Programmieren mit

Mag. Stefan Hagmann
BG und BRG Frauengasse, Baden

Diese Dokument wird unter folgenden creative commons veröffentlicht:
http://creativecommons.org/licenses/by-nc-sa/3.0/at/

Inhaltsverzeichnis

1. Voraussetzungen

Die Voraussetzungen sind die folgenden

- ein installiertes .NET Framework bzw. MONO
- Ein Editor wie
  - Visual Studio Express (http://www.microsoft.com/germany/Express/)
  - Sharp Develop (http://sharpdevelop.net/)
  - MonoDevelop (www.mono-project.com)

Alle diese Dinge sind Freeware und du bekommst sie in der Schule.
Unser Ziel ist es, C# auf MONO Basis zu erlernen. Nur dann ist man **plattformunabhängig** und kann seine Programme für **Windows**, **Linux** und **Mac** veröffentlichen. Der einzige Unterschied zu reinem .NET ist die Grafikbibliothek GTK# die verwendet wird.

2. MONO vs .NET

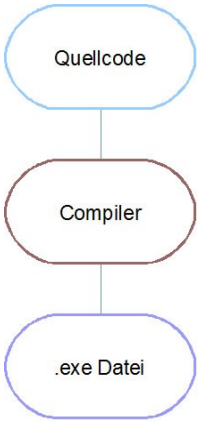
Programme die auf dem .NET Framework programmiert wurden, sollten eigentlich auf jeder Plattform (Windows, Linux, Apple, …) laufen. Doch Microsoft veröffentlicht das Framework nur für Windows. Das MONO Projekt hat sich die Aufgabe gestellt, für alle Plattformen ein Framework zu veröffentlichen, das heißt dann halt MONO. Das MONO Framework benötigt daher immer etwas Zeit, um den aktuellen Vorgaben von .NET gerecht zu werden.
Auf .NET erstellte Anwendungen können auch auf MONO laufen, nur muss man zuerst mit dem **MOMA** Tool testen, ob bereits alle Funktionen auf MONO vorhanden sind (siehe dazu Kapitel 13. MONO auf Windows, Linux und Mac)

3. Einleitung

3.1. Arbeitsweise eines herkömmlicher Compiler

Herkömmliche Compiler sind z.B: Delphi, C++.
Der Quellcode wird vom Compiler in Maschinensprache übersetzt, und erzeugt eine ausführbare Datei (meistens *.exe Dateien auf Windows BS). Das Programm kann sofort gestartet werden.

Solche Programme sind nur am aktuellen BS ausführbar!

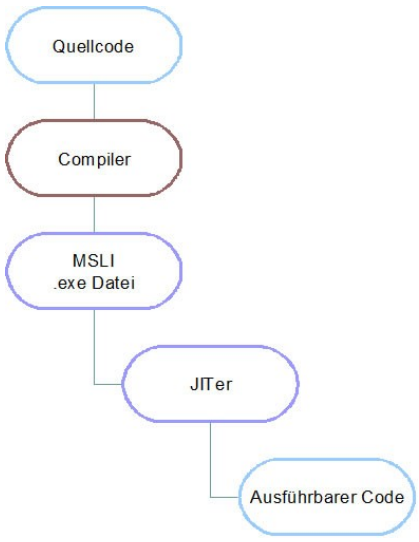


3.2. Arbeitsweise von .NET

Der Compiler erzeugt auch hier eine .exe Datei, die ist aber noch nicht lauffähig. Die **MSLI** = Microsoft Indermediate Language ist eine Zwischensprache, die zwar zum teil schon compiliert ist, aber noch nicht lauffähig ist.

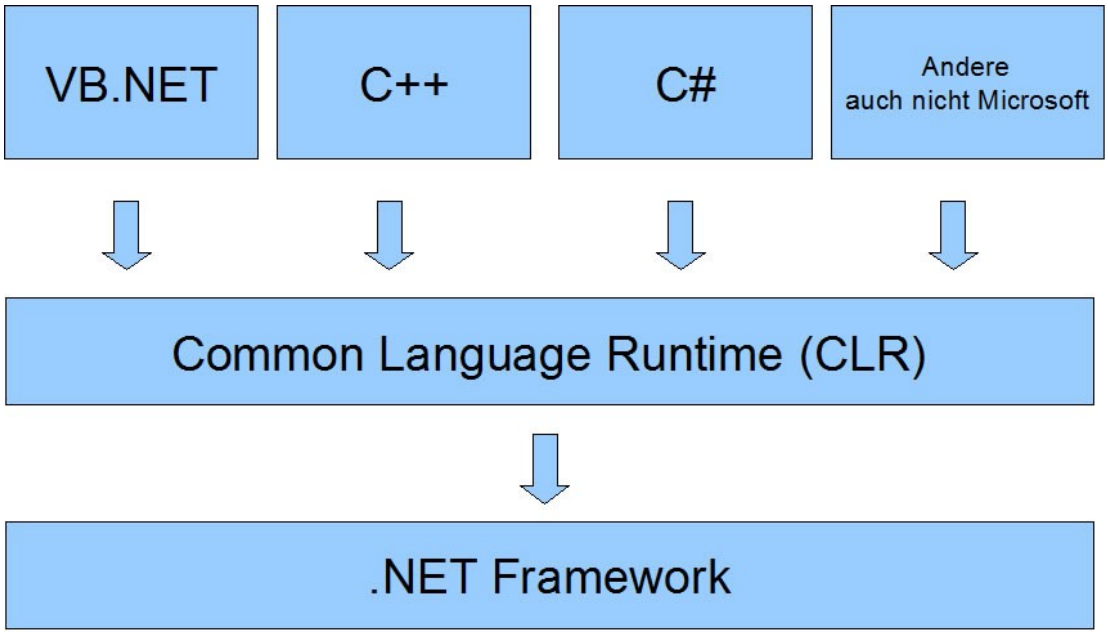
Der **JITer** wird vom Endcompiler auf dem System erst compiliert (Just-In-Time). Dann ist das Programm auf jedem BS ausführbar (ähnlich zu JAVA). Denn erst beim starten des Programms werden die Anpassungen für das BS vorgenommen.

So kann man nun Programme für Linux und Windows erstellen, es muss nur das .NET Framework[1] installiert sein.



Hier noch einmal eine grafische Veranschaulichung von der Arbeitsweise. Das .NET Framework ist übrigens auch mit einer beliebigen Sprache zu programmieren.

