|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Technische Berichtsnr.: **3881**  Projektname: **Wifi Komplettausbildung**  Bearbeitet: Markus-Kragl  Geprüft: -  Datum: 2018\_12\_14\_ | | | | | | Lunzerstrasse 78  Postfach 28  4031 Linz, Austria  Phone: +43 (732) 6986-0 | | |
| **WifiKurs-3881-Software Developer in C# - Komplettausbildung** | | | | | | | | |
| **Zusammenfassung**  Dieser Bericht beschreibt:   * Einleitung * Anlagendaten * Materialdaten * Statische Finite-Elemente-Analyse * Modale Finite-Elemente-Analyse * Dynamische (HRA) Finite-Elemente-Analyse * Betriebsfestigkeit * Schlussfolgerung | | | | | | | | |
| Revision | .Bearbeitet | Datum | Geprüft | Datum | .Rev. folgender Seiten | | | |
| .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | | | |
|  | | | | | | | Anzahl der Seiten  28 Pages |  |
| Seiten Nr.: | Seite 1 von 24 |

**Contents**

[1 .Net-Framework 3](#_Toc5612508)

[1.1 Symbole 3](#_Toc5612509)

[1.2 Aufbau 5](#_Toc5612510)

[2 C#-Referenz 6](#_Toc5612511)

[2.1 Schlüsselwörter 6](#_Toc5612512)

[2.2 Kontextabhängige Schlüsselwörter 6](#_Toc5612513)

[2.3 Typen 6](#_Toc5612514)

[2.4 Zugriffsmodifizierer 6](#_Toc5612515)

[2.5 Typen 9](#_Toc5612516)

[3 C#-Programmstruktur 12](#_Toc5612517)

[3.1 Assemblies 12](#_Toc5612518)

[3.2 Namespaces 12](#_Toc5612519)

[3.3 Klassen 13](#_Toc5612520)

[3.4 Konstruktor und Destruktor 15](#_Toc5612521)

[3.5 Methode 17](#_Toc5612522)

[3.6 Eigenschaft 18](#_Toc5612523)

[3.7 Ereignis 19](#_Toc5612524)

[3.8 Schnittstellen 20](#_Toc5612525)

[3.9 Delegate 21](#_Toc5612526)

[4 Mehrschichtenmodel 22](#_Toc5612527)

[4.1 Model View Controller 22](#_Toc5612528)

[4.2 Firmenframework 22](#_Toc5612529)

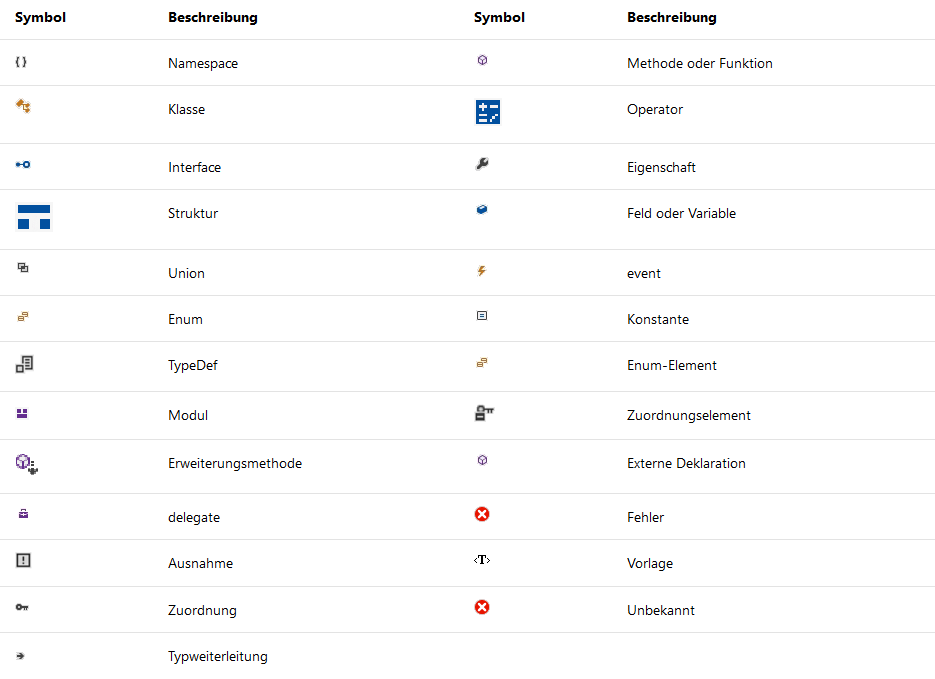
[5 Generika 24](#_Toc5612530)

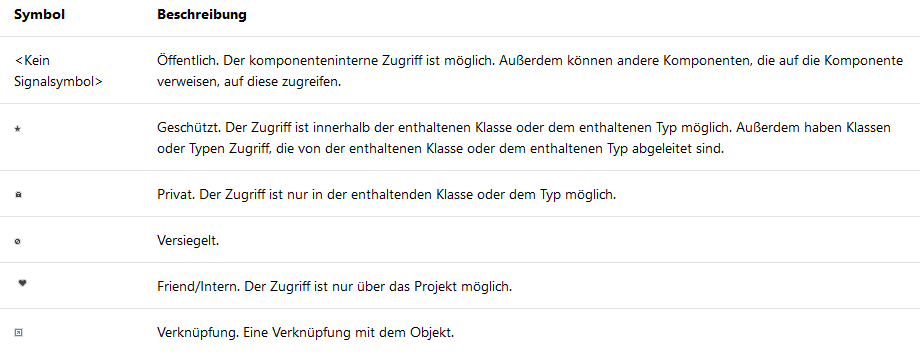
# .Net-Framework

Viele Klassen sind bereits im .Net-Framework beschrieben, damit muss nicht alles von neue beschrieben werden

<https://docs.microsoft.com/de-at/visualstudio/ide/class-view-and-object-browser-icons?view=vs-2017>

## Symbole





## Aufbau

⏹-⏹Assembly

{} Namespace

Symbol "Klasse" Klasse

Symbol "Deklaration" Methode

Symbol "Eigenschaften" Eigenschaft

Symbol "Ereignis" Ereignis

Schnittstellensymbol "Lollipop" Interfaces

Symbol "Struktur" Struktur

# C#-Referenz

## Schlüsselwörter

<https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/>

delegate

as

cachen

this

Multithreadings

Using!!!

