, ist das Programm zu Ende.



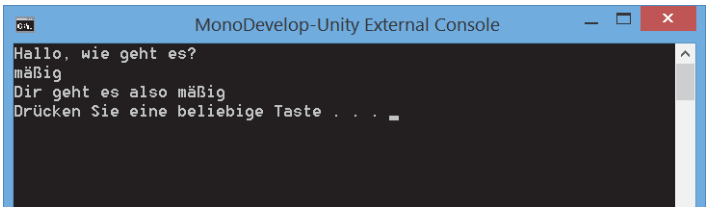
Sollte es aber noch nicht. Denn erst wollen wir schon wissen, ob sich der PC deine Antwort gemerkt hat.

≫ Füge also noch diese Zeile hinzu (→ HALLO1):

```
Console.WriteLine ("Dir geht es also " + Antwort);
```

Das Plus ist ein Verkettungsoperator, der dafür sorgt, dass die String-Konstante `"Dir geht es also "` und die String-Variable `Antwort` zu einer Zeichenkette werden.

Ein Programmlauf könnte dann so aussehen:



Auswertung

Kann der Computer eigentlich nicht mehr als nur etwas nachplappern? Wie wäre es, wenn er eine Antwort gibt, die zu unserer Antwort passt? Natürlich ist das möglich, und es führt uns zu dieser Hallo-Version (→ HALLO2):

```
using System;

namespace Hallo2 {
  class MainClass {
    public static void Main (string[] args) {
    String Antwort;
    Console.WriteLine ("Hallo, wie geht es?");
    Antwort = Console.ReadLine ();
    if (Antwort == "gut")
      Console.WriteLine ("Das freut mich!");
    if (Antwort == "schlecht")
      Console.WriteLine ("Das tut mir leid!");
    }
  }
}
```

Hier haben wir es mit einer **Kontrollstruktur** zu tun:

Wenn Antwort:	Dann zeige an:
gut	Das freut mich
schlecht	Das tut mir leid

Eingeleitet wird eine solche Struktur mit `if`, dann folgt in runden Klammern die **Bedingung**, schließlich die Anweisung, die bei erfüllter Bedingung ausgeführt werden soll.

```
if (Antwort == "gut")
  Console.WriteLine ("Das freut mich!");
if (Antwort == "schlecht")
  Console.WriteLine ("Das tut mir leid!");
```

Was hat es mit den doppelten Gleichheitszeichen (==) auf sich? Das ist einer von den Vergleichsoperatoren, die du hier in einer Tabelle zusammengefasst siehst:

<	kleiner	>=	größer oder gleich
>	größer	<=	kleiner oder gleich
==	gleich	!=	ungleich

Dem einfache Gleichheitszeichen (=) bist du ja schon begegnet: als Zuweisungsoperator.

Sollte eine Bedingung mal nicht erfüllt sein, gibt es noch den `else`-Zweig. Der bietet dann die Möglichkeit für die Ausführung alternativer Anweisungen. In unserem Fall könnte das z.B. so aussehen (→ HALLO3):

```
if (Antwort == "gut")
  Console.WriteLine ("Das freut mich!");
else
  Console.WriteLine ("Na ja ...");
```

Wird etwas anderes als »gut« eingetippt, so wird der zweite Text angezeigt.

Dass die `if`-Struktur auch bei Zahlen funktioniert, kannst du dir zwar denken, aber das sollten wir lieber ausprobieren. Bis jetzt hatten wir mit Zahlen ja noch nichts zu tun, sondern nur mit Zeichenketten (Strings).

Versuchen wir es gleich mit einem kleinen Spiel, in dem es um das Raten von Zahlen geht. Dazu benötigen wir zwei Zahl-Variablen, die so vereinbart werden:

```
int Zahl, Eingabe;
```

`int` ist die Abkürzung für »Integer«, womit die ganzen Zahlen gemeint sind. Als Nächstes benötigen wir Methoden, die einen Zufallszahlengenerator starten und eine zufällige ganze Zahl erzeugen. Die entsprechenden Zeilen könnten dann z.B. so aussehen:

```
Random Zufall = new Random();
Zahl = Zufall.Next(1000) + 1;
```

Und hier ist er wieder, der Zuweisungsoperator (=), der im Gegensatz zum Vergleichsoperator (==) aus nur einem Gleichheitszeichen besteht.

In der ersten Zeile wird nicht nur eine Variable namens `Zufall` vereinbart, sondern hier handelt es sich um ein Objekt der Klasse `Random`. Das Schlüsselwort `new` sorgt dafür, dass aus der Klasse ein Objekt entsteht, mit dessen Hilfe man Zufallszahlen erzeugen kann.

Kommen wir zur zweiten Zeile: Erst wird auf der rechten Seite eine Formel bearbeitet. `Next()` bewirkt, dass eine zufällige ganze Zahl zwischen 0 und 999 erzeugt wird, und durch Addition von 1 wie gewünscht zwischen 1 und 1000 liegt. Anschließend wird das Ergebnis der Variablen `Zahl` zugewiesen.

