1 Die Debug-Klasse

In diesem Kapitel befassen wir uns mit Fehlern, die ein unerwartetes Verhalten des Programms verursachen, aber keine Ausnahme zur Laufzeit auslösen. Diese Fehler werden als *logische Fehler* bezeichnet und sie werden auch nicht vom Compiler erfasst.

Sicherlich ist dem einen oder anderen schon aufgefallen, dass man mit Hilfe der Konsole leicht überprüfen kann, ob Variablen zu gewissen Zeitpunkten korrekt belegt sind.

Mittels der Klasse `System.Diagnostics.Debug` ist es möglich, Variableninhalte in einem extra Ausgabefenster in Visual Studio auszugeben. Das ist hilfreich, wenn man keine Konsole verwenden möchte (zum Beispiel bei der Programmierung einer grafischen Benutzeroberfläche) oder wenn man die Konsole für sein Programm tatsächlich benötigt (dann würden zusätzliche Ausgaben in der Konsole stören).

Etwas in dieses Ausgabefenster zu schreiben, ist genau so einfach, wie etwas in die Konsole zu schreiben:

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      string hallo = "Hallo Welt";

      Debug.WriteLine(hallo);

      Console.ReadKey();
    }
  }
}
```

Das Ausgabefenster befindet sich in der Regel unterhalb Ihres Programms. Wenn es momentan jedoch nicht angezeigt wird, so können Sie es anzeigen lassen, indem Sie unter dem Menüpunkt »Ansicht« → »Ausgabe« auswählen.

Folgende Formatierung ist mit der Methode `Debug.WriteLine()` nicht mehr möglich:

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
```

```
  {
    static void Main(string[] args)
    {
      string hallo = "Hallo Welt";

      Debug.WriteLine("hallo: {0}", hallo);

      Console.ReadKey();
    }
  }
}
```

Wir können jedoch einfach den Operator + verwenden, wie wir es auch schon kennen.

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      string hallo = "Hallo Welt";

      Debug.WriteLine("hallo: " + hallo);

      Console.ReadKey();
    }
  }
}
```

Wer das erste Beispiel ausgeführt hat, wird festgestellt haben, dass folgender Text in dem Ausgabefenster erschienen ist:

```
Hallo Welt: hallo: {0}
```

Das liegt daran, dass man der Methode `Debug.WriteLine()` auch zwei Parameter übergeben kann. Dabei ist der zweite Parameter vom Datentyp *string* eine Beschreibung, die vor der eigentlichen Ausgabe erscheint.

Das zweite Beispiel schreibt diesen Text in das Ausgabefenster:

```
hallo: Hallo Welt
```

1.1 Methoden zur Ausgabe von Informationen

Die Klasse `System.Diagnostics.Debug` bietet uns auch noch weitere Methoden zur Ausgabe von Informationen an.

Die untenstehende Tabelle listet diese Methoden auf.

Methode	Beschreibung
`Write()`	Schreibt Informationen ohne Zeilenumbruch.
`WriteLine()`	Schreibt Informationen mit Zeilenumbruch.
`WriteIf()`	Schreibt Informationen ohne Zeilenumbruch bei Erfüllung einer angegebenen Bedingung.
`WriteLineIf()`	Schreibt Informationen mit Zeilenumbruch bei Erfüllung einer angegebenen Bedingung.

Die konditionalen Methoden erwarten bei ihrem Aufruf zwei Parameter (in dieser Reihenfolge):

- Einen Wahrheitswert, der angibt, ob die Information in das Ausgabefenster geschrieben wird (`true`) oder nicht (`false`).
- Informationen, die in das Ausgabefenster geschrieben werden sollen.

Die folgenden Beispiele demonstrieren die Verwendung der soeben vorgestellten Methoden.

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      string hallo = "Hallo Welt";
      int zahl = 10;
      bool check = true;

      Debug.WriteLineIf(check, zahl);
      Debug.WriteLineIf(false, zahl);
      Debug.WriteLineIf(check, hallo);
      Debug.WriteLineIf(check, "Inhalt von Zahl: " + zahl);

      Console.ReadKey();
    }
  }
}
```

Das Ausgabefenster dazu sieht dann dementsprechend aus:

```
10
Hallo Welt
Inhalt von Zahl: 10
```

Die Methoden Indent() und Unindent()

Es existieren verschiedene Methoden, die dazu dienen, ausgegebene Informationen einzurücken. Dadurch lässt sich die Ausgabe übersichtlicher gestalten.

Methode	Beschreibung
`Indent()`	Erhöht die Einzugsebene um eins.
`Unindent()`	Verringert die Einzugsebene um eins.

Zudem existieren noch Eigenschaften, mit denen die Einrückung manipuliert werden kann:

Eigenschaft	Beschreibung
`IndentLevel`	Speichert die Einzugsebene.
`IndentSize`	Speichert die Anzahl der Leerzeichen eines Einzugs.