Arrays

List

Dispatcher

Async

## Kontextabhängige Schlüsselwörter

valu

## Typen

|  |  |
| --- | --- |
| **Deklarierter Zugriff** | **Bedeutung** |
| [void](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/void) | Aus der Methode wird kein Wert zurückgegeben |
| [var](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/var) | Ab Visual Studio C# 3.0 können Variablen, die im Methodenbereich deklariert wurden, den impliziten „Typ“ var haben. Eine implizit typisierte lokale Variable ist stark typisiert, als hätten Sie den Typ selbst deklariert. Tatsächlich legt der Compiler den Typ fest. |

## Zugriffsmodifizierer

Modifizierer werden zum Ändern von Typ- und Typmemberdeklarationen verwendet. In diesem Abschnitt werden die C#-Modifizierer beschrieben.

<https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/modifiers>

|  |  |
| --- | --- |
| **Modifizierer** | **Zweck** |
| [Zugriffsmodifizierer](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/access-modifiers) -[public](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/public) -[private](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/private) -[internal](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/internal) - [protected](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/protected) | Gibt den deklarierten Zugriff von Typen und Typmembern an. |
| [abstract](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/abstract) | Zeigt an, dass eine Klasse lediglich als Basisklasse anderer Klassen verwendet werden soll. |
| [async](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/async) | Zeigt an, dass die geänderte Methode, der Lambdaausdruck oder die anonyme Methode asynchron ist. |
| [const](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/const) | Gibt an, dass der Wert des Felds oder der lokalen Variablen nicht geändert werden kann. |
| [event](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/event) | Deklariert ein Ereignis. |
| [extern](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/extern) | Zeigt an, dass die Methode extern implementiert wurde. |
| [new](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/new) | Blendet einen von einer Basisklasse geerbten Member explizit aus. |
| [override](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/override) | Stellt eine neue Implementierung eines virtuellen Members bereit, der von einer Basisklasse geerbt wurde. |
| [partial](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/partial-type) | Definiert Teilklassen, Strukturen und Methoden innerhalb derselben Assembly. |
| [readonly](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/readonly) | Deklariert ein Feld, dem Werte nur als Teil einer Deklaration oder in einem Konstruktor derselben Klasse zugewiesen werden können. |
| [sealed](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/sealed) | Gibt an, dass eine Klasse nicht geerbt werden kann. |
| [static](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/static) | Deklariert einen Member, der zum Typ selbst und nicht zu einem bestimmten Objekt gehört. |
| [unsafe](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/unsafe) | Deklariert einen nicht sicheren Kontext. |
| [virtual](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/virtual) | Deklariert eine Methode oder eine Zugriffsmethode, deren Implementierung durch einen überschreibenden Member in einer abgeleiteten Klasse geändert werden kann. |
| [volatile](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/volatile) | Gibt an, dass ein Feld im Programm z. B. vom Betriebssystem, der Hardware oder einem gleichzeitig ausgeführten Thread geändert werden kann. |

### Zugriffsebenen

Verwenden Sie die Zugriffsmodifizierer public, protected, internal oder private, um eine der folgenden deklarierten Zugriffsebenen für Member anzugeben.

<https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/accessibility-levels>

|  |  |
| --- | --- |
| **Deklarierter Zugriff** | **Bedeutung** |
| [public](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/public) | Der Zugriff ist nicht beschränkt. |
| [protected](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/protected) | Der Zugriff ist auf die enthaltende Klasse oder auf Typen beschränkt, die von der enthaltenden Klasse abgeleitet sind. |
| [internal](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/internal) | Der Zugriff ist auf die aktuelle Assembly beschränkt. |
| [protected internal](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/protected-internal) | Der Zugriff ist auf die aktuelle Assembly oder auf Typen beschränkt, die von der enthaltenden Klasse abgeleitet sind. |
| [private](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/private) | Der Zugriff ist auf die enthaltende Klasse beschränkt. |
| [private protected](https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/private-protected) | Der Zugriff ist auf die enthaltende Klasse oder auf Typen beschränkt, die von der enthaltenden Klasse innerhalb der aktuellen Assembly abgeleitet sind. Verfügbar seit C# 7.2. |

## Typen

Das C#-Typensystem enthält die folgenden Kategorien:

* Werttypen
* Verweistypen
* Zeigertypen

Variablen, die Werttypen sind speichern Daten, und solche, die Verweistypen sind, verweisen auf die tatsächlichen Daten. Instanzen von Verweistypen werden auch als Objekte bezeichnet. Zeigertypen können nur im unsafe-Modus genutzt werden.

<https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/types>

### Werttypen

Eine Variable eines Werttyps enthält einen Wert des Typs. Beispielsweise kann eine Variable von Typ int den 42-Wert enthalten

Die einfachen Typen sind eine Reihe von vordefinierten Strukturtypen, die von C# bereitgestellt werden, und umfassen die folgenden Typen:

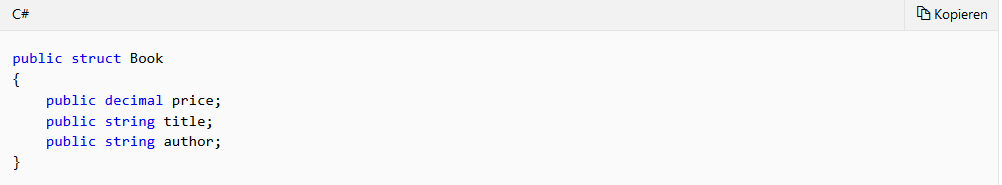
* Ganzzahltypen: ganzzahlige numerische Typen und der char-Typ, int-Typ
* Gleitkommatypen
* bool

Es gibt zwei Werttypen:

* Strukturen
* Enumerationen

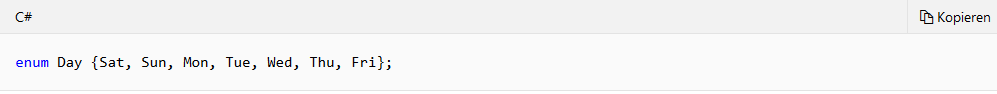
#### Strukturen

Ein struct-Typ ist ein ein Werttyp, der in der Regel verwendet wird, um eine kleine Gruppe verwandter Variablen zusammenzufassen



#### Enumerationem

Das Schlüsselwort enum wird zum Deklarieren einer Enumeration verwendet. Dies ist ein eigener Typ, der aus einer Gruppe benannter Konstanten besteht, die Enumeratorliste genannt wird.