Damit gäbe es eine Zufallszahl, die nur der Computer kennt. Nun kommt die Zahl, die wir nachher beim Spielen eingeben. Und dann wird über mehrere `if`-Strukturen getestet, ob unsere Eingabe zu niedrig oder zu hoch liegt – oder ob sie passt. Hier ist unser erster Versuch (→ RATEN1):

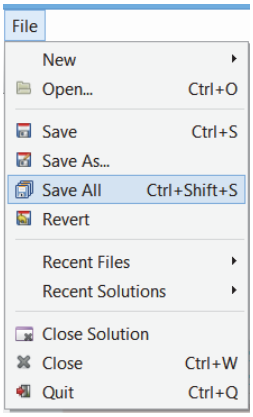
```
using System;

namespace Raten1 {
  class MainClass {
    public static void Main (string[] args) {
      int Zahl, Eingabe;
      Random Zufall =  new Random();
      Zahl = Zufall.Next(1000);
      Console.WriteLine
        ("Ich denke mir eine Zahl zwischen 1 und 1000");
      Console.WriteLine ("Rate mal: ");
      Eingabe = Console.ReadLine();
      if (Eingabe < Zahl)
        Console.WriteLine ("Zu klein!");
      if (Eingabe > Zahl)
        Console.WriteLine ("Zu groß!");
      if (Eingabe == Zahl)
        Console.WriteLine ("Richtig!");
```

```
        }
    }
}
```

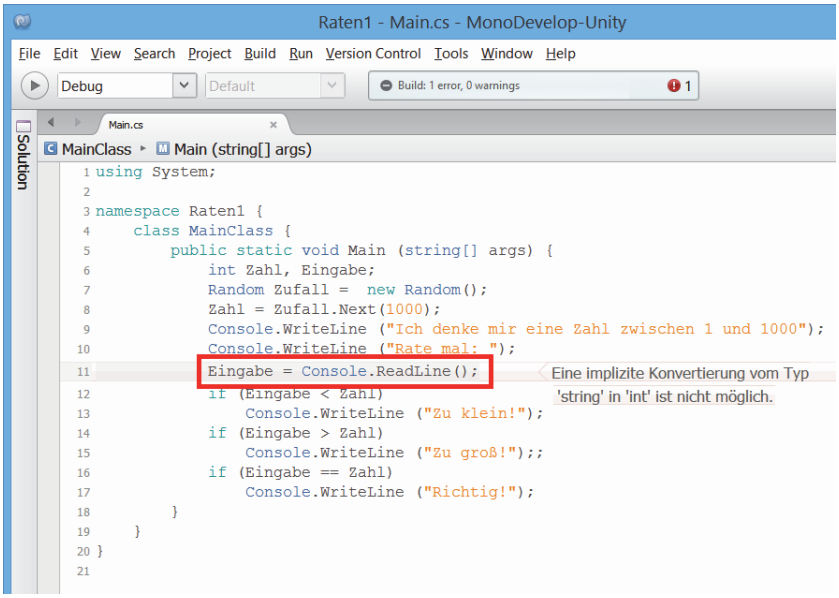
Es gibt drei Kontrollstrukturen, nur wenn Eingabe und Zahl übereinstimmen, wird »Richtig!« gemeldet.

≫ Erzeuge ein neues Projekt, tippe das Ganze ein und speichere es über FILE und SAVE ALL.



≫ Dann starte es über das RUN-Menü.

Du hast es geahnt, dass das Projekt nicht sofort fehlerfrei ist? Dann hattest du Recht:



Bemängelt wird diese Zeile:

```
Eingabe = Console.ReadLine();
```

Aber warum? Schauen wir uns mal an, was links und rechts vom Zuweisungsoperator steht:

```
Typ Zahl = Typ String;
```

Wie soll das gehen? Eine Zahl und eine Zeichenkette ist (nicht nur) für den Computer etwas verschiedenes. Man kann also nicht einfach mal eben einer Zahl einen String zuweisen (und auch umgekehrt geht es nicht).

Wir müssen also den eingetippten Text (auch wenn er aus Ziffern besteht) in eine Zahl umwandeln. Dabei helfen uns die Methoden der Klasse `Convert`:

```
Eingabe = Convert.ToInt32(Console.ReadLine());
```

Würde nicht `ToInt` genügen? Nein, weil es in C# ziemlich viele Zahlentypen gibt. Eine davon (die wir hier benutzen) heißt `int`. Die Zahl dahinter gibt die Größe der Variablen in Bit an, das ist der kleinste Speicherplatz, den es bei Computern gibt. 32-Bit heißt: Eine Variable des Typs `int` braucht 32 kleine Plätze im Arbeitsspeicher deines Computers (der viele Milliarden davon hat). Wie du in einem kleinen Überblick sehen kannst, gibt noch ein paar Umwandlungsfunktionen mehr (u.a. auch für Dezimalzahlen):

Typ	Funktion	Typ	Funktion
short	ToInt16()	float	ToSingle()
int	ToInt32()	double	ToDouble()
long	ToInt64()	String	ToString()

≫ Korrigiere die betreffende Zeile in deinem Quelltext. Dann starte das Projekt erneut.

Wiederholungen

Diesmal klappt es. Aber nicht wirklich: Denn wie soll man nach nur einem Mal Raten die passende Zahl finden?

