Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

C# Grundlagen III

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Einführung in die Basiselemente der Programmiersprache C# - Eingabe/Ausgabe, Ausnahmen`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/05_CsharpGrundlagenIII.md
Autoren	@author

I/O Schreiboperation

C# selbst besitzt keine Anweisungen für die Ein- und Ausgabe von Daten, dazu existieren aber mehrere Bibliotheken, die im folgenden für die Bildschirmausgabe und das Schreiben in Dateien vorgestellt werden sollen.

Für das Schreiben stehen zwei Methoden `System.Console.Write` und `System.Console.WriteLine`.

Diese decken erstens eine **große Bandbreite von Übergabeparametern** und bedienen zweitens verschiedene **Ausgabeschnittstellen**.

	WriteLine() Methoden	Anwendung
1	`WriteLine()`	Zeilenumbruch
2	`WriteLine(UInt64)`, `WriteLine(Double)`, `WriteLine(Object)`	Ausgabe von Variablen der Basisdatentypen
3	`WriteLine(Object)`	
4	`WriteLine(String)`	Ausgabe einer `string` Variable
5	`WriteLine(String, Object)`	Kombinierte Formatierung- String mit Formatinformationen und nachfolgendes Objekt …
6	`WriteLine(String, Object, Object)`	… nachfolgende Objekte
7	`WriteLine(String, Object, Object, Object)`	…
8	`WriteLine(String, Object[])`	…

Einzelner Datentyp

Die Varianten 1-4 aus vorhergehender Tabelle übernehmen einen einzelnen Datentypen und geben dessen Wert aus. Dabei wird implizit die Methode `toString()` aufgerufen.

- Erinnerung an Steuerzeichen /n /t –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    Console.Write("One");
    Console.Write("Two");
    Console.WriteLine("Three");
    Console.WriteLine();
    Console.WriteLine("Four");
  }
}
```

Achtung, für Referenzdatentypen bedeutet dies, dass die Referenz ausgegeben wird.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    double[] values = new double[] {43.2234, 1.23123243, -123234.09};
    Console.WriteLine(values);
    Console.WriteLine(values.ToString());
  }
}
```

Lassen Sie uns schon mal etwas in die Zukunft schauen und die objektorientierte Implementierung dieser Idee erfassen:

```
using System;

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}

public class Program
{
  static void Main(string[] args)
  {
    Person person = new Person { Name = "Bernhard von Cotta", Age = 55 };
    Console.WriteLine(person);
  }
}
```

Kombinierte Formatierung von Strings

Die “Kombinierte Formatierung” unter C# ermöglicht eine breite Festlegung bezüglich des Formats der Ausgaben. Die folgenden Aussagen beziehen sich dabei aber nicht nur auf die Anwendung im Zusammenhang mit `WriteLine()` und `Write()` sondern können auch auf:

- String.Format (und damit ToString()!)
- String.Builder
- Debug.WriteLine …

angewandt werden.

Die Ausdrücke folgen dabei folgenden Muster. Achtung, die rechteckigen Klammern illustrieren hier optionale Elemente `[...]`!

```
{ index [, alignment] [width][:format][precision]}
```

Element	Bedeutung
`index`	Referenz auf die nachfolgenden Folge der Objekte
`alignment`	Ausrichtung links- (`-`) oder rechtsbündig
`width`	Breite der Darstellung
`format`	siehe nachfolgende Tabellen
`precision`	Nachkommastellen bei gebrochenen Zahlenwerten

- Diskussion der Indizes, Durchtauschen der Indizes,
- Erzeugung Fehler Indizes
- Darstellung der Breite
- Positives negatives Vorzeichen der Breite
- Angabe der Formate, precision –>

Probieren sie es doch in folgendem Beispiel mal aus:

```
using System;

public class Program
{
  static void Main(string[] args)
  {
        double[] values = new double[] {43.2234, 1.23123243, -123234.09};
        for (int index = 0; index < values.Length; index++)
        {
        Console.WriteLine("{0} -> {1}", index, values[index]);
        }
  }
}
```

Format Symbol	Bedeutung	Beispiel
G	Default	
E, e	Exponentiell	1.052033E+003
F, f	Festkomma	123.45
X, x	Hexadezimal	1FF
P, p	Prozent	-38.8%
D, d	Dezimal	1231
N, n	Dezimal mit Trennzeichen	1.23432,12