### Verweistypen

Variablen von Verweistypen speichern Verweise auf ihre Daten (Objekte). Bei Verweistypen können zwei Variablen auf dasselbe Objekt verweisen. Daher können auf eine Variable angewendete Operationen das Objekt beeinflussen, auf das von der anderen Variablen verwiesen wird. Bei Werttypen besitzt jede Variable eine eigene Kopie der Daten, und auf eine Variable angewendete Operationen können die andere Variable nicht beeinflussen (außer im Fall von ref- und out-Parametervariablen, siehe in, ref und out-Parametermodifizierer).

Die folgenden Schlüsselwörter werden verwendet, um Verweistypen zu deklarieren:

* class
* interface
* delegate

C# enthält auch die folgenden integrierten Referenztypen:

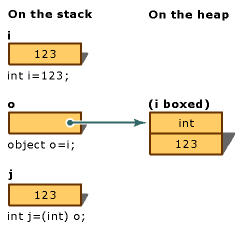
* dynamic
* object
* string

**dynamic**

Wird einstweilen nicht näher behandelt

**object**

Der object-Typ ist ein Alias für Object in .NET. Im vereinheitlichen Typsystem von C# erben alle Typen, vordefiniert und benutzerdefiniert sowie Verweis- und Werttypen, direkt oder indirekt von Object. Sie können Werte eines beliebigen Typs Variablen des Typs object zuweisen. Wenn eine Variable eines Werttyps in ein Objekt konvertiert wird, gilt es als geschachtelt. Wenn eine Variable eines Typobjekts in ein Wertobjekt konvertiert wird, gilt es als nicht geschachtelt. Weitere Informationen finden Sie unter Boxing und Unboxing (casten).



**string**

Wird einstweilen nicht näher behandelt

### Zeigertypen

Wird einstweilen nicht näher betrachtet.

# C#-Programmierhandbuch

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/index>

## Assemblies

Beinhaltet alle Projektmappen

## Namespaces

Hier werden Gültigkeitsreiche für eine Gruppe von Objekten u. Codes deklariert.

Innerhalb eines Namespace können Sie 0 (null) oder mehr der folgenden Typen deklarieren:

* Class
* Interfaces
* Struct
* Enum
* Delegate
* Nested namespace

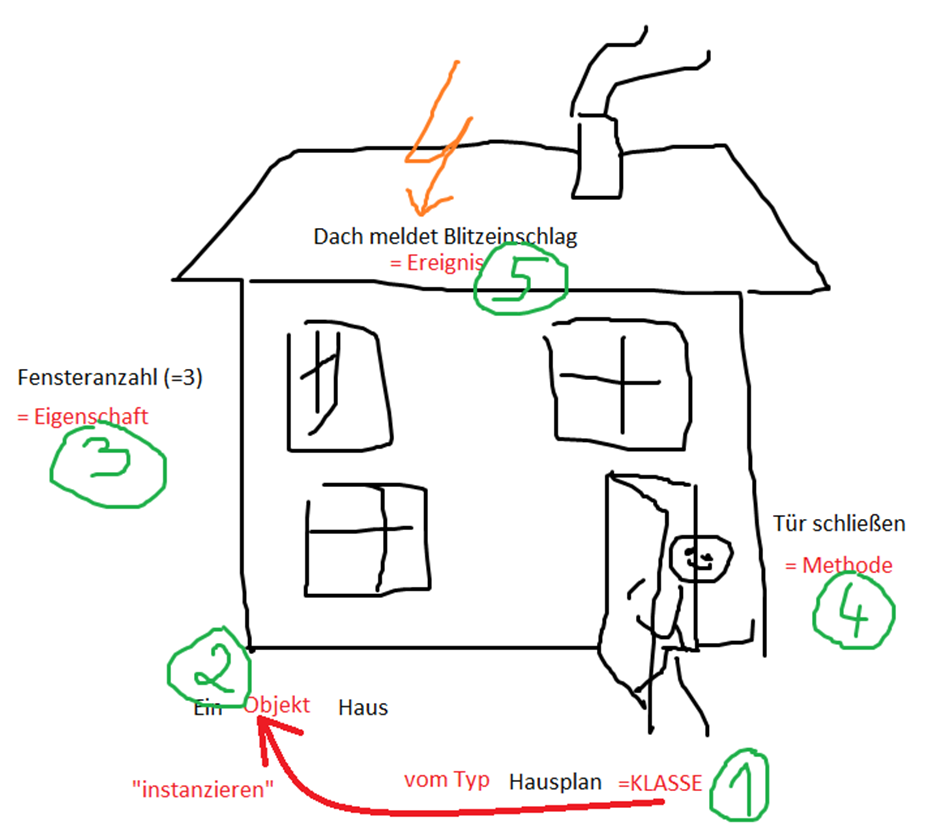
## Klassen

Die Klasse stellt den Haus Plan eines Programms dar. Klassen im westlichen durch

* Methoden => arbeitet
* Eigenschaften => beschreibt
* Ereignis => Mitteilungen

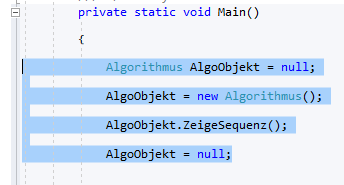
beschrieben werden.