Was wir brauchen, ist eine weitere **Kontrollstruktur**, allerdings hilft uns `if` diesmal nicht weiter, es muss eher etwas von dieser Art sein, Schleife genannt::

```
do {
  // Anweisungen
} while (Eingabe != Zufall);
```

Hier nutzen wir den Operator für »Ungleich«. Eingesetzt in unser Ratespiel sähe das dann so aus (→ RATEN2):

```
using System;

namespace Raten2 {
  class MainClass {
    public static void Main (string[] args) {
      int Zahl, Eingabe;
      Random Zufall =  new Random();
      Zahl = Zufall.Next(1000);
      Console.WriteLine
        ("Ich denke mir eine Zahl zwischen 1 und 1000");
      do {
        Console.WriteLine ("Rate mal: ");
        Eingabe = Convert.ToInt32
          (Console.ReadLine ());
        if (Eingabe < Zahl)
          Console.WriteLine ("Zu klein!");
        if (Eingabe > Zahl)
          Console.WriteLine ("Zu groß!");
      } while (Eingabe != Zahl);
      if (Eingabe == Zahl)
        Console.WriteLine ("Richtig!");
    }
  }
}
```

≫ Ergänze deinen Quelltext und starte das Programm.

Nun kannst du so lange raten, bis du die richtige Zahl gefunden hast. Bei dieser Schleife findet der Bedingungstest am Schluss statt. Aber es geht auch umgekehrt, eine Bedingung lässt sich auch am Anfang einer Schleife testen, womit sich der entsprechende Quelltext im Ratespiel so ändert (→ RATEN3):

```
using System;

namespace Raten3 {
  class MainClass {
    public static void Main (string[] args) {
      int Zahl = 0, Eingabe = 0;
      Random Zufall =  new Random();
      Zahl = Zufall.Next(1000);
      Console.WriteLine
        ("Ich denke mir eine Zahl zwischen 1 und 1000");
      while (Eingabe != Zahl) {
        Console.WriteLine ("Rate mal: ");
        Eingabe = Convert.ToInt32
          (Console.ReadLine ());
        if (Eingabe < Zahl)
          Console.WriteLine ("Zu klein!");
        if (Eingabe > Zahl)
          Console.WriteLine ("Zu groß!");
      }
      if (Eingabe == Zahl)
        Console.WriteLine ("Richtig!");
    }
  }
}
```

Und so sieht diese Schleifenart aus:

```
while (Eingabe != Zufall) {
  // Anweisungen
}
```

Es kann aber tückisch sein, wenn die betroffenen Variablen am Anfang keine klar definierten Werte haben. Bei einer Vereinbarung nämlich reserviert C# nur den nötigen Speicherplatz. Damit aber hier auch die Variable `Eingabe` einen passenden Wert hat (`Zahl` bekommt ja seinen Zufallswert), sollten wir schon bei der Vereinbarung einen Startwert festlegen:

```
int Zahl = 0, Eingabe = 0;
```

Wie du siehst, habe ich das auch gleich für Zahl erledigt (auch wenn es nicht unbedingt nötig ist). Generell gilt: Haben alle Variablen einen Startwert, geht man auf Nummer sicher.

Welchen Sinn hat es, die Wiederholungsbedingung mal an den Anfang, mal ans Ende einer Schleife zu setzen? Wird am Eingang eine Bedingung überprüft, kann es sein, dass die Schleife **sofort übersprungen** wird (wenn nämlich die Bedingung nicht erfüllt wurde). Beim Bedingungstest am Ausgang wurde die Schleife jedoch **mindestens einmal** durchlaufen.

Kapital und Zinsen

Jetzt geht es weniger ums Raten, aber dafür umso mehr ums Rechnen. Aber keine Bange: Nur ein bisschen Zinsrechnung ist jetzt angesagt: Du gibst ein, welches Kapital du einsetzen willst, wie viel Prozent Zinsen dir die Bank bietet und wie lange du dein Geld liegen lassen willst.

Als Erstes benötigen wir dafür diese Zahlvariablen:

```
float Kapital, Zinsen, Prozent;
int Laufzeit;
```

Weil es sich nur bei `Laufzeit` (in Jahren) um eine Ganzzahl handelt, wurde auch nur diese Variable als `int` vereinbart. Bei den anderen Variablen geht es um Geld, daher sind sie allesamt als Dezimalzahlen bzw. »Gleitpunktzahlen« vereinbart.

Dabei gibt es in C# mehrere Möglichkeiten: `float` sind die einfachen Dezimalzahlen mit meistens ausreichender Genauigkeit. Sollen die Stellen hinter dem Dezimalpunkt (Komma) einmal besonders hoch sein, lässt sich auch `double` verwenden. Das kann die Rechengenauigkeit erhöhen. (Durch die vielen möglichen Nachkommastellen lassen sich Rundungsfehler z. B. beim Dividieren von Zahlen verringern.)

Und hier ist erst einmal der gesamte schon recht umfangreiche Quelltext (→ GELD1):

```
using System;

namespace Geld1 {
  class MainClass   {
    public static void Main (string[] args) {
```

```
      float Kapital, Zinsen, Prozent;
      int Laufzeit;
      // Eingabe der Daten
      Console.WriteLine
        ("Gib das Kapital in Euro ein    : ");
      Kapital = Convert.ToSingle (Console.ReadLine());
      Console.WriteLine
        ("Gib den Zinssatz in Prozent ein: ");
      Prozent = Convert.ToSingle (Console.ReadLine());
      Console.WriteLine
        ("Gib die Laufzeit in Jahren ein : ");
      Laufzeit = Convert.ToInt32
        (Console.ReadLine());
      // Berechnungsschleife
      for (int i = 1; i <= Laufzeit; i++) {
        Zinsen  = Kapital * Prozent / 100;
        Kapital = Kapital + Zinsen;
      }
      // Ausgabe des Ergebnisses
      Console.WriteLine
       ("Das ist dann auf dem Konto: ");
      Console.WriteLine (Kapital + " Euro");
    }
  }
}
```

Nach der Vereinbarung der Variablen geht es gleich zur Dateneingabe, wobei das schon bekannte Methodenpaar `WriteLine()` und `ReadLine()` gut zusammenarbeitet

Ausgabe	Eingabe	Umwandlung
"Gib das Kapital in Euro ein : "	Kapital	ToSingle()
"Gib den Zinssatz in Prozent ein: "	Prozent	ToSingle()
"Gib die Laufzeit in Jahren ein : "	Laufzeit	ToInt32()

Bei den Umwandlungen brauchen wir jetzt mit `ToSingle()` für die Dezimalzahlen eine andere Methode.