Eine komplette Auflistung findet sich unter https://docs.microsoft.com/de-de/dotnet/standard/base-types/standard-numeric-format-strings

> **Achtung:** Die Formatzeichen sind typspezifisch, es exisiteren analoge Zeichen mit unterschiedlicher Bedeutung für Zeitwerte

- Illustration, dass die Formatierungsmethoden auf verschiedenen Ebenen funktionen
- Hinweis auf fehlender Möglichkeit einer Breitenangabe
- Einführung einer cultural Instanz

–>

```
using System;
using System.Globalization;
using System.Threading;

public class Program
```

```
{
  static void Main(string[] args)
  {
    DateTime thisDate = new DateTime(2020, 3, 12);
    Console.WriteLine(thisDate.ToString("d"));   // d = kurzes Datum
                                                 // D = langes Datum
                                                 // f = vollständig

    string[]  cultures = new string [] {"de-DE", "sq-AL", "hy-AM"};
    foreach(string culture in cultures){
         Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
         Console.WriteLine("{0:5} - {1:D}", culture, thisDate);
    }
  }
}
```

Merke: Die kulturbezogene Ausgabe ist auch für das Komma bei den gebrochenen Zahlen relevant. Dieser Aspekt wird in den Übungen thematisiert Link.

Zeichenfolgeninterpolation

Wesentliches Element der Zeichenfolgeninterpolation ist die "Ausführung" von Code innerhalb der Ausgabespezifikation. Die Breite der Möglichkeiten reicht dabei von einfachen Variablennamen bis hin zu komplexen Ausdrücken. Hier bitte Augenmaß im Hinblick auf die Lesbarkeit walten lassen! Angeführt wird ein solcher Ausdruck durch ein `$`.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    // Composite formatting:
    var date = DateTime.Now;
    string city = "Freiberg";
    Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", city, date.DayOfWeek, date);
    // String interpolation:
    Console.WriteLine($"Hello, {city}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
  }
}
```

Die Darstellung des Ausdrucks folgt dabei der Semantik:

```
{<interpolatedExpression>[,<alignment>][:<formatString>]}
```

Damit lassen sich dann sehr mächtige Ausdrücke formulieren vgl. Link oder aber in den Beispielen von

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int ivalue = 33;
    double fvalue = 43.1231;

    // Berechnungen
    Console.WriteLine($"{ivalue * 10000,20}");
    Console.WriteLine($"{ivalue * 10000,20:E}");

    // Fallunterscheidungen mit ternärem Operator
    Console.WriteLine($"{ivalue < 5 ? ivalue.ToString() : "invalid"}");
    Console.WriteLine($"{(fvalue < 1000 ? fvalue : fvalue/1000):f2}"+
                      $"{fvalue < 1000 ? "Euro" : "kEuro"}");
```

```
  }
}
```

Ziele der Schreiboperationen

Üblicherweise möchte man die Ausgabegeräte (Konsole, Dateien, Netzwerk, Drucker, etc.) anpassen können. `System.IO` bietet dafür bereits verschiedene Standardschnittstellen.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
     Console.WriteLine("Ausgabe auf das Standard-Gerät");
     Console.Out.WriteLine("Ausgabe nach Out (default die Konsole)");
     //Console.Error.WriteLine("Ausgabe an die Fehlerschnittstelle");
  }
}
```

Die dafür vorgesehenen Standardeinstellungen können aber entsprechend umgelenkt werden. Dafür greift C# analog zu vielen anderen Sprachen (und Betriebssysteme) das Stream Konzept auf, dass eine Abstraktion für verschiedene Quellen und Senken von Informationen bereitstellt. Darauf aufbauend sind dann Lese- / Schreiboperationen möglich.

Einen guten Überblick dazu bietet das nachfolgende Tutorial:

!?Streams

Von der Klasse `System.IO.Stream` leiten sich entsprechend MemoryStream, FileStream, NetworkStream ab. Diese bringen sowohl eigene Lese-/Operationen mit, gleichzeitig ist aber auch das "Umlenken" von Standardoperationen möglich.