Damit man mit einer Klasse arbeiten kann muss diese instanziiert werde also ein Objekt daraus erstellt werden.



### Instanzieren (Objekt erstellen)

Die Instanziierung der Klasse (Objekt Haus) erfolgt mit dem Schlüsselwort new.



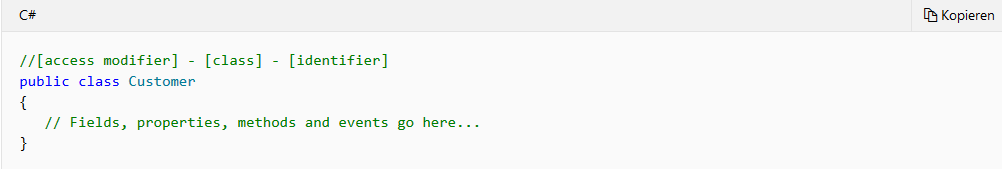
### Code - Klasse

public class Custommer

- Name der Methode

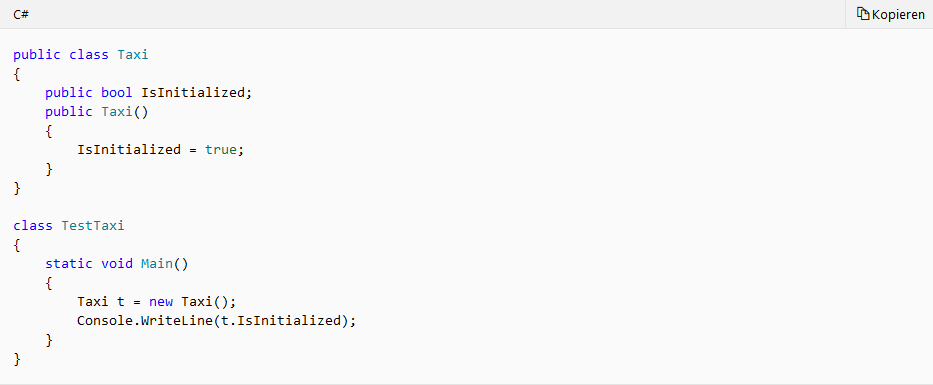
- Rückgabetyp

- Zugriffsmodifizierer

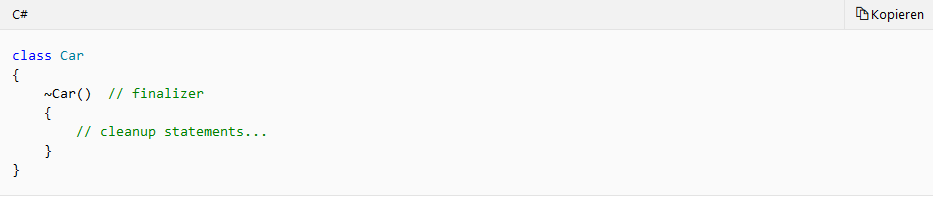


## Konstruktor und Destruktor

Der Konstruktor ist eine spezielle Funktion, welche bei der Initialisierung (also dem Erstellen) einer Klasse aufgerufen wird. Der Name der Konstruktor-Funktion entspricht immer dem Klassennamen. Die Funktion besitzt keinen Rückgabewert (auch nicht void), da der Konstruktor sozusagen das neu initialisierte Objekt zurückgibt. Wird kein Konstruktor in der Klasse erstellt, gibt es immer einen sogenannten Standard-Konstruktor, welcher keine weiteren Aktionen ausführt. Der Standard-Konstruktor besitzt keine Parameter. Jedoch kann dies bei der Implementierung eigener Konstruktoren geändert werden. Des Weiteren ist auch die Überladung des Konstruktors möglich. Wichtig zu wissen ist, dass sobald Konstruktoren in der Klasse deklariert werden, es den Standard-Konstruktor den C# automatisch erstellt, nicht mehr gibt. Der Zugriffsmodifizierer bei Konstruktoren ist normalerweise public. Wird als Zugriffsmodifzierer private verwendet, ist die Objektinitialisierung dieser Klasse nicht mehr möglich. Der private Konstruktor wird deshalb verwendet, um die Objektinitialisierung einer Klasse, welche lediglich statische Funktionen enthält, zu unterbinden.



Der Destruktor ist das Gegenteil des Konstruktors, da der Destruktor aufgerufen wird, wenn das Objekt zerstört / gelöscht wird. Wie bereits angesprochen, wird ein Objekt (welches auf dem Heap gespeichert ist) automatisch gelöscht, sobald das .NET Framework keine Referenzierung mehr findet. Dieses Löschen der Objekte geschieht automatisch durch den Garbage Collector (kurz GC), wenn das Programm aktuell keine Vorgänge zu bearbeiten hat oder der Speicherplatz kritisch wird. Des Weiteren ist es auch möglich, über die statische Funktion Collect() der Klasse GC das Löschen unreferenzierter Objekte manuell auszulösen. Der Name des Destruktors entspricht ebenfalls dem Klassennamen, jedoch mit einem vorangestellten Tilde-Zeichen (~). Beim Destruktor darf des Weiteren kein Zugriffsmodifizierer und Rückgabewert angegeben werden.



## Methode

Eine Methode ist ein Codeblock der eine Reihe von Anweisungen enthält

Die MAIN Methode ist der Einstiegspunkt.