1 Für Linux ist es das MONO Projekt http://www.mono-project.com



4. Grundlagen

4.1. Das erste Programm

```
using System;

namespace Console1
{
    /// <summary>
    /// Zusammenfassung für Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Fügen Sie hier Code hinzu, um die Anwendung zu starten
            //

        }
    }
}
```

Zur Erklärung

using System bindet die Klasse System ein (alles zur Konsole)

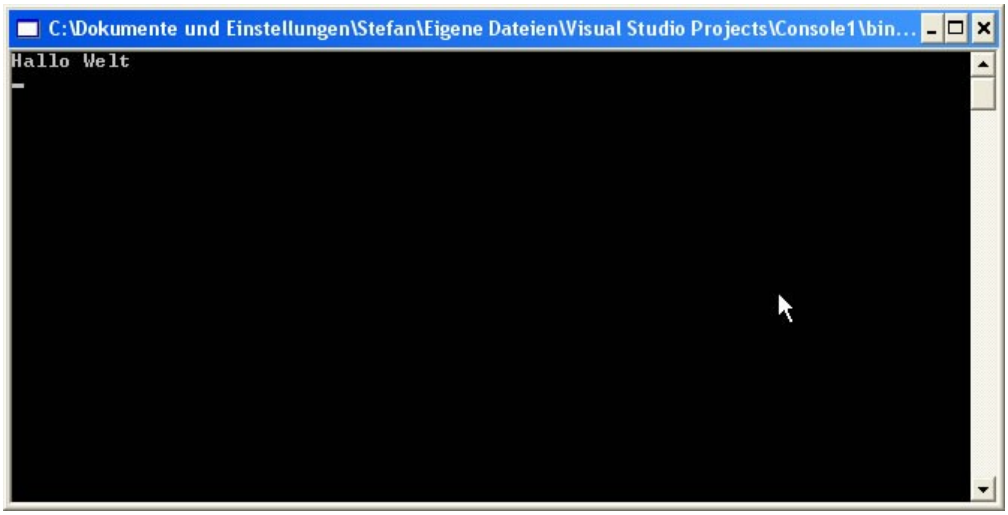
namespace beschreibt das Objekt innerhalb des .NET Frameworks.
class ist die Klasse (OOP später) die man schreibt
static void Main(string[] args)
void es wird kein Wert zurückgeliefert
Main der Startpunkt der Anwendung
string[] args Stringarray das Übergabeparameter beinhaltet

```
///       sind XML Kommentare und dienen der Dokumentation
//        sind kommentare und werden ignoriert
```

Main ist der Haupteinstiegspunkt des Programms. Dort beginnen wir zu programmieren mit

```
Console.WriteLine("Hallo Welt!");
Console.ReadLine();
```

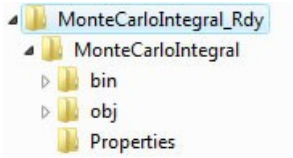
Die Ausgabe sieht dann wir folgt aus



Den Befehl `ReadLine` benötigen wir, damit das System auf eine Tasteneingabe des Benutzers wartet. Ohne diesen Befehl schließt sich sofort das Fenster, und wir sehen keine Ausgabe.

4.2. Die Struktur der Dateien

Welche Verzeichnisstruktur mit welchen Dateien erhalte ich nun? Schauen wir uns dazu die Verzeichnisstruktur genauer an.



Alle Quellcodes liegen in einem Ordner. Im Ordner **bin** findet man die compilierten ***.exe Dateien** und ***.dll Dateien**. Diese Dateien werden dann vom Framework interpretiert.

4.3. Variablentypen

Folgende Zahlentypen gibt es in C#

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Andere Typen

Type	Approximate range	Precision
float	-1.5×10^{-45} ... $+3.4 \times 10^{38}$	7 digits
double	-5.0×10^{-324} ... $+1.7 \times 10^{308}$	15-16 digits
decimal	$\pm 79 \times 10^{27}$	
char	0 to 65535	Unicode Zeichen
string	ca. 2^{31} Unicode Zeichen	
bool	true oder false	
object	Object kann alle anderen Datentypen beinhalten	Universell

Wie legt man nun Variable an, z.B. so

```
Variablentyp Name;
```

```
int x;          //Variable nicht initialisiert
int x = 2;      //initialisiert

String was = "Hallo Welt";
Char x = 'C';
float fltValue = 0.123456789;

bool myBool = true;
```

4.4. Konvertierung von Zahlentypen

Ein Standardproblem ist es, einen Zahlentyp in einen anderen über zuführen.

```
float x,y;
x = 0.123456789;
y = 0.123456789F;
```

Im obigen Beispiel wird bei der Zuweisung von x der Wertebereich des types `float` überschritten werden, da IMMER der Typ `double` angenommen wird! Damit das nicht passiert, kann man am Ende der Zahl das **Suffix F** anhängen. Damit wird der Typ `float` erzwungen.
Also Achtung bei `float` Typen!

Die Konvertierung kann **explizit** erfolgen:

```
(<Zieldatentyp>) <Ausdruck>

float fltVar = 3.12f;
decimal decVar = (decimal) fltVar;

sbyte bytVar = 20;
byte b = (byte) byteVar;
```

oder mit dem **Convert-Objekt**

```
float fltVar = 3.12f;
decimal decVar = Convert.ToDecimal(fltVar);

sbyte bytVar = 20;
byte b = Convert.ToByte(byteVar);
```

5. Strukturen

5.1. Operatoren

Math. Operatoren	Aktion
+=	i += 10 entspricht i = i + 10
-=	i -= 10 entspricht i = i – 10
+	Plus
-	Minus
*	Mal
/	Division
%	Modulo (berechnet Rest der Division)

++	i++ entspricht i=i+1
--	i-- entspricht i=i-1

Vergleichsoperatoren	
==	Gleich
!=	Ungleich
>	Größer
<	Kleiner
>=	Größer oder gleich
<=	Kleiner oder gleich
Logische Operatoren	
&&	UND (beide Ausdrücke müssen wahr sein)
||	ODER incl (mind. 1 Ausdruck muss wahr sein)
^^	ODER excl (genau nur 1 Ausdruck darf wahr sein)
bitweise Operatoren	
<<	bitweises verschieben nach links
>>	bitweises verschieben nach rechts

5.2. If - Abfragen

Dient dazu Fälle zu unterscheiden. Üblicherweise gibt es nur 2 Möglichkeiten

```
if (<Anweisung>){
   …
}
else{
  …
}
```

Wenn die **Anweisung** eintritt, wird der **if Bereich** ausgeführt. Tritt die **Anweisung** aber nicht ein, der **else Bereich.**
Hier ein Beispiel

```
if (zahl1<zahl2){
  Console.WriteLine("Die Zahl {0} ist größer als die Zahl {1}",
zahl2, zahl1);
}
else{
  Console.WriteLine("Die Zahl {0} ist größer als die Zahl {1}",
zahl1, zahl2);
}
```

Wie können Anweisungen aussehen, in etwa so

```
zahl1 == zahl2

    zahl1 ist gleich zahl2?

zahl1 != zahl2

    zahl1 verschieden zahl2?

zahl1 % zahl2 == 0

    zahl1 modulo zahl2 gleich Null?

(x ==1 ) && (y != 1)

    x gleich 1 UND y verschieden zu 1?

Ausdruck1 || Ausdruck2

    Boolscher Ausdruck1 oder Ausdruck2
```

Es ist auch möglich, eine Kurzschreibweise zu schreiben

```
if (x<10){
  y = 1;
}
else{
  y = 0;
}
```

wird zu

```
y = x<10 ?1:0;
```

5.3. Switch – Anweisung

Die **switch Anweisung** ersetzt viele verschachtelte If Anweisungen. Die Syntax sieht wie folgt aus

```
switch (<Ausdruck>)
{
    case <Konstante_1>
          /*Anweisungen*/
    break;
    case <Konstante_2>
          /*Anweisungen*/
    break;

    u.s.w
}
```

Als Beispiel wollen wir die Note eines Schülers abfragen

```
int Note = …;

switch (Note)
{
    case 1:
      Console.Write("Sehr gut");
    break;
    case 2:
      Console.Write("Gut");
    break;

    case 3:
    case 4:
    case 5:
      Console.Write("Auch nicht schlecht!");
    break;

    default:
      Console.Write("Nix gefunden");
    break;
}
```

Wie hättest du diese Abfrage mit `if-Anweisungen` erstellt? Überlege!