Damit das Ganze etwas aufgelockerter wird und man als Programmierer eine besseren Überblick behält, lassen sich Zeilen mit eigenen Bemer-

kungen einfügen. Damit der Computer sie überliest, müssen sie markiert werden:

```
// Eingabe der Daten
// Berechnungsschleife
// Ausgabe des Ergebnisses
```

Ein solcher **Kommentar** muss stets durch zwei direkt aufeinander folgende Schrägstriche (//). eingeleitet werden. Kommentare dürfen überall vorkommen, also auch hinter einer Anweisung, z.B.:

```
Console.WriteLine ("Hallo!");  // kurzer Gruß
```

Eine andere Möglichkeit, einen Kommentar in den Quelltext einzufügen: ist diese:

```
/* Kurzer Gruß */
```

Der Vorteil einer solchen Klammerung mit /* und */ ist, dass Erläuterungen über beliebig viele Zeilen gehen können.

Kontrolle muss sein

Kommen wir nun zu dem, was hier neu ist. Hier gibt es wieder eine **Kontrollstruktur**, diesmal aber mit dem Wörtchen for. Dabei handelt es sich um eine Zählschleife. FÜR eine bestimmte Bedingung sollen die nachfolgenden Anweisungen ausgeführt werden:

```
for (int i = 1; i <= Laufzeit; i++) {
  Zinsen  = Kapital * Prozent / 100;
  Kapital = Kapital + Zinsen;
}
```

Den Inhalt in den runden Klammern hinter for nehmen wir gleich unter die Lupe. Kümmern wir uns erst um das Innere der geschweiften Klammerung:

Im ersten Schritt werden die Zinsen für ein Jahr berechnet, im nächsten wird das Kapital um diese Zinsen erhöht. Dabei kommen fast alle Rechenoperatoren zum Einsatz:

Addition	Subtraktion	Multiplikation	Division
+	-	*	/

Angenommen, du gibst für `Laufzeit` den Wert 10 ein, dann werden diese Berechnungen genau zehnmal ausgeführt. (Auf die Null solltest du bei der Eingabe verzichten.)

Der Startwert ist 1 (`i = 1`), der Zielwert (`Laufzeit`) wäre 10. Damit sind wir bei den drei Parametern der `for`-Schleife:

```
// Variable i vereinbaren und auf 1 setzen
int i = 1;
// Testen, ob i kleiner oder gleich Laufzeit
i <= Laufzeit;
// Wert von i um 1 erhöhen
i++;
```

In der Mitte liegt die eigentliche Bedingung, davor der Startwert und dahinter die Zählung. Wie du hier sehen kannst, wurde die Vereinbarung der Zählvariablen direkt in die Parameterliste gepackt. Das `i` steht als Abkürzung für »Index«, für Zählvariablen werden meist nur Buchstaben wie i, j, k benutzt.

Diese »Mittendrin«-Vereinbarung ist nicht nur reine Bequemlichkeit, sie macht eine Variable auch nur in diesem Block gültig. Man nennt das **lokale Variable**. Eine Zählvariable wird ja auch nur innerhalb der `for`-Schleife gebraucht.

Denkbar wäre auch diese Möglichkeit, wobei die Variable allerdings über den `for`-Block hinaus gilt.

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

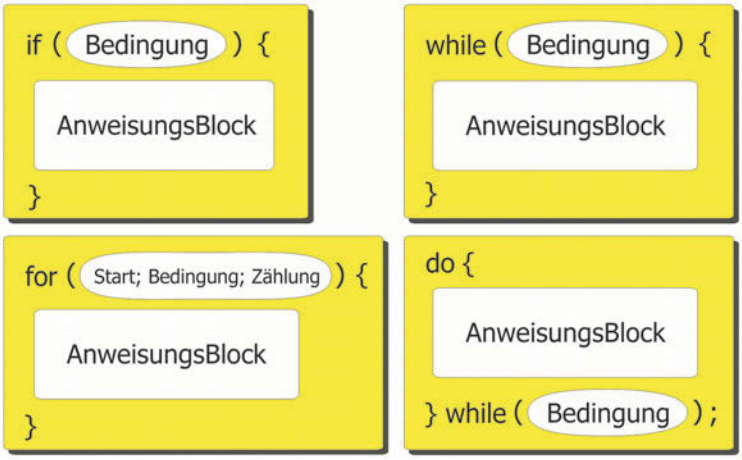
Die übrigen Variablen, die wir bisher verwendet haben gelten überall **innerhalb** der gesamten Hauptmethode (die ja durchaus recht umfangreich sein kann).

Was noch fehlt, ist eine Anmerkung zum `i++`: Das könnte eigentlich auch so aussehen:

```
i = i + 1;  // man sieht, dass das keine Gleichung ist
```

In C# geht es eben kürzer – übrigens auch mit Minus, da heißt es dann: `i--`.