### Code - Methode

public static void MethodenName ()

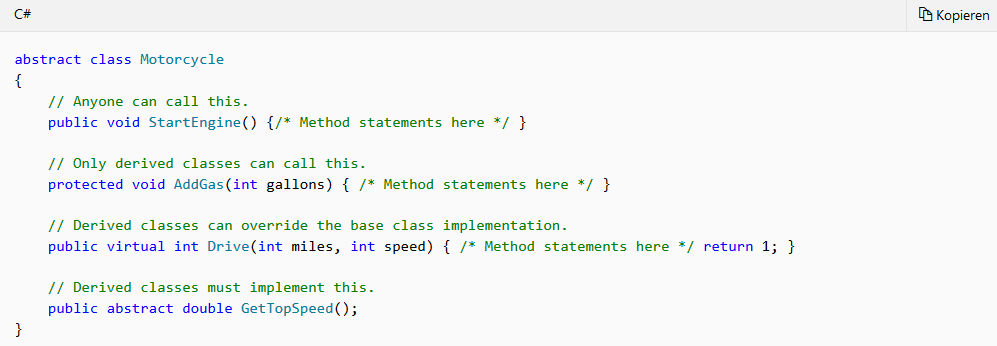
- Schnittstelle

- Name der Methode

- Rückgabetyp

- Zugriffsmodifizierer

Zugriffsmodifizierer



## Eigenschaft

Eine Eigenschaft ist ein Member, das einen flexiblen Mechanismus zum Lesen, Schreiben oder Berechnen des Werts eines privaten Felds bietet.

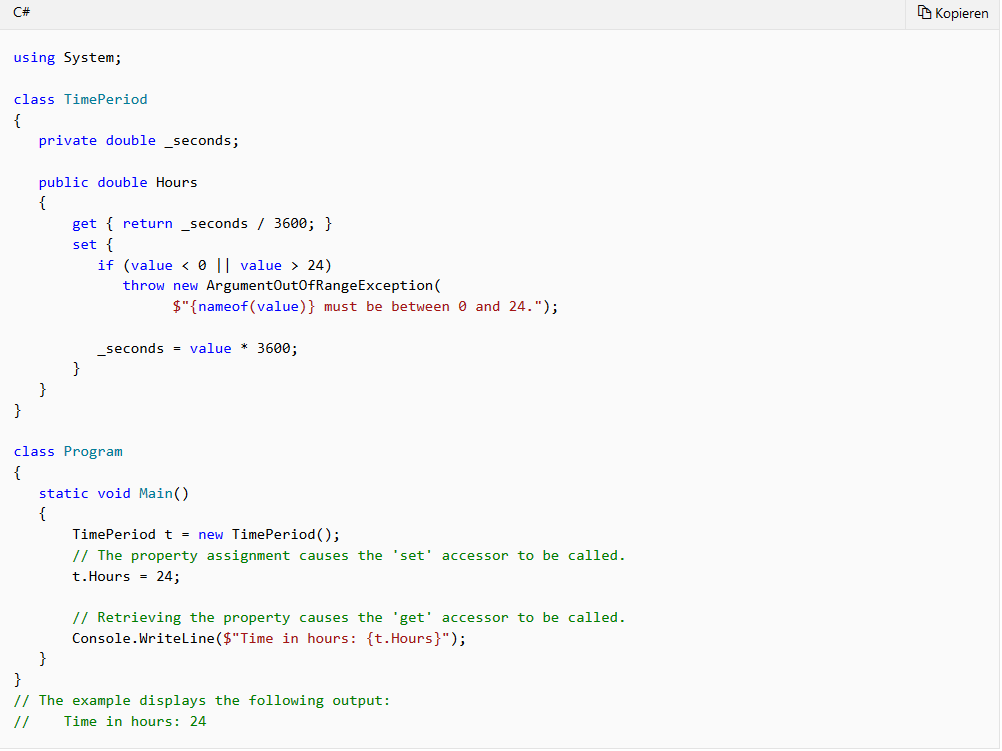
Mithilfe von Eigenschaften kann eine Klasse eine öffentliche Methode zum Abrufen und Festlegen von Werten verfügbar machen und dabei den Implementierungs- oder Verifizierungscode ausblenden.

Eine get-Eigenschaftenaccessor wird verwendet, um den Wert der Eigenschaft zurückzugeben. Ein set-Eigenschaftenaccessor wird verwendet, um einen neuen Wert zuzuweisen. Diese Zugriffsmethoden können über verschiedene Zugriffsebenen verfügen. Weitere Informationen finden Sie unter Einschränken des Accessorzugriffs.

Das **value**-Schlüsselwort wird verwendet, um den Wert zu definieren, der vom set-Accessor zugewiesen wird.

Eigenschaften können sein: Lesen/Schreiben (beide verfügen über einen get- und set-Accessor), schreibgeschützt (verfügen über einen get-Accessor, jedoch keinen set-Accessor), oder lesegeschützt (verfügen über einen set-Accessor, jedoch keinen get Accessor). Lesegeschützte Eigenschaften sind selten und werden am häufigsten verwendet, um den Zugriff auf vertrauliche Daten einzuschränken.

### Code - Eigenschaft



## Ereignis

### Code - Ereignis

## Schnittstellen

<https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/interfaces/>

### Code - Schnittstelle

## Delegate

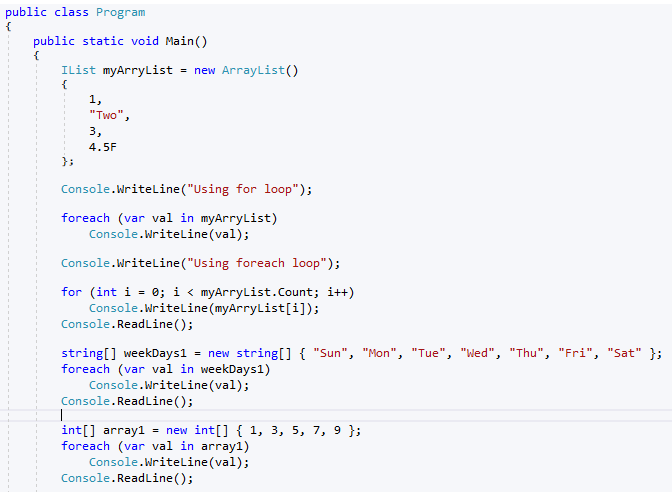
### Code - Delegate

## Arrays

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>

Wird nicht weiter behandelt.