5.4. Schleifenarten

Schleifen sind ein wichtiges Instrument beim Programmieren. Sie haben die Aufgabe gewisse Befehle solange zu wiederholen, bis ein Ziel erreicht worden ist. Welchen Schleifentyp man verwendet ist an und für sich egal.

5.4.1. For-Schleife

Die Syntax sieht wie folgt aus

```
for (Variable; Bedingung; Weiterschalten)
{
    //Anweisungen
}
```

Solange die **Bedingung** erfüllt ist, zählt die Schleife in der **Variable** mit **Weiterschalten** weiter.

```
for (int i=0; i<=101; i++)
{
  Console.Write("Hallo das ist Satz Nr. {0} ", i);
}
```

Solange wie i kleiner oder gleich 101 ist, wird in Einserschritten `i++` weiter gezählt und die Anweisungen ausgeführt. Mit dem `break` Befehl kann man auch vorzeitig eine Schleife verlassen.

```
for (int i=0; i<=101; i++)
{
```

```
  if (i==99)
  {
    break; /*Beenden der Schleife*/
  }
  Console.Write("Hallo das ist Satz Nr. {0} ", i);
}
```

5.4.2. foreach - Schleife

Die Syntax sieht wie folgt aus

```
foreach (<Datentyp> <Bezeichner> in <Array>)
{
    /*Anweisungen*/
}
```

Als Beispiel

```
int[] intArray = {2,4,5,6}

foreach (int tempInt in intArray)
{
  Console.Write("Element: {0} ", tempInt );
}
```

Die foreach-Schleife arbeitet alle Einträge im Array (vergl. dazu Kapitel 5.6. Arrays) automatisch ab. Die Definition einer Zählvariable entfällt hier (was sehr praktisch ist). Jeder Eintrag wird in der Variable `tempInt` gespeichert, und innerhalb der Schleife greift man nur noch auf diese Variable zu.

5.4.3. while - Schleife

Die Syntax sieht wie folgt aus

```
while (<Bedingung>)
{
    //Anweisungen
}
```

Schauen wir uns dieses Beispiel an

```
int intVar = 0;

while (intVar < 5)
{
  intVar++;
  if (intVar == 3) break;
  Console.Writeln(intVar);
}
```

Wie man sehen kann, muss außerhalb der Schleife eine Zählvariable `intVar` definiert werden. Innerhalb der Schleife muss man diese Variable weiterzählen. Gerne vergisst man das, und

erzeugt damit ungewollt eine **Endlosschleife**.

5.4.4. do - Schleife

Syntax wie folgt

```
do
{
  //Anweisungen
} while (<Bedingung>)
```

Diese Schleife läuft auf jeden Fall einmal durch! Dadurch wird sie manchmal eingesetzt.

Ein Beispiel

```
int intVar = 6;

do
{
   intVar++;
   if (intVar == 3) break;
   Console.Writeln(intVar);
} while (intVar < 5)
```

5.5. Exeptions

Exeptions werden immer dann ausgelöst, wenn im Programmfluss ein Fehler auftritt. Im Regelfall stürzt das Programm mit einer Fehlermeldung ab (bekannt von Windows). Der fleißige Programmierer kann aber solche Fehler abfangen und darauf reagieren, und somit das Programm vor Abstürzen schützen.
Klingt kompliziert? Ist es nicht. Schauen wir uns den folgenden Programmcode an

```
int zahl=0;
bool err = false;

try
{
    zahl = Convert.ToInt16(eingabe_zahl);
}
catch (ArgumentException)
{
    Console.WriteLine("Bitte IRGENDWAS eingeben!");
    err = true;
}
catch (FormatException)
{
    Console.WriteLine("Bitte nur Zahlen von 0..9 eingeben!");
    err = true;
}
finally
```

```
{
    Console.WriteLine("finally wird IMMER ausgeführt");
    if (!err) Console.WriteLine(zahl);
}
```

Für jeden Fall eines Fehlers gibt es im Allgemeinen einen eigenen Exceptiontyp. Welchen Typ man abfragen kann, findet man in der Hilfe des Programms. Üblicherweise fängt man Fehler in einem **try catch** Block ab.
Passiert im **try** Block ein Fehler, geht das Programm zum entsprechenden **catch** Block über, und arbeitet dort weiter. Das Programm stürzt nicht ab!
Der **finally** Block wird auf jeden Fall ausgeführt.

Möchte man NICHT auf einen speziellen Fehlertyp reagieren, sondern auf jeden beliebigen Fehler, schriebt man einfach

```
catch (Exception e)
{

}
```

oder man lässt die Klammer komplett weg. In der Variable `e` findet man die entsprechenden Fehlermeldungen.

5.6. Arrays

Werden wie in php folgendermaßen definiert, `Typ[] Name`.

```
int[] chessboard;

Point[][] points;
```

Mit Inhalten befüllt wird ein Array so

```
int[] prim = { 1, 2, 3, 5, 7, 7 + 4, };
```

Zugriffe geschehen über eine **Indexnummer**

```
int[] zahlen = { 1, 2, 3, 5, 7, 7 + 4, };

Console.WriteLine( zahlen.length );

Console.WriteLine( zahlen[0]);

Console.WriteLine( zahlen[3]);
```

Auch mehrdimensionale Arrays sind Standard wie folgendes Beispiel zeigt

```
zahlen[3][0] = 1;

zahlen[1][3] = 4;
```

5.7. Dynamische Listen – List<>

Anders als Arrays, sind dynamische Listen flexibler zu verwenden. Sie bieten zahlreiche Methoden zur Verwaltung, und sind nicht in ihrer Größe beschränkt.

```
List<string> StringListe = new List<string>();
```

Verwendet werden Listen wie oben angeführt. Nach dem Schlüsselwort `List<>` schreibt man in den spitzen Klammern den Typ, den man speichern möchte. Bevor man die Liste verwenden kann, muss man das Objekt mit **new** anlegen!

Hier einige Beispiele was so eine Liste alles kann

```
Fav.Add("Juhhuuu C#");          /*Werte hinzufügen*/
Console.WriteLine(Fav[3]);      /*Werte laden*/
Fav.Remove("Text");             /*Werte löschen*/
Fav.Sort();                     /*sortieren*/
Fav.Clear();                    /*Alle Elemente löschen*/
```

Das Arbeiten gestaltet sich sehr elegant mit so einer Liste.