Wir haben es bis jetzt mit zwei wichtigen Kontrollstrukturen zu tun gehabt: Das eine wird auch als **Verzweigung** bezeichnet, das andere als **Schleife**.



Ständig geht es dabei um **Bedingungen**, die erfüllt sein müssen:

- WENN eine Bedingung erfüllt ist, DANN führe Anweisungen aus (`if`-Struktur)
- FÜR eine bestimmte Anzahl bzw. Bedingung führe Anweisungen aus (`for`-Struktur)
- SOLANGE eine Bedingung erfüllt ist, führe Anweisungen aus (`while`-Struktur)
- Führe eine Anweisung aus, SOLANGE eine Bedingung erfüllt ist (`do-while`-Struktur)

Besteht ein Anweisungsblock aus mehr als einer Anweisung, so muss er mit geschweiften Klammern markiert werden. (Wenn du aber auch nur eine einzige Anweisung schon umklammern möchtest, gilt das nicht als Fehler.)

Eigene Methoden

Eine Besonderheit und Stärke (nicht nur) von C# ist die Fähigkeit, mit Objekten umzugehen.

Natürlich können wir hier auch unsere eigenen Klassen vereinbaren und Objekte erstellen. Genau dies wollen wir jetzt an einem Beispiel ausprobieren. Und zwar geht es um ein Rechteck, genauer um die Berechnung seines Flächeninhalts und seines Umfangs. Beginnen wir mit dem klassischen Weg, der etwa so aussehen könnte (→ RECHTECK1):

```
using System;

namespace Rechteck1 {
  class MainClass {
    public static void Main (string[] args) {
      float Breite, Hoehe, Flaeche, Umfang;
      // Eingabe der Daten
      Console.WriteLine
        ("Wie breit ist das Rechteck? : ");
      Breite = Convert.ToSingle(Console.ReadLine());
      Console.WriteLine
        ("Wie hoch ist das Rechteck?  : ");
      Hoehe = Convert.ToSingle(Console.ReadLine ());
      // Berechnungen
      Flaeche = Breite * Hoehe;
      Umfang  = 2 * (Breite + Hoehe);
      // Ausgabe der Ergebnisse
      Console.WriteLine
        ("Flaeche: " + Convert.ToString (Flaeche));
      Console.WriteLine
        ("Umfang : " + Convert.ToString (Umfang));
    }
  }
}
```

An sich nichts Besonderes, auf jeden Fall nichts Neues. Wieder mal auffällig – für mich – ist jedoch die Unterteilung des Programms in **Eingabe**, Berechnung (bzw. **Verarbeitung**, wie man auch sagt) und **Ausgabe**. Das hatten wir auch schon beim letzten Projekt.

Hier möchte ich diese drei Teile einmal in eigene Methoden packen. Wie diese »Kapselung« vonstattengeht, siehst du hier (→ RECHTECK2):

```
using System;

namespace Rechteck2 {
  class MainClass {

    static float Breite, Hoehe, Flaeche, Umfang;

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

```
      InputData();
      ProcessData();
      OutputData();
    }

    // Eingabe der Daten
    static void InputData () {
      Console.WriteLine ("Breite: ");
      Breite = Convert.ToSingle (Console.ReadLine ());
      Console.WriteLine ("Höhe: ");
      Hoehe = Convert.ToSingle (Console.ReadLine ());
    }
    // Berechnungen
    static void ProcessData () {
      Flaeche = Breite * Hoehe;
      Umfang = 2 * (Breite + Hoehe);
    }
    // Ausgabe der Ergebnisse
    static void OutputData () {
      Console.WriteLine ("Flaeche: "
        + Convert.ToString (Flaeche));
      Console.WriteLine ("Umfang : "
        + Convert.ToString (Umfang));
    }
  }
}
```

Sieht auf den ersten Blick üppiger aus und ist es auch. Aber schauen wir mal in die Hauptmethode `Main()`. Die ist ja nun wirklich recht mager geworden:

```
InputData();
ProcessData();
OutputData();
```

Sie besteht nur noch aus drei Anweisungen, die wir selbst »erfunden« bzw. definiert haben – siehe weiter unten im Quelltext.

Dieses Verfahren unterteilt große Programme in kleinere »verdauliche« Happen, indem aus einzelnen Abschnitten eigene Methoden werden, wie z.B.:

```
static void InputData() { }
static void ProcessData() { }
static void OutputData() { }
```

Ob du dich dabei für englische oder deutsche Namen entscheidest, ist deine Sache. Die Grundstruktur einer selbst vereinbarten Funktion ist die gleiche wie die der Hauptmethode `Main()`:

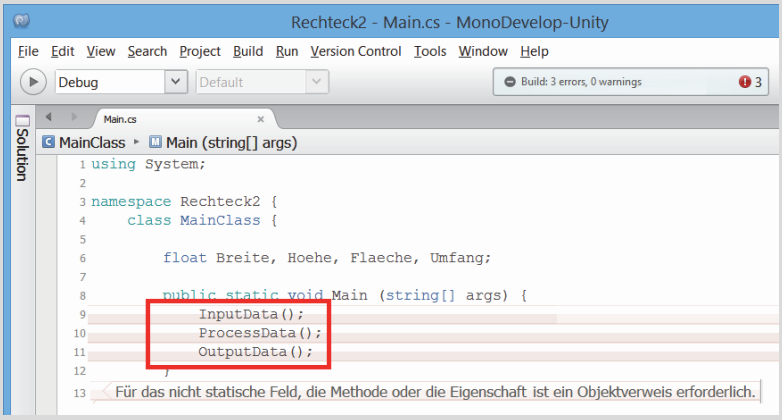
```
static void Name() {
  // Anweisungen
}
```

Was in unserem einfachen Beispiel aufwändiger erscheint, entpuppt sich bei umfangreichen Programmen als Segen: Man behält leichter den Überblick und im Falle von Problemen kann man die mögliche Fehlerstelle leichter lokalisieren.

Apropos »lokal«: Wie du hier siehst, ist die Vereinbarung der Variablen für `Breite` und `Höhe` wie `Fläche` und `Umfang` aus der `Main`-Funktion heraus ganz nach oben gewandert. Damit gelten diese Variablen hier als **global**, sind also außer in `Main()` in jeder anderen Methode bekannt, die in dieser Klasse (`MainClass`) vorkommt:

```
static float Breite, Hoehe, Flaeche, Umfang;
```

Ohne das vorgesetzte `static` geht es hier nicht. Sonst erntest du einige Fehlermeldungen.



Und wenn dann jede Methode als `static` definiert wurde, hagelt es weitere Fehlermeldungen, bis auch die globale Vereinbarung der Variablen ihren `static`-Vorsatz hat.

Eine Klasse für sich

Aber was ist denn nun mit dem Rechteck als Klasse? Das wollen wir nicht aus den Augen verlieren und kommen damit zum nächsten Quelltext. Zuerst vereinbaren wir ein neues Klassengerüst:

```
class Rectangle {
  // Elemente und Methoden
}
```

Das steht oberhalb der Vereinbarung der Hauptklasse. Und dorthin verschieben wir nun alle drei Methoden, die wir selbst definiert haben, ebenso wie die Vereinbarungszeile mit den Variablen:

```
class Rectangle {
  static float Breite, Hoehe, Flaeche, Umfang;
  // Eingabe der Daten
  static void InputData () {
    Console.WriteLine ("Breite: ");
    Breite = Convert.ToSingle (Console.ReadLine());
    Console.WriteLine ("Höhe: ");
    Hoehe = Convert.ToSingle (Console.ReadLine());
  }
  // Berechnungen
  static void ProcessData () {
    Flaeche = Breite * Hoehe;
    Umfang = 2 * (Breite + Hoehe);
  }
  // Ausgabe der Ergebnisse
  static void OutputData () {
    Console.WriteLine
      ("Flaeche: " + Convert.ToString (Flaeche));
    Console.WriteLine
      ("Umfang : " + Convert.ToString (Umfang));
  }
}
```

Womit die Klasse `MainClass` ziemlich leer geworden ist:

```
class MainClass {
  public static void Main (string[] args) {
    InputData();
```

```
    ProcessData();
    OutputData();
  }
}
```

Allerdings funktioniert natürlich nun unser ganzes Programm nicht mehr, denn die drei in `Main()` aufgerufenen Methoden sind nur **innerhalb** der Klasse `Rectangle` bekannt. Erst müssen wir jetzt aus der neuen Klasse ein Objekt erzeugen. Und das geht so:

```
Rectangle Rechteck = new Rectangle();
```

Du kennst das bereits: Beim Ratespiel hatten wir auch schon ein Objekt vereinbart, bloß von einer bereits in C# vorhandenen Klasse:

```
Random Zufall = new Random();
```

Ich habe ganz da oben in der Abbildung etwas gemogelt. In Wirklichkeit gibt es eine Methode, die **Konstruktor** genannt wird.

Sie hat immer den gleichen Namen wie eine Klasse. Klasse und Konstruktor gehören fest zusammen. Der Konstruktor dient dazu, alles zu erledigen, was für die Erzeugung eines Objekts nötig ist.

Nachdem wir das Objekt `Rechteck` erzeugt haben, müssen wir seine Methode aufrufen. Das klappt so aber nicht:

```
InputData();
ProcessData();
OutputData();
```

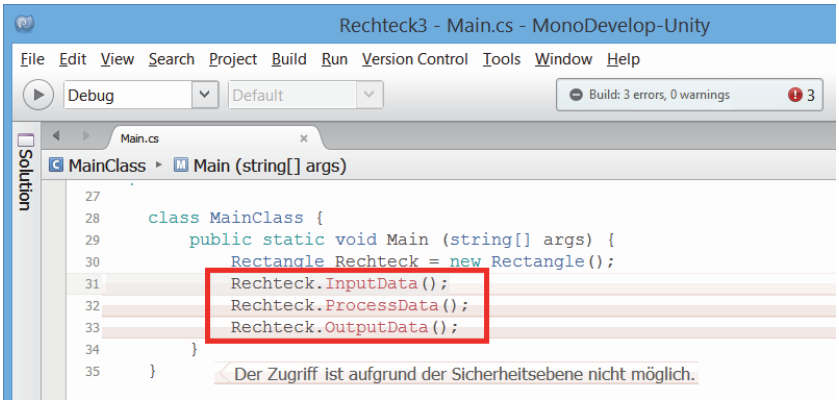
Sondern irgendwie muss `Main()` ja erfahren, dass diese drei Methoden nun zum Objekt `Rechteck` gehören. Und das sieht dann so aus (→ RECHTECK3):