### Code - Arrays



## Generika

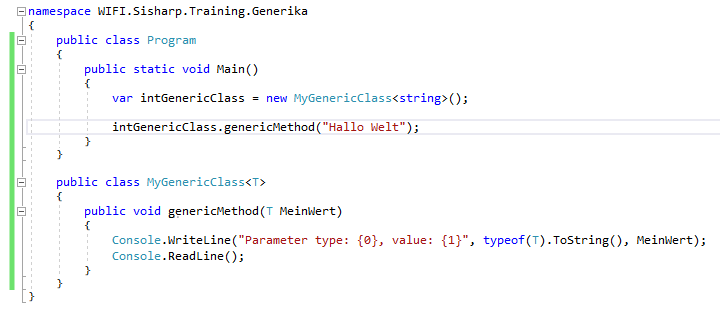
<https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/generics/>

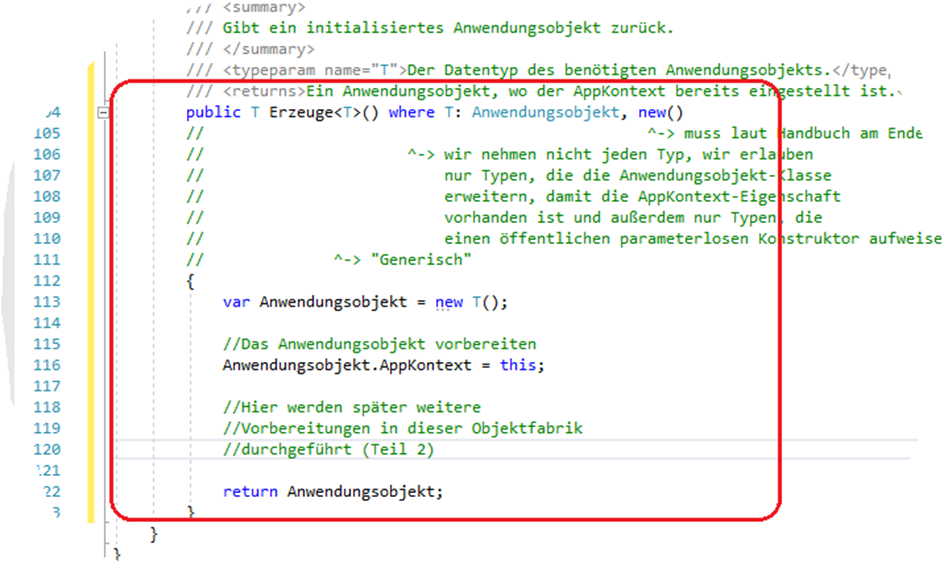
Generische Klassen und Methoden vereinen Wiederverwendbarkeit, Typsicherheit und Effizienz so, wie es ihre nicht generischen Gegenstücke nicht können. Generics werden am häufigsten für Auflistungen und deren Methoden verwendet.

Der Typparameter T wird an mehreren Stellen verwendet, wo für gewöhnlich ein konkreter Typ verwendet werden würde, um den Typ des Elements anzugeben, das in der Liste gespeichert wurde. Er wird wie folgt verwendet:

* Als Typ eines Methodenparameters in der Methode AddHead.
* Als Rückgabetyp der Eigenschaft Data in der geschachtelten Klasse Node.
* Als Typ des privaten Members data in der geschachtelten Klasse.

### Code – Generika



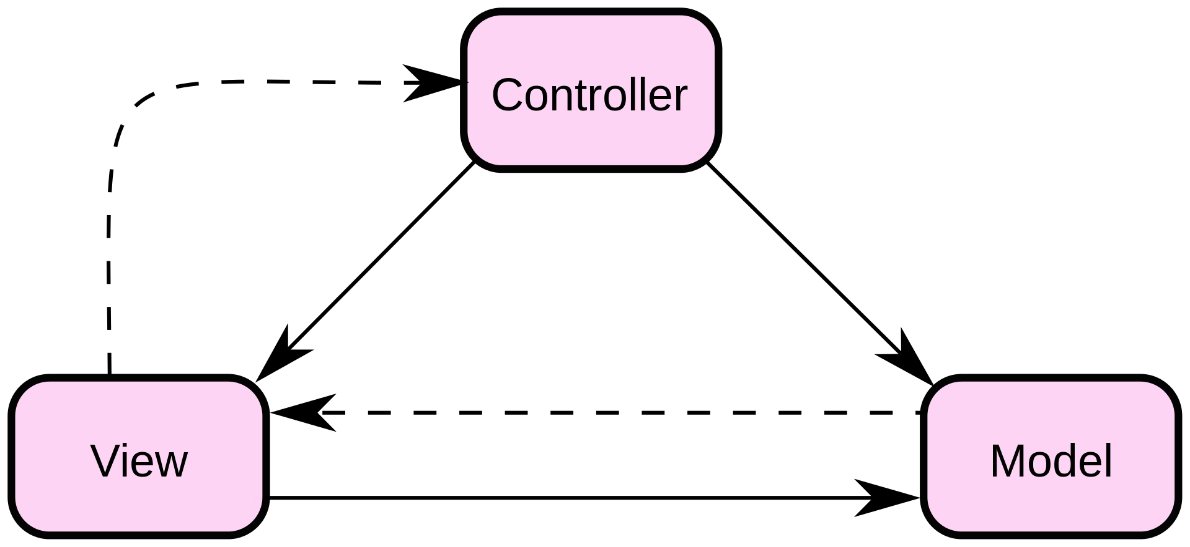


# Mehrschichtenmodel

Teil 1 setzt sich aus dem Model View Controller Mehrschichtenmodell zusammen, das die Datentransferobjekte und die Infrastruktur enthält.