5.8. Methoden

Methoden gibt es mit und ohne Rückgabewert. Vergleiche die beiden Beispiele

```
private void Methode(int parameter1, string str)
{

}
```

```
private string Methode2(int parameter1, string str)
{
  return "Yes, ist C#"
}
```

Das Schlüsselwort **void** bezeichnet, dass die Methode keinen Rückgabewert hat. Liefert die Methode etwas zurück, dann steht vor dem Methodennamen der Variablentyp. Hier ist das `string`.

5.9. Paramterübergabe

In den obigen Beispielen werden auch **Parameter** an die Methoden übergeben. Diese Parameter stehen nur innerhalb der Methode zur Verfügung, und werden beim Verlassen der Methode wieder gelöscht. Gibt es keine Parameter, bleibt die Klammer leer.

```
private void Methode()
private void Methode(int parameter1, string str)
```

Globale Parameter sind solche, die im gesamten Programm gültig sind. Solche Parameter werden aber außerhalb, also im Hauptprogramm oder in einer Klasse, definiert.

5.10. Parameterübergabe mit ref und out

Auch ohne eine Methode mit **Rückgabewert** kann man Variablen zurückgeben lassen. Dazu muss man in der übergebenen Parameterliste noch ein Schlüsselwort einfügen, nämlich

- ref
- out

Beide Typen funktionieren gleich. Bei `ref` **MUSS** allerdings der übergebene Parameter bereits initialisiert sein, d.h. er muss existieren!

Die Parameter die mit `ref` oder `out` markiert werden, dienen als Rückgabeparameter.
Bei `ref` muss der Parameter global existieren. Jede Veränderung innerhalb der Methode, verändert den globalen Parameter = selbe Funktion wie beim `return`-Befehl!

Hier ein **Beispiel**:

```
static void GetFlaeche2(ref double flaeche, double l,double b){
    flaeche = l*b;
}

static void Main(string[] args)
{
    double flaeche=0;
    Ausgabe("Hallo das ist eine Procedure");
    GetFlaeche2(ref flaeche, 3.12, 4.234);

    /* flaeche ist nun global verändert worden */
    Console.WriteLine("Fläche {0}", flaeche);
}
```

Die `flaeche` wird demnach ohne einem `return` Statement an das Hauptprogramm übergeben!

5.11. Dateien, Streams und Filestreams

Der zugehörige **Namespace** um IO-Operation (Input-Output) durchzuführen, lautet `System.IO`. Darin findet man alle Methoden um mit IO Ausgaben zu programmieren.

Das Objekt File beinhaltet alles, was man benötigt, um mit Dateien zu arbeiten. Ein Beispiel

```
If (File.Exists("Test.txt"")) {}
```

Hier wird die Datei Test.txt im Pfad der ***.exe** Datei gesucht. Dieser Pfad kann so abgefragt werden

```
string verz = Environment.CurrentDirectory;

string SysDir =
```

```
Path.GetDirectoryName(Application.ExecutablePath) + "\\";
```

Achtung: einfache \ werden als EscapeZeichen behandelt!
Ein Backslash muss durch \\ dargestellt werden!

Noch ein Beispiel

```
/*Kopieren einer Datei*/
File.Copy("C:\\Test.txt", "C:\\Neu.txt" );
```

Um aus einer Datei zu lesen, oder in eine Datei zu schreiben, verwendet man Streams. Streams sind Datenströme beliebiger Art.

```
StreamWriter sw = File.CreateText("Test.txt");
sw.Write();
sw.WriteLine();
sw.Close();

StreamReader sw = new StreamReader("Test.txt");
sw.Read();
sw.ReadLine();
sw.Close();
```

Im obigen Beispiel sieht man die prinzipielle Anwendung eines Streams mit einer Datei.

Hier noch ein Beispiel, wie man an eine bestehende Datei etwas anhängen kann.

```
StreamWriter sw = File.AppendText("Logfile.txt");
sw.WriteLine("Huhu");
sw.Close();
```

5.12. Events - Ereignisse

Ereignisse werden immer dann ausgelöst, wenn eine bestimmte **Aktion** eintritt, z.B.

- Benutzer klickt Button an
- Maus wird bewegt
- Exception wird ausgelöst
- Fenster wird minimiert
- etc.

Auf solche Ereignisse muss man als Programmierer reagieren, nur wie?

```
/* Anlegen von zwei Buttons zur Laufzeit */
Button btn1 = new Button();
Button btn2 = new Button();

/* Zum Fenster dazugeben */
```

```
this.Controls.Add(btn1);
Controls.Add(btn2);

btn2.Location = new Point(0, 200);

/* die Events der Buttons mit Methoden belegen */
btn1.Click += new EventHandler(btn1_Click);
btn2.Click += new EventHandler(btn2_Click);
```

Die EventHandler sind dabei **Delegaten**, die auf einen Methodenkopf verweisen. Der Delegat sieht so aus

```
public delegate void EventHandler(object sender, EventArgs e);
```

In den **Designer Dateien** kannst du mitlesen, wie die GUI solche Events automatisch erstellt.

Die Methode selbst muss dann den Methodenkopf des **Delegaten** haben, und könnte so aussehen

```
private void btn1_Click(object sender, EventArgs e)
{
    /* Quellcode was passieren soll */
}
```

5.13. Controls zur Laufzeit anlegen

```
for (int i = 0; i < 16; i++)
        {
            //Zeile weiterschalten
            if ((i % anordnung) == 0) { zeile++; }

            Button btn = new Button();
            btn.Width = breite;
            btn.Height = breite;
            btn.Text = ".";
            btn.Location = new Point((i % anordnung) * breite, zeile * breite);

            this.Controls.Add(btn);
        }
    }
```

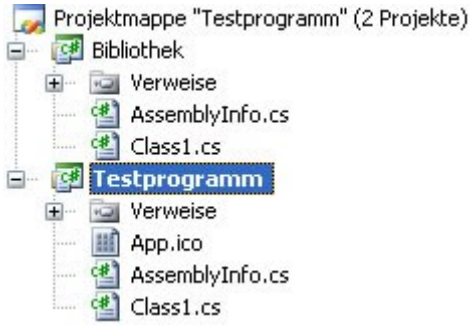
6. Klassenbibliotheken

Oft programmiert man Routinen, die man wiederverwenden möchte. Stell dir Methoden vor, die auf Knopfdruck z.B. die Fläche von Rechtecken, Parallelogrammen oder Trapezen berechnet. Warum

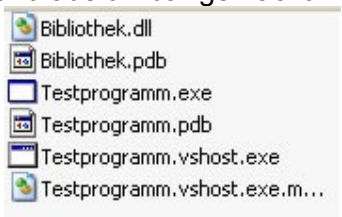
soll man diese Methoden jedes mal neu programmieren wenn man sie benötigt. Hier kommt die **Klassenbibliothek** ins Spiel. Eine **Klassenbibliothek** ist eine Sammlung von Routinen, die man jederzeit wiederverwenden kann.
Das Erstellen ist ganz einfach. Wichtig zu wissen ist, das jede **Klassenbibliothek** ein eigene `*.dll` Datei erzeugt (=dynamic link libary).

6.1. Erstellen

Das Erstellen ist recht einfach. Eine Klassenbibliothek ist ein eingestehendes Projekt in Visual Studio, und wird einfach in den Projektmappenexplorer integriert. Das hat den Vorteil, dass man weiterhin am Quelltext arbeiten kann, und gleichzeitig die Bibliothek verwenden kann.