Ein Beispiel dazu:

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      Debug.WriteLine("Hallo");
      Debug.Indent();
      Debug.WriteLine("Welt");
      Debug.Unindent();
      Debug.WriteLine("!");
      Debug.IndentLevel = 5;
      Debug.WriteLine("!");
      Debug.IndentSize = 1;
      Debug.WriteLine("!");

      Console.ReadKey();
    }
  }
}
```

Dieses Beispiel schreibt folgenden Text in das Ausgabefenster:

```
Hallo
   Welt
!
                !
    !
```

Dabei ist die erste Ausgabe nicht eingerückt.

Die zweite Ausgabe ist einfach eingerückt (mit einer standardmäßigen Einstellung von vier Leerzeichen).

Die dritte Ausgabe ist wiederum nicht eingerückt, da die Einrückung rückgängig gemacht wurde.

Die vierte Ausgabe ist nun fünffach eingerückt (wobei eine Einrückung weiterhin vier Leerzeichen entspricht).

Die fünfte und letzte Ausgabe ist ebenfalls fünffach eingerückt, jedoch entspricht eine Einrückung hier nur noch einem Leerzeichen.

Die Methode Fail()

Mit der Methode `Fail()` können wir eine Fehlermeldung erzeugen. Diese Fehlermeldung wird während der Laufzeit ausgegeben, sobald die Methode aufgerufen wird.

Beispiel:

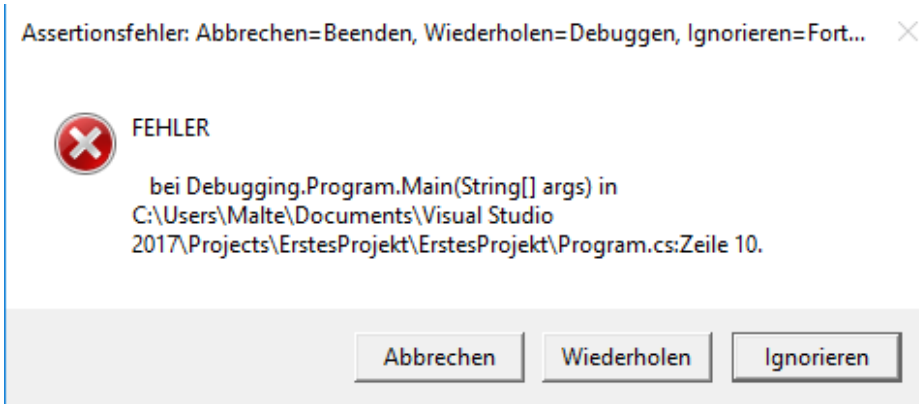
```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      Debug.Fail("FEHLER");

      Console.ReadKey();
    }
  }
}
```

Die Methode erwartet dabei als Parameter einen Wert vom Datentyp *string*, der die Fehlerbezeichnung angibt.

Das vorherige Beispiel würde nun eine solche Fehlermeldung erzeugen:



Die Methode Assert()

Durch die Verwendung dieser Methode können wir ebenfalls Fehlermeldungen erzeugen, jedoch machen wir diese von einer Bedingung abhängig. Diese Bedingung wird als erster Parameter übergeben, die anzuzeigende Meldung als zweiter Parameter.

Hierbei gilt zu beachten, dass die Fehlermeldung genau dann erscheint, wenn die Bedingung `false` ist (nicht, wenn sie `true` ist).

Auch hierzu soll es Beispiele geben:

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      bool check = true;
      Debug.Assert(check, "FEHLER");

      Console.ReadKey();
    }
  }
}
```

In diesem Beispiel würde *keine* Fehlermeldung auftreten, da die Bedingung wahr ist.

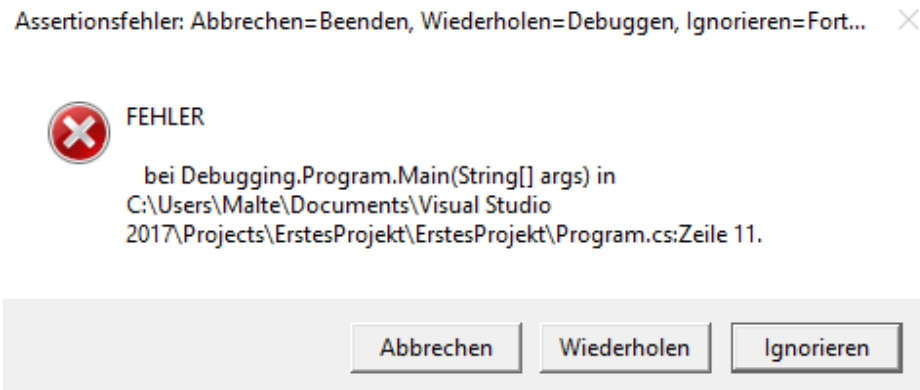
Das nächste Beispiel hingegen produziert eine Fehlermeldung.

```
using System;
using System.Diagnostics;
```

```
namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      bool check = false;
      Debug.Assert(check, "FEHLER");

      Console.ReadKey();
    }
  }
}
```

Nun erscheint wieder ein Fenster, das in etwa so aussieht:



Sinnvoll ist ein Einsatz der Methode `Assert()` zum Beispiel, wenn man prüfen möchte, ob eine Variable in einem bestimmten Wertebereich liegt (beispielsweise, ob der Wert einer Variablen negativ oder Null ist).

```
using System;
using System.Diagnostics;

namespace Debugging
{
  class Program
  {
    static void Main(string[] args)
    {
      int zahl = 20;
      Debug.Assert((zahl < 0), "FEHLER");

      Console.ReadKey();
    }
  }
}
```

Dieses Programm verursacht keine Fehlermeldung.

Allgemein können Fehlermeldungen dieser Art auf drei verschiedene Weisen behandelt werden:

- **Abbrechen**: Beendet das Programm.
- **Wiederholen**: Wechselt in den Debug-Modus.
- **Ignorieren**: Beendet die Fehlermeldung und setzt die Programmausführung fort.

Verweise

- »Rheinwerk Computing :: Visual C# 2012 - 7 Fehlerbehandlung und Debugging«, `http://openbook.rheinwerk-verlag.de/visual_csharp_2012/1997_07_002.html#dodtp38a43e10-de8f-40f7-9787-04abea14189a`, 03.10.2017
- »Debug-Klasse (System.Diagnostics)«, `https://msdn.microsoft.com/de-de/library/system.diagnostics.debug(v=vs.110).aspx`, 03.10.2017