<https://de.wikipedia.org/wiki/Model_View_Controller>

## Model View Controller



UserInterface (Windows Forms, WPF,…)

* Benutzeroberfläche, mit dem der User arbeiten kann

Controller

* Logik für den Zugriff auf die Daten bzw. dessen Erstellung

Geschäftslogik (Modell)

* Enthält Daten die von der Benutzeroberfläche dargestellt werden

## Firmenframework

Infrastruktur

* Hat zugriff auf alle Ebenen (ModelViewController) via Interfaces

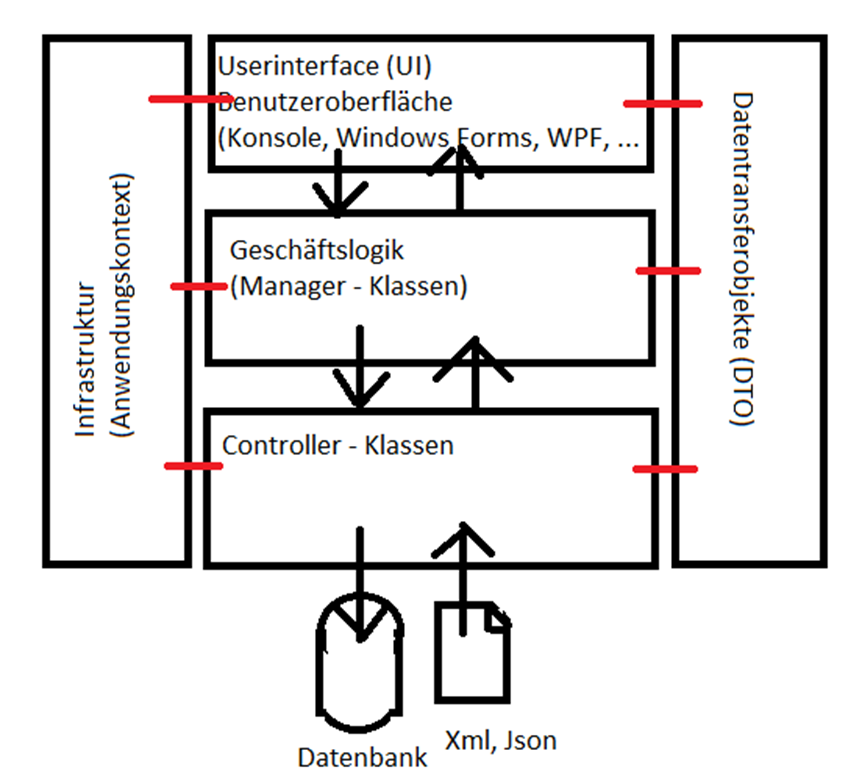
Datentranferobjekte (DTO)

* Hier werden die Daten aus dem Firmenframwork hinterlegt (Sprachen und Fenster)

Datenbank

* Hier sind alle relevanten Daten hinterlegt bzw. werden sie aus dieser abgerufen

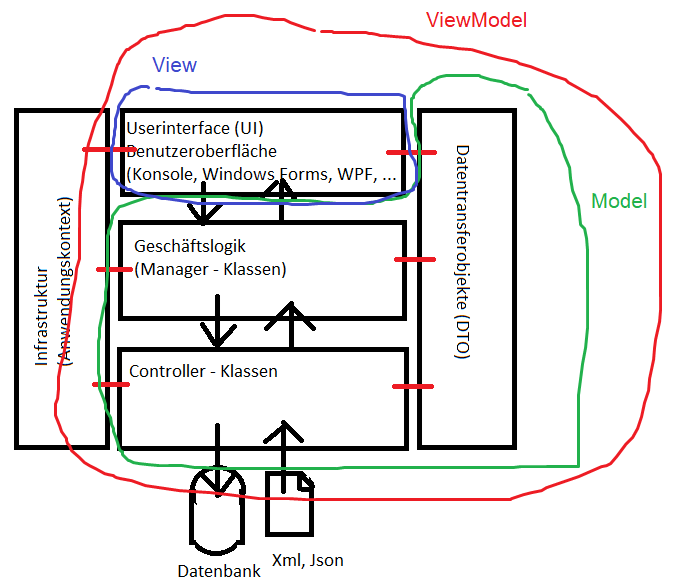
XML, Json



# WPF-Anwendung

<https://docs.microsoft.com/de-de/visualstudio/designers/getting-started-with-wpf?view=vs-2019>

## MVVM – Mehrschichtenmodel



### Model

Alles was mit den Daten zu tun hat, wird als „Model“ bezeichnet.

Controller

* Logik für den Zugriff auf die Daten bzw. dessen Erstellung

Geschäftslogik

* Enthält Daten die von der Benutzeroberfläche dargestellt werden

### View

Alles was mit der Benutzeroberfläche zu tun hat wird als “View” bezeichnet.

### View Model

Das Model und die Views werden von „ViewModel“ Klassen (Objekten) kontrolliert und gesteuert.

# Xaml

XAML Cheat Sheet

# Databinding

## Peer Control’s Properties

<Slider Name="myName" />

<TextBox Text="{Binding ElementName=myName, Path=Value}" />

<!-- SAME AS -->

<TextBox Text="{Binding Value, ElementName=myName}" />

<!-- EXAMPLE: -->

<TextBlock Text="{Binding SelectedItem.Header, ElementName=treeView}" />

## Binding Modes

<TextBox Text="{Binding Path=Text, ElementName=txtUsername Mode=TwoWay,

UpdateSourceTrigger=LostFocus}" />

Mode=OneTime

Mode=OneWay

Mode=OneWayToSource

Mode=TwoWay

## CLR Objects

### Properties

<!-- Binding to Local Properties -->

<ComboBox ItemsSource="{Binding Path=ClientsList, ElementName=myClass, Mode=Default}" />

<TextBlock Text="{Binding Source={StaticResource CurrentPerson}, ElementName=myClass,

Path=Age}" />

### Native Objects

<!-- Namespace Declaration for System namespace -->

xmlns:clr="clr-namespace:System;assembly=mscorlib"

<!-- Regular CLR Object Instantiation -->

<clr:Decimal x:Key="Current">352</clr:Decimal>

<!-- Complex Classes Instantiation -->

<src:Client x:Key="CurrentClient" ClientID="32" Active="True" />

<!-- Binding to CLR Objects -->

<TextBlock Text="{Binding Source={x:Static clr:DateTime.Now}}" />

### Methods

<ObjectDataProvider x:Key="InitializedData" ObjectType="{x:Type src:Person}" MethodName="GetUserByID">