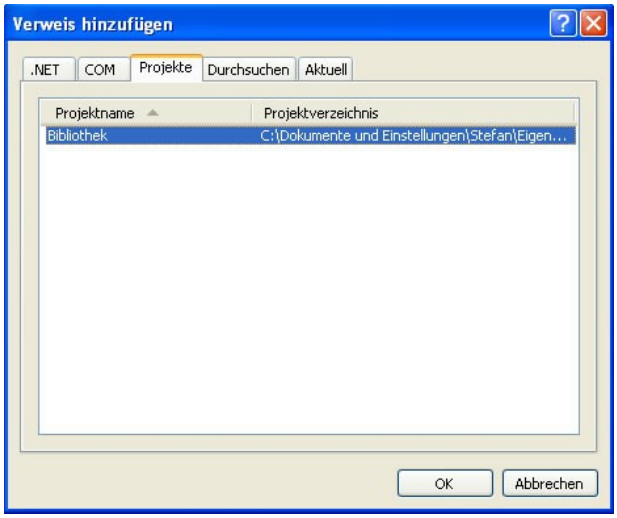
Nebenstehend siehst du einen Ausschnitt aus einem Projektmappenexplorer. Sowohl Bibliothek als auch das aktuelle Programm sind ersichtlich. Mit der Taste F6 werden beide kompiliert und ausführbar gemacht.



So sieht dann das Ergebnis aus.

Eine Klassenbibliothek wird ganz normal programmiert und benötigt keine besonderen Kennzeichen. Im Hauptprogramm muss man aber einige Sachen berücksichtigen.

1. Ein Verweis ist auf die **.dll* zu setzen, durch Rechtsklick auf *Verweise >> Verweis hinzufügen*.



   Damit wird die erzeugte *.dll automatisch in das bin Verzeichnis des Hauptprogramms kopiert.

2. Wenn z.B. die Klassenbibliothek so beginnt, dann ist die Kenntnis des `namespace` wichtig.

```
namespace Bibliothek
{
    /// <summary>
    /// Klasse Kreis
    /// </summary>
    public class Kreis
    {}
}
```

3. Im Hauptprogramm wird dieser Namespace mit diesem Befehl eingebunden.

```
using System;

using Bibliothek;
```

Ab nun kann man auf die Klassenbibliothek ganz normal zugreifen und damit arbeiten.

7. Zeichnen mit der GDI+

GDI ist die Zeichenschnittstelle von C# auf .NET. Bei MONO wäre das GTK.
Zeichnen sollte man immer dort, wo das Programm sich von selbst neu zeichnet, damit es zu keinen Grafikbugs kommt. Das Fenster zeichnet sich immer dann neu, wenn es verschoben, minimiert, maximiert etc. wird. Hierbei wird immer die `Paint()` Methode des Fensters aufgerufen, und genau das ist der Ort wo wir eingreifen werden!

Schauen wir uns einmal die Paint() Methode des Fensters selbst an

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
}
```

Diese Methode wird immer aufgerufen, wenn sich das Fenster neu zeichnen muss. Dazu muss man gleich sagen, dass man in allen Objekten zeichnen kann, man muss nur definieren wo gezeichnet wird!

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
  //hole dir das Zeichenobjekt von this = Fenster
  Graphics fx = this.CreateGraphics();
  fx.SmoothingMode = SmoothingMode.HighQuality;
  //Koordinatenursprung in den Mittelpunkt des Fensters
verschieben
  fx.TranslateTransform(
          this.Width / 2,
          this.Height /2

  );
```

```
    //y-achse wird nach oben positiv gezählt
    fx.ScaleTransform(1f, -1f);
    //blauer Kreis
    Pen myPen = new Pen(Color.Blue, 1.2f);
    Rectangle rect = new Rectangle(10,10, 200, 200);
    Brush myBrush = Brushes.Coral;
    fx.FillEllipse(myBrush, rect);
    fx.DrawEllipse(myPen, rect);
}
```

Paint Methode
creategraphics
Zeichnen Quadrat, Kreis
Farben

```
//hole dir das Zeichenobjekt von this = Fenster
        Graphics fx = this.CreateGraphics();
        fx.SmoothingMode = SmoothingMode.HighQuality;
        //Koordinatenursprung in den Mittelpunkt des Fensters verschieben
        fx.TranslateTransform(
           this.Width / 2,
           this.Height /2
        );
        //y-achse wird nach oben positiv gezählt
        fx.ScaleTransform(1f, -1f);




        //blauer Kreis
        Pen myPen = new Pen(Color.Blue, 1.2f);

        Rectangle rect = new Rectangle(10,10, 200, 200);

        Brush myBrush = Brushes.Coral;
        fx.FillEllipse(myBrush, rect);
        fx.DrawEllipse(myPen, rect);
```

Vektorgrafik

, Update, repaint
zeichenproggi

8. Dialoge

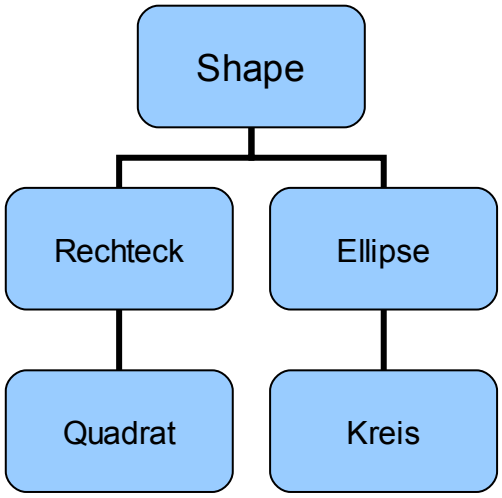
Öffnen, Speichern

9. OOP – Objektorientiertes Programmieren

Alle modernen Programmiersprachen setzen auf Objekte beim Programmieren. Die Programmierung wird dadurch viel einfacher in der Handhabung. Anstatt eines linearen Codes hat man Objekte zur Verfügung, die gewisse Aufgaben erledigen können. Diese Objekte erstellt man, wenn man sie benötigt und teilt ihnen Aufgaben zu. Im Allgemeinen ist OOP flexibler, leistungsfähiger und bei größeren Programmen einfach übersichtlicher.
Die Übersichtlichkeit kann man mit diversen Zugriffsmodifizierer (vergl. 9.3. Zugriffsmodifizierer auf Seite 24) steuern.