```
Rechteck.InputData();
Rechteck.ProcessData();
Rechteck.OutputData();
```

Der Punkt, der das Objekt und seine Methode verbindet, heißt Verknüpfungsoperator.

≫ Und nun tippe das Ganze erst einmal ein und probiere dann das Programm aus.

Verärgert? Schon wieder diese Fehlermeldungen.



Sicherheitsebene? Ja, so was gibt es in C#, Datenschutz. Auf die drei Methoden kann nicht zugegriffen werden, weil sie als privat gelten. So wie sie definiert wurden, sind sie allesamt nicht von außen zugänglich. Das ändert sich sofort, wenn wir das Wort `static` durch `public` ersetzen (→ RECHTECK3):

```
class Rectangle {
  private float Breite, Hoehe, Flaeche, Umfang;
  // Eingabe der Daten
  public void InputData () {
    Console.WriteLine ("Breite: ");
    Breite = Convert.ToSingle (Console.ReadLine());
    Console.WriteLine ("Höhe: ");
    Hoehe = Convert.ToSingle (Console.ReadLine());
  }
  // Berechnungen
  public void ProcessData () {
    Flaeche = Breite * Hoehe;
    Umfang = 2 * (Breite + Hoehe);
  }
  // Ausgabe der Ergebnisse
  public void OutputData () {
    Console.WriteLine
      ("Flaeche: " + Convert.ToString (Flaeche));
    Console.WriteLine
```

```
      ("Umfang : " + Convert.ToString (Umfang));
  }
}
```

Wenn du genau hinschaust, dann wirst du zwei Wörter entdecken:

public	Element einer Klasse ist auch außerhalb (öffentlich) nutzbar
private	Element einer Klasse ist nur innerhalb (privat) nutzbar

Die Variablen Breite, Hoehe, Flaeche und Umfang werden nur in den Methoden der Klasse Rectangle verwendet, also müssen sie nicht public sein. Aber die Methoden müssen auch »draußen« nutzbar sein, deshalb sind sie öffentlich. (Das static spielt also hier keine Rolle und entfällt komplett.)

Parameter und Rückgabewerte

Die Methoden, die wir bis jetzt erzeugt haben, sahen allesamt so aus:

```
void MachMal () {
  // Anweisungen
}
```

Davor konnte dann noch ein public stehen, wenn die Methode öffentlich sein sollte. Ansonsten galt sie als private. Doch die Klammern hinter dem Methodennamen blieben stets leer. Dabei kennst du doch mindestens eine Methode mit gefüllten Klammern, z.B.:

```
Console.WriteLine (Irgendwas);
```

Und auch die Hauptmethode hat irgendwas in Klammern zu übernehmen:

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

Wobei das erste ein Methodenaufruf ist und das zweite eine Methodendefinition. Wir nutzen jetzt erneut das Rechteck-Projekt und erstellen zwei Methoden mit Parametern:

```
private float GetArea (float bb, float hh) {
  return bb * hh;
}
```

```
private float GetPerimeter (float bb, float hh) {
  return 2 * (bb + hh);
}
```

Beide Methoden übernehmen die Breite und Höhe eines Rechtecks, dazu sind in der Parameterliste, wie man das Ganze in den Klammern auch nennt, zwei Zahlen vom Typ `float` vereinbart. Und innerhalb der Methode werden dann Flächeninhalt und Umfang des Rechtecks ermittelt.

Das ist aber nicht alles. Wo ist das `void` geblieben? Das hast du einfach so hingenommen. Und erst jetzt erfährst du, was es bedeutet: `void` macht eine Methode typlos, sie führt einfach eine oder mehrere Anweisungen aus. Das gilt für Methoden wie `WriteLine()` oder `Main()` ebenso wie für `InputData()` und `OutputData()`. Man nennt solche Methoden auch **Prozeduren**.

Anders ist es z.B. bei der Methode `ReadLine()`. Die führt auch eine Anweisung aus, doch sie gibt auch am Ende eine Wert zurück, den man z.B. einer Variablen zuweisen kann:

```
Eingabe = Console.ReadLine();
```

Auch die ganzen Umwandlungsfunktionen wie `ToString()` oder `ToSingle()` liefern einen Wert zurück. Und ebenso ist es mit den beiden selbstdefinierten Methoden `GetArea()` und `GetPerimeter()`. Die haben einen Typ (`float`) und geben über `return` das Ergebnis ihrer Berechnung zurück. Zum Beispiel an eine Zahlvariable:

```
Flaeche = GetArea (Breite, Hoehe);
Umfang = GetPerimeter (Breite, Hoehe);
```

Oder man verwendet sie direkt in der Ausgabe, wie du gleich sehen wirst. Diese Methoden nennt man auch **Funktionen**.

Hier nochmal der Unterschied zwischen Methoden mit und ohne Typ:

```
void MachEinfachNur () {
  // Anweisungen
}
Typ MachUndGibZurueck () {
  // Anweisungen
  return Wert;
}
```

Eine Funktion muss **immer** eine `return`-Zeile haben!

Und jetzt kommt alles zusammen in einem großen Block (→ RECHTECK4):

```
namespace Rechteck4 {
  // eigene Klasse
  class Rectangle {
    private float Breite, Hoehe;
    // Eingabe der Daten
    public void InputData () {
      Console.WriteLine ("Breite: ");
      Breite = Convert.ToSingle (Console.ReadLine());
      Console.WriteLine ("Höhe: ");
      Hoehe = Convert.ToSingle (Console.ReadLine());
    }
    // Berechnung der Fläche
    private float GetArea (float bb, float hh) {
      return bb * hh;
    }
    // Berechnung des Umfangs
    private float GetPerimeter (float bb, float hh) {
      return 2 * (bb + hh);
    }
    // Ausgabe der Ergebnisse
    public void OutputData () {
      Console.WriteLine ("Flaeche: " + Convert.ToString
        (GetArea (Breite, Hoehe)));
      Console.WriteLine ("Umfang : " + Convert.ToString
        (GetPerimeter (Breite, Hoehe)));
    }
  }
  // Hauptklasse und -funktion
  class MainClass {
    public static void Main (string[] args) {
      Rectangle Rechteck = new Rectangle();
      Rechteck.InputData();
      Rechteck.OutputData();
    }
  }
}
```

Wie du siehst, werden in `Main()` nur noch zwei Methoden aufgerufen. Alles andere wird klassenintern erledigt. Wobei die Methode `OutputData()` eine Menge Arbeit leisten muss:

```
Console.WriteLine ("Flaeche: " + Convert.ToString
  (GetArea (Breite, Hoehe)));
Console.WriteLine ("Umfang : " + Convert.ToString
  (GetPerimeter (Breite, Hoehe)));
```

Erst wird jeweils die Funktion zum Ermitteln von Fläche oder Umfang aufgerufen, dann die Umwandlungsfunktion, und erst dann kann `WriteLine()` das Ergebnis anzeigen.

≫ Wenn du das alles eingetippt hast und das Programm startest, wirst du feststellen, dass es weiterhin dasselbe tut wie die allererste Version. Aber du hast dich weiterentwickelt und kannst jetzt schon recht gut mit Klassen, Objekten und Methoden umgehen.

Zusammenfassung

Puh! Dies war nun notgedrungen ein ganz schön dickes Kapitel, beim Versuch, alles Wichtige in Kürze zu vermitteln, wurde der Koffer eben immer praller. Trotzdem kann man nicht sagen, dass du nun völlig fit in C# bist. Doch du wirst in den Kapiteln des Buchs genug Gelegenheit haben, deine Programmierfähigkeiten in dieser Sprache zu trainieren.

Immerhin verfügst du schon einmal über ein solides Grundwissen in C#. So kannst du einiges an Datentypen einsetzen:

`int`	Ganze Zahl
`float`	Einfachgenaue Dezimalzahl
`double`	Doppeltgenaue Dezimalzahl
`String`	Zeichenkette, String

Außerdem gibt es da zahlreiche Symbole wie Klammern und Operatoren:

`{ }`	Marken für Anweisungsblöcke
`( )`	Klammern u.a. für Parameter/Bedingungen
`=`	Zuweisungsoperator

==; !=	Operatoren für gleich und ungleich
<; >	Operatoren für größer und kleiner
.	Operator für Klassen, Objekte, Methoden
// /* */	Kommentar-Marken

Auch hast du eine ganze Reihe von Kontrollstrukturen kennengelernt:

if	Verzweigung
else	Verzweigung
while	Schleife mit Anfangstest
do-while	Schleife mit Schlusstest
for	u.a. Zählschleife

Und dir sind Möglichkeiten bekannt, wie man Klassen und Methoden vereinbart und benutzt:

class	Klasse definieren
using	Bibliotheken einbinden
return	Rückgabewert einer Methode (Funktion)
private	Zugriff nur innerhalb einer Klasse
public	Zugriff von überall im ganzen Programm
new	Neues Objekt erzeugen
void	Methode ohne Typ (Prozedur)

Aus der System-Bibliothek kennst du bereits einige Klassen:

Console	Klasse u.a. für Ein- und Ausgabe von Daten
WriteLine()	Methode zum Anzeigen von Text (und Zahlen)
ReadLine()	Methode zur Eingabe von Text (und Zahlen)
Convert	Klasse für die Typumwandlung
ToInt32()	Methode, um String in int-Zahl umzuwandeln
ToSingle()	Methode, um String in float-Zahl umzuwandeln
ToString()	Methode, um Zahl in String umzuwandeln
Random	Klasse für Zufallszahlen
Next()	(Nächste) Zufallszahl holen/erzeugen
Main()	Hauptmethode (für Konsolenanwendungen)

Ein paar Fragen ...

1. Was sind die wichtigsten Datentypen?
2. Welche Kontrollstrukturen kennst du?
3. Was genau ist der Unterschied zwischen einer `while`-Struktur und einer `do-while`-Struktur?
4. Erkläre den Unterschied zwischen Prozedur und Funktion.
5. Was sind Parameter?
6. Wie wird eine Klasse definiert, wie ein Objekt erzeugt?

... und ein paar Aufgaben

1. Erweitere das Hallo-Projekt so, dass es auf die Frage »Wie geht es dir?« mindestens fünf Antwortmöglichkeiten gibt.
2. Vereinbare im Raten-Projekt eine weitere Variable, die die Anzahl der Rateversuche mitzählt und sie am Schluss anzeigt.
3. Erstelle ein kleines Projekt, in dem der Kehrwert einer Zahl berechnet wird. Ggf. soll die Eingabe so lange wiederholt werden, bis die entsprechende Zahl keine Null ist.
4. Ändere das Geld-Projekt so um, dass nach der Eingabe von Kapital und Zinssatz berechnet wird, wie lange es dauert, bis aus dem Kapital eine Million geworden ist.
5. Erstelle ein neues Projekt mit jeweils einer Klasse `TQuadrat` und `TKreis`.

[Die Lösungen findest du auf der DVD im Ordner PROJEKTE.]