<ObjectDataProvider.ConstructorParameters>

<clr:Int32>152</clr:Int32>

</ObjectDataProvider.ConstructorParameters>

</ObjectDataProvider>

## Converters

<!-- Converters must impliment IValueConverter -->

<BooleanToVisibilityConverter x:Key="VisConverter" />

<CheckBox Name="chkVisible" Content="Show Text" IsChecked="True" />

<TextBlock Text="Hidden Text" Visibility="{Binding IsChecked, ElementName=chkVisible,

ConvertParameter=n, Converter={StaticResurce VisConverter}}" />

## Data Template

<!-- Data Template -->

<DataTemplate DataType="{x:Type src:Client}">

<TextBlock Text="{Binding CleintName}" />

</DataTemplate>

<!-- The following will display using the data template as it's Listbox.ItemTemplate -->

<ListBox ItemsSource="{Binding Source={StaticResource CurrentClients}}" />

<!-- OR -->

<ListBox ItemsSource="{Binding Source={StaticResource CurrentClients}}" >

<ListBox.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding CleintName}" />

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

<!-- Hierarchical Data Templates for a TreeView -->

<HierarchicalDataTemplate DataType="{x:Type src:Client}"

ItemsSource="{Binding Path=Projects}">

<TextBlock Text="{Binding Path=Name}" />

</HierarchicalDataTemplate>

<HierarchicalDataTemplate DataType="{x:Type src:Project}"

ItemsSource="{Binding Path=Tasks}">

<TextBlock Text="{Binding Path=ProjectName}" />

</HierarchicalDataTemplate>

<DataTemplate DataType="{x:Type src:Task}">

<TextBlock Text="{Binding Path=TaskName}" ToolTip="{Binding Path=TaskName}" />

</DataTemplate>

## XPath

<!-- Binding to XML -->

<XmlDataProvider x:Key="xmlData" XPath="/" Source="Employees.xml" />

<ListBox ItemsSource="{Binding Source={StaticResource xmlData},XPath=Employees/IT}" />

## Relative Source

<TextBlock Text="{Binding RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type TextBlock}}}" />

<!-- FindAncestor Syntax -->

RelativeSource FindAncestor, AncestorLevel=n, AncestorType={x:Type desiredType}}

<!-- To bind to the previous data item in a data-bound collection -->

Text="{Binding RelativeSource={Binding RelativeSource={RelativeSource PreviousData}}}"

<!-- Useful for Templates -->

Text="{Binding RelativeSource={Binding RelativeSource={RelativeSource TemplatedParent}}}"

<!-- Binding to self -->

Text="{Binding RelativeSource={Binding RelativeSource={RelativeSource Self}}}"

<!-- EXAMPLE: -->

<Slider ToolTip="{Binding RelativeSource={RelativeSource Self}, Path=Value}" />

## Static Resources

<!-- Static Resources set a property with a value only once. -->

<!-- They are resources instantiated when the form loads. -->

<!-- They are very fast and efficent to apply, but you do have -->

<!-- the cost of instantiating the object up front even if you -->

<!-- never use the object. -->

<Window.Resources>

<Image x:Key="expandNode" Source="expand.png" />

</Window.Resources>

<Button Content="{StaticResource expandNode}" />

## Dynamic Resources

<!-- Dynamic resources allow the property to be updated when the resource changes -->

<!-- Also, Dynamic resources are not loaded until needed unlike StaticResources that-->

<!-- Load as soon as the XAML is loaded. However, DynamicResources are more expensive.-->

<!-- Perfect for themes because they change when a user changes colors in the CP -->

<Button Background="{DynamicResource {x:Static System.WindowBrush}}" />

## Style

<Application.Resources>

<!-- Inherit a style by useing BasedOn="{StaticResource GeneralStyle}" -->

<Style x:Key="TopButton" TargetType="{x:Type Button}" >

<Setter Property="Background" Value="#FFDDDDDD" />

</Style>

</Application.Resources>

…

<Button Style="{StaticResource TopButton}" />

## Triggers

<Style TargetType="{x:Type Button}">

<Style.Triggers>

<!-- Property Trigger -->

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Blue" />

</Trigger>

<!-- Multi Trigger -->

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsMouseOver" Value="True" />

<Condition Property="IsFocused" Value="False" />

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter Property="Foreground" Value="Blue" />

</MultiTrigger.Setters>

</MultiTrigger>

<!-- Data Trigger -->

<DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},

Path=Text}" Value="Blue">

<Setter Property="Forecolor" Value="Blue" />

</DataTrigger>

<!-- Event Trigger Example using a Storyboard Animation-->

<EventTrigger RoutedEvent="Loaded">

<EventTrigger.Actions>

<BeginStoryboard>

<!-- Insert Animations here-->

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

</Style.Triggers>

</Style>

## Control Template

<!-- Control Template -->

<Style x:Key="RoundButton">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate>

<Border Width="22" Height="22" CornerRadius="15">

<Border.Background>

<RadialGradientBrush GradientOrigin=".3, .3">

<GradientStop Color="White" Offset=".15" />

<GradientStop Color="#888" Offset="1" />

</RadialGradientBrush>

</Border.Background>

<TextBlock Foreground="#333" Text="{Binding Text}" />

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

<!-- Control Template for Content Control -->

<Style x:Key="GridListbox" TargetType="ListBox">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate>

<Grid>

<ScrollViewer VerticalScrollBarVisibility="Auto">

<ItemsPresenter />

</ScrollViewer>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

## Transforms

<!-- Rotate Transform -->

<Button RenderTransformOrigin="1,0" Content="Exit">

<Button.RenderTransform>

<RotateTransform Angle="90" />

</Button.RenderTransform>

</Button>

<!-- Other Transforms -->

<TranslateTransform <!-- Moves an object from its origin -->

<ScaleTransform <!-- Shrinks or Expands an object -->

<SkewTransform <!-- Skews an object -->

<MatrixTransform <!-- Custom Transform an object -->

# MySql