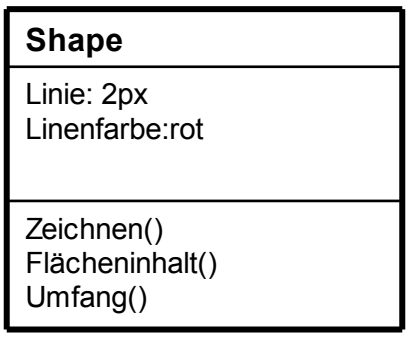
9.1. Grundlagen

Sehen wir uns ein Beispiel aus der Mathematik an



- Alle angeführten Objekte sind abgeleitet von der Basisklasse **Shape**.
- Die Eigenschaften der Basisklasse werden an die nachfolgenden Objekte vererbt
- Jedes Objekt hat Eigenschaften und Methoden
- Methoden der Basisklasse könne überschrieben werden

Warum leitet sich z.B. ein Quadrat von einem Rechteck ab? Die Antwort kennst du aus der Mathematik. Jedes Quadrat ist auch ein Rechteck, das Quadrat ist nur eine spezielle Form des Rechtecks. Wenn also die Fläche eines Rechtecks mit Seite x Seite zu berechnen ist, dann ist es auch beim Quadrat so.
Shape ist das allgemeinste Objekt. Es hat allgemeine Eigenschaften eines Körpers in 2D, also eine Begrenzungslinie, eine Fläche und einen Umfang etc.
Die Definition könnte so aussehen



Objektnamen

Eigenschaften

Methoden

Von dieser Klasse leiten wir nun das Rechteck ab.

Länge und **Breite** sind eine spezielle Eigenschaft von eine Rechteck, während *Linie*, *Linienfarbe* bereits von **Shape** geerbt worden sind,

Die Methode **Diagonale()** ist wieder eine spezielle Eigenschaft, der Rest wurde geerbt.

Der dazu passende Quellcode sieht so aus:

```
public abstract class Shape
{
   public int Linie;
   public int Liniendicke;

   public abstract double Umfang();
   public double Flaeche()
   {
     return 0;
   }
}
```

Das Objekt / Klasse wird hier zusätzlich **abstract** definiert. Abstrakte Klassen stellen nur einen Rumpf dar. Die abgeleiteten Klassen müssen dann genauer definieren was passieren soll. Die Methode `Umfang()` ist abstrakt, es muss nur der Kopf definiert werden. Die Methode `Flaeche()` hingegen ist nicht abstrakt, und muss bereits hier aus programmiert werden.

Das abgeleitete Rechteck sieht nun so aus:

```
public class Rechteck : Shape
{
   public double laenge = 0, breite = 0;
```

```
    public override double Umfang()
    {
        return 2 * (laenge + breite);
    }

    public new double Flaeche()
    {
        return laenge * breite;
    }
}
```

Eine Klasse wird von einer anderen Klasse abgeleitet mit der Schreibweise `Neue Klasse : Mutterklasse`.

Abstrakte Methoden müssen mit `override` überschreiben werden. Die Methode Flaeche() muss mit `new` neu definiert werden. Ansonsten weiß man nicht ob die Methode `Flaeche()` von der Klasse **Shape** oder von der Klasse **Rechteck** kommt!

Rechteck definiert neu `laenge` und `breite`, hat aber zusätzlich noch die Eigenschaften `Linie` und `Linienfarbe` der Klasse Shape (vererbt).

Überlege selbst, wir die Klasse Quadrat aussehen muss.

9.2. Konstruktoren

Konstruktoren sind Methoden, die beim Erstellen des Objektes ausgeführt werden. Sie heißen daher immer so, wie das Objekt heißt.

```
public partial class CustomRichtextBox : UserControl
{
    /* Konstruktor */
    public CustomRichtextBox()
    {
        this.SetStyle(ControlStyles.DoubleBuffer |
            ControlStyles.UserPaint |
            ControlStyles.AllPaintingInWmPaint |
            ControlStyles.SupportsTransparentBackColor |
            ControlStyles.ResizeRedraw, true);
        this.BackColor = Color.Transparent;

    }
}
```

9.3. Zugriffsmodifizierer

private, public

9.4. Getter und Setter Methoden

9.5. Überladungen von Methoden

10. Threading in C#

10.1. Erstellen einer Producer Queue

Stellt eine Liste von Threads dar, die Schritt für Schritt abgearbeitet wird. Jeder Thread sperrt dabei die Methode des Threads, somit ist diese Liste auch Threadsicher.

A simple Wait/Pulse application is a producer-consumer queue – the structure we wrote earlier using an AutoResetEvent. A producer enqueues tasks (typically on the main thread), while one or more consumers running on worker threads pick off and execute the tasks one by one.
In this example, we'll use a string to represent a task. Our task queue then looks like this:

```
Queue<string> taskQ = new Queue<string>();
```

Because the queue will be used on multiple threads, we must wrap all statements that read or write to the queue in a lock. Here's how we enqueue a task:

```
lock (locker) {
  taskQ.Enqueue ("my task");
  Monitor.PulseAll (locker);   // We're altering a blocking condition
}
```

Because we're modifying a potential blocking condition, we must pulse. We call PulseAll rather than Pulse because we're going to allow for multiple consumers. More than one thread may be waiting.
We want the workers to block while there's nothing to do, in other words, when there are no items on the queue. Hence our blocking condition is taskQ.Count==0. Here's a Wait statement that performs exactly this:

```
lock (locker)
  while (taskQ.Count == 0) Monitor.Wait (locker);
```

The next step is for the worker to dequeue the task and execute it:

```
lock (locker)
  while (taskQ.Count == 0) Monitor.Wait (locker);

string task;
lock (locker)
  task = taskQ.Dequeue();
```

This logic, however, is not thread-safe: we've basing a decision to dequeue upon stale

information – obtained in a prior lock statement. Consider what would happen if we started two consumer threads concurrently, with a single item already on the queue. It's possible that neither thread would enter the while loop to block – both seeing a single item on the queue. They'd both then attempt to dequeue the same item, throwing an exception in the second instance! To fix this, we simply hold the lock a bit longer – until we've finished interacting with the queue:

```
string task;
lock (locker) {
  while (taskQ.Count == 0) Monitor.Wait (locker);
  task = taskQ.Dequeue();
}
```

(We don't need to call Pulse after dequeuing, as no consumer can ever unblock by there being fewer items on the queue).
Once the task is dequeued, there's no further requirement to keep the lock. Releasing it at this point allows the consumer to perform a possibly time-consuming task without unnecessary blocking other threads.
Here's the complete program. As with the AutoResetEvent version, we enqueue a null task to signal a consumer to exit (after finishing any outstanding tasks). Because we're supporting multiple consumers, we must enqueue one null task per consumer to completely shut down the queue:

11. Externe Programme starten

12. Socket Programmierung

http://www.developerfusion.com/article/3918/socket-programming-in-c-part-1/3/
http://www.codeplanet.eu/tutorials/csharp/6-csharp/4-tcp-ip-socket-programmierung-in-csharp.html

Mit Sockets ist es möglich Verbindungen zu einem Netzwerk herzustellen.