```
// Das Beispiel entstammt der Dokumentation des .Net Frameworks
// https://docs.microsoft.com/de-de/dotnet/api/system.console.out?view=netframework-4.7.2

using System;
using System.IO;

public class Example
{
   public static void Main()
   {
      // Get all files in the current directory.
      string[] files = Directory.GetFiles(".");
      Array.Sort(files);

      // Display the files to the current output source to the console.
      Console.WriteLine("First display of filenames to the console:");
      Array.ForEach(files, s => Console.Out.WriteLine(s));
      Console.Out.WriteLine();

      // Redirect output to a file named Files.txt and write file list.
      StreamWriter sw = new StreamWriter(@".\Files.txt");
      sw.AutoFlush = true;
      Console.SetOut(sw);
      Console.Out.WriteLine("Display filenames to a file:");
      Array.ForEach(files, s => Console.Out.WriteLine(s));
      Console.Out.WriteLine();

      // Close previous output stream and redirect output to standard output.
      Console.Out.Close();
      sw = new StreamWriter(Console.OpenStandardOutput());
      sw.AutoFlush = true;
```

```
      Console.SetOut(sw);

      // Display the files to the current output source to the console.
      Console.Out.WriteLine("Second display of filenames to the console:");
      Array.ForEach(files, s => Console.Out.WriteLine(s));
   }
}
```

Darüber hinaus stehen aber auch spezifische Zugriffsmethoden zum Beispiel für Dateien zur Verfügung.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO; //Erforderlicher Namespace
namespace Wiki
{
    class Program
    {
        static void Main(string[] args)
        {
            // Unmittelbare Ausgabe in eine Datei
            File.WriteAllText(@"C:\test1.txt", "Sehr einfach");

            // Einlesen des gesamten Inhalts einer Textdatei und
            // Ausgabe auf dem Bildschirm
            string text = File.ReadAllText(@"C:\test1.txt");
            Console.WriteLine(text);
        }
     }
}
```

> **Zur Erinnerung:** Das `@` vor einen String markiert ein verbatim string literal - alles in der Zeichenfolge, was normalerweise als Escape-Sequenz interpretiert werden würde, wird ignoriert.

Beispiel

Zur Erinnerung, in Markdown werden Tabellen nach folgendem Muster aufgebaut:

```
| Name     | Alter     | Aufgabe          |
|:---------|----------:|:-----------------|
| Peter    | 42        | C-Programmierer  |
| Astrid   | 23        | Level Designer   |
```

Name	Alter	Aufgabe
Peter	42	C-Programmierer
Astrid	23	Level Designer

Geben Sie die Daten bestimmte Fußballvereine in einer Markdown-Tabelle aus.

```
using System;


public class Program
{
  static void Main(string[] args)
  {
    string [] clubs = {"Blau Weiß", "Grün Gelb 1905", "Borussia Tralla Trulla", "Eintracht"};
    int [] punkte = {12, 10, 9, 5};
    // Wie lang ist ein Clubname maximal?
    int maxlength = 0;
    foreach(string club in clubs)
    {
```

```
      maxlength =  club.Length < maxlength ? maxlength : club.Length ;
    }
    maxlength += 1;
    // Ausgabe
    string output;
    output  = "| ";
    output += "Verein".PadRight(maxlength, ' ') + "| Punkte |\n";
    output += "|:"+ "".PadRight(maxlength, '-') + "|:-------|\n";
    for (int i = 0; i < clubs.Length; i++){
       output += String.Format("| {0}| {1, -7}|\n", clubs[i].PadRight(maxlength, ' '), punkte[i]);
    }
    Console.WriteLine(output);
  }
}
```

Frage: Welche Annahmen werden implizit bei der Erstellung der Tabelle getroffen? Wo sehen Sie Verbesserungsbedarf?

I/O Leseoperationen

Eine Software ist zumeist von Nutzereingaben abhängig. Diese können:

- in Form von Dateiinhalten daherkommen,
- als Eingaben beim Start des Programms vorgegeben oder
- zur Laufzeit eingelesen werden.

Vergegenwärtigen Sie sich die Vor- und Nachteile der unterschiedlichen Formen mit Blick auf die Entwicklung einer Software!

Kommandozeilenargumente

Eine grundlegende Eingabemöglichkeit ist die Übergabe von Parametern beim Aufruf des