> ***Definition:*** *Ein Socket ist ein Endpunkt einer bi-direktionalen Software-Schnittstelle zur Interprozess- (IPC) oder Netzwerk-Kommunikation zwischen zwei Programmen. Ein Socket ist gebunden an eine Port-Nummer, so dass die TCP Schicht die Anwendung identifizieren kann für die die Informationen bestimmt sind.*

Wir werden Tcp oder UDP Verbindungen nutzen. Um Sockets nutzen zu können, muss man folgendes einbinden

```
using System.Net;
using System.Net.Sockets;
```

Ein Socket wird wie folgt angelegt

```
Socket sock = new Socket(AddressFamily.InterNetwork,
                         SocketType.Stream,
                         ProtocolType.Tcp);
```

AddressFamily legt dabei fest ob man mit IP4 oder IP6 arbeitet. Nachdem nun der Socket steht, definieren wir einen **Endpunkt**, das ist die Gegenstelle der Verbindung.

```
IPEndPoint ipep =
new IPEndPoint(IPAddress.Parse("192.168.1.1"), 1200);
```

Man muss halt die IP der Gegenstelle kennen, und zusätzlich muss man den Port angeben, über den die Verbindung laufen soll.
Da wr auch Antworten erhalten wollen, müssen wir unserem Socket noch mitteilen an welchen Netzwerkschnittstellen er lauschen soll.

```
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
EndPoint remoteEP = (EndPoint)sender;
bcSocket.Bind(sender);
```

Die erste Zeile teilt mit, das auf allen Schnittstellen auf allen Ports gelauscht werden soll. Mit `Bind()` wird der Socket auf diesen **Endpunkten** gebunden.

```
bcSocket.SendTo(data, data.Length, SocketFlags.None, ipep);
rbytes = bcSocket.ReceiveFrom(rcdata, ref remoteEP);
```

Mit diesem Code werden nun Daten gesendet, und empfangen. Die `ReceiveFrom()` Methode liefert dabei noch den Endpunkt zurück, d.h. hier erfährt man wer gesendet hat.
Zum Schluss schließen wir noch die Verbindung

```
bcSocket.Shutdown(SocketShutdown.Both);
bcSocket.Close();
```

13. MONO auf Windows, Linux und Mac

Mit MONO ist es möglich, C# Anwendungen auf jeder Plattform (Windows, Linux, Apple) laufen zu lassen. Was benötigt man?

- Die **MONO Runtime**
- Eine Entwicklungsumgebung, z.B. **MonoDevelop**
- **GTK#**, um GTK Anwendungen zu entwickeln. Das sind Fensteranwendungen, wobei GTK die Grafikbibliothek für MONO ist. Wer daheim GTK auf .NET programmieren möchte, braucht noch **GTK# for .NET**.

Das alles kann man gratis von www.mono-project.com runterladen.

MonoDevelop steckt noch in der Entwicklung, weshalb man sehr genau programmieren muss.

13.1. Allgemeine Tipps

13.1.1. Arbeiten auf Windows

Über die *Bearbeiten → Einstellungen → .NET Runtimes* die Runtime auf MONO einstellen

13.1.2. Arbeiten auf Linux

Bei Konsolenanwendungen für jedes Projekt über *Projekt → Einstellungen → ...?* das Ausführen auf einer externen Konsole einstellen.

13.2. Für den Umstieg gut zu wissen – FAQ's

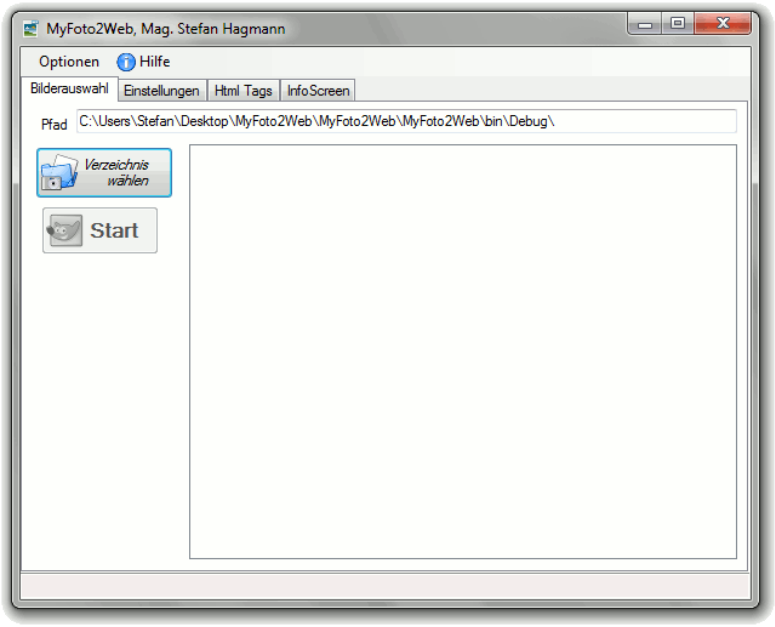
Hier sind einige Punkte gesammelt, die den Umstieg von Visual Studio zu Mono Develop erleichtern sollen.

1. Alle **Namespaces**, die per `using` Klausel eingebunden werden, müssen über die Verweise eingebunden werden.
2. Die Grafik-Engine ist GTK#, und ist anders zu bedienen als die .NET – Engine.

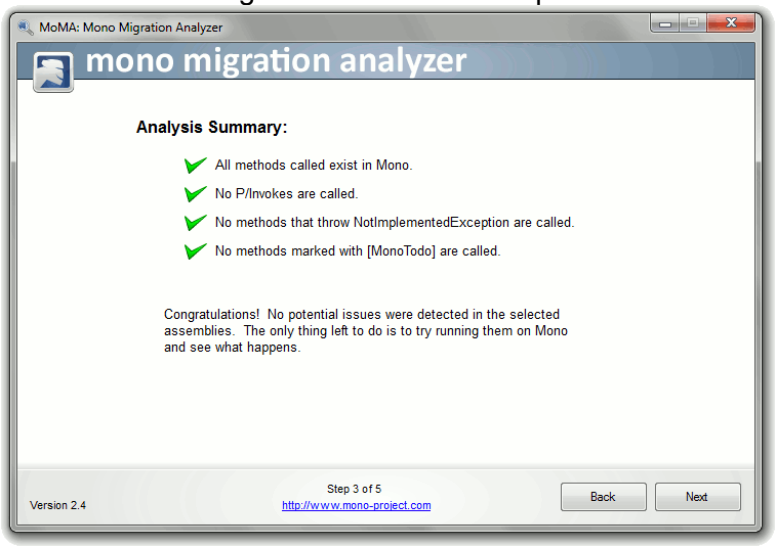
13.3. Beispiel: .NET zu MONO

Hat man eine Anwendung auf .NET entwickelt, läuft sie auch auf MONO. Allerdings braucht das MONO Projekt immer etwas Zeit, um auf die neuesten .NET Versionen zu reagieren. Mit dem **MoMA Tool**, dem *Mono Migration Analyzer*, kann man seine bestehende Anwendung auf Kompatibilität testen.
Ich zeige das hier mit einen Programm, das ein html-Fotoalbum erstellt:

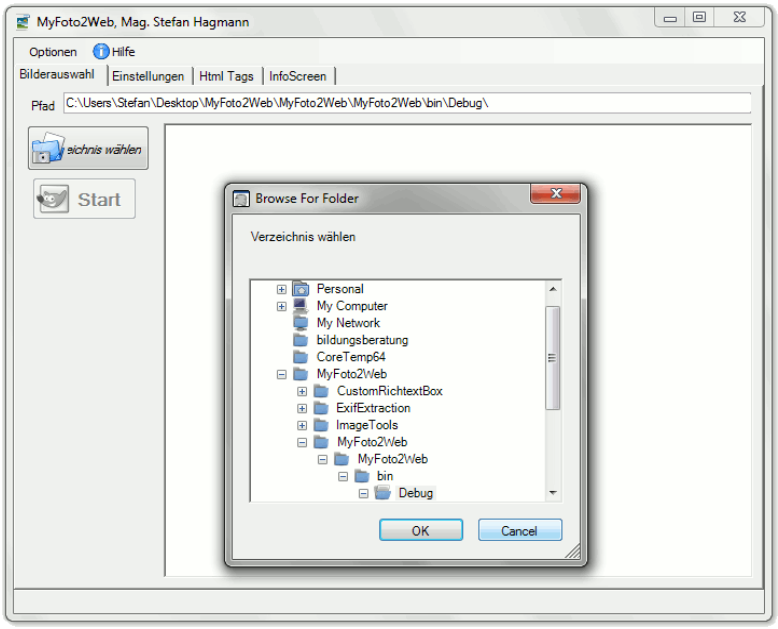
Schritt1: Starte das Programm mit einem Doppelklick



Schritt2: Teste mit MoMA ob das Programm für MONO kompatibel ist.



Schritt3: Führe das Programm mit MONO aus
Zum Starten führe den Befehl *mono Program.exe* aus. Hier kannst du dir eine Verknüpfung anlegen.



MoMA meldet dir die Codeteile die noch nicht in MONO integriert sind. Damit kannst du für beide Runtimes, .NET und MONO, kompatible Versionen erstellen.

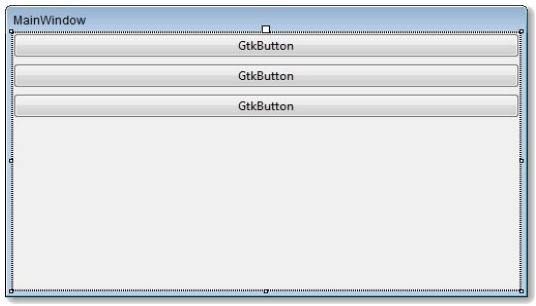
13.4. Für MONO und .NET entwickeln

Konsolenanwendungen laufen ohne Probleme, keine weitere Erklärung nötig.
Bei Fensteranwendungen muss man beachten, das die Grafik-Engine für MONO GTK# ist, das im .NET nicht enthalten ist. Wenn man also universell für beide Plattformen ein Programm entwickeln möchte, sollte man ausschließlich auf GTK# Basis arbeiten. Für .NET benötigt man dazu noch eine Umgebung, nämlich die **GTK# for .NET**.

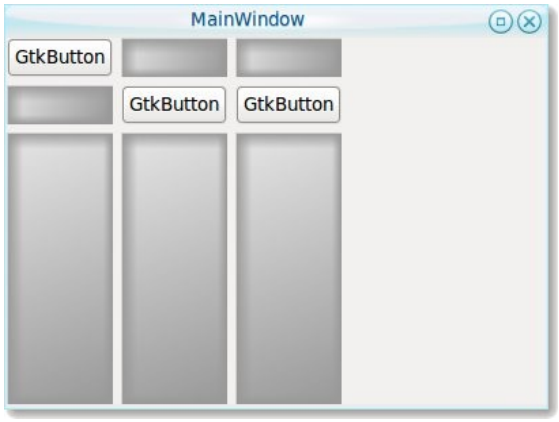
13.5. Grafische Oberfläche

Schaue z.B auch auf http://monodevelop.com/Documentation/Stetic_GUI_Designer.
Ähnlich wie bei JAVA verwendet GTK# diverse Container, genauer Widget-Container, für die grafische Oberfläche. Ohne diese Container kannst du keine Elemente darstellen lassen. Deshalb ist es sehr empfehlenswert, sich genau den Aufbau des Fensters zu überlegen. Es gibt

Boxes
die gibt es vertikal und horizontal, im Screenshot siehst du vertikale Boxes.
Am Beginn des Designs sind die Boxen relativ groß, sie passen sich dann immer an den Inhalt an!



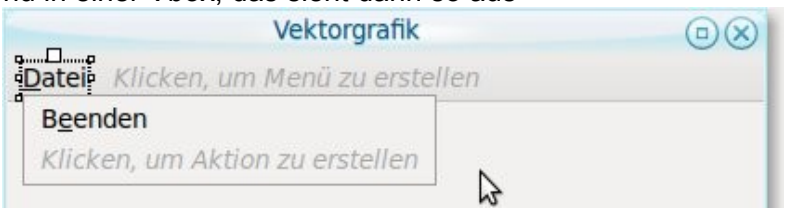
Tables
haben eine gewisse Anzahl an Spalten und Zeilen, vergleiche den Screenshot



Fixed Container
damit kann man alle Elemente beliebig auf dem Container plazieren, es gibt hier keine automatische Ausrichtung der Elemente.

13.6. Einige Tipps für grafische Oberflächen

Nachdem die Container angelegt worden sind, kann man sie mit Leben erfüllen. Zum Beispiel erstellen wir ein Menü in einer Vbox, das sieht dann so aus



Den Unterstrich der Menüeintragungen erhält man wenn man einfach einen Unterstrich schreibt, also anstatt von **Datei** schreibt man **_Datei**. Was mit den Menüpunkten passieren soll, erstellt man mit den Ereignissen (auch bekannt unter Actions, oder so wie hier Signale). Die Signale findet man dabei im Eigenschaftenfenster des entsprechenden Elementes.

Um nun eine Methode für das Element zu erstellen, macht man einen Doppelklick auf das entsrechende Action Feld, also z.B. auf das Feld **<Activated>**. Damit wird im Quellcode der entsprechende Code angelegt.

Und das sieht dann so aus

```
protected virtual void OnBeendenActionActivated (object sender,
System.EventArgs e)
{
}
```

13.7. Zeichnen mit GTK

Das zeichnen von Objekten mit GTK ist etwas anders als bei .NET. Die typische `OnPaint` Methode findet man hier leider (muss ich sagen) nicht. Im folgenden Beispiel zeichnen wir in eine abgeleitete Klasse einer `DrawingArea`, also eines Zeichenbereiches.

```
using Gtk;
using System;
using System.Drawing;

public class MyDrawing : DrawingArea {
    public MyDrawing (){
          SetSizeRequest (200, 200);
    }

    protected override bool OnExposeEvent (Gdk.EventExpose args)
    {
          using (Graphics fx = Gtk.DotNet.Graphics.FromDrawable
                (args.Window)){
          Pen p = new Pen (Color.Blue, 1.0f);
          for (int i = 0; i < 600; i += 60)
                for (int j = 0; j < 600; j += 60)
                      fx.DrawLine (p, i, 0, 0, j);
          }
          return true;
    }
}
```

Also wie man sieht verwendet man anstatt eines `OnPaint` Events ein `OnExposeEvent`. Interessant ist auch, das hier die normale `System.Drawing` zum Einsatz kommt, und der Graphics-Kontext mit
`Graphics fx = Gtk.DotNet.Graphics.FromDrawable (args.Window)` geholt wird. Ansonsten ist die Handhabung so wie bei .NET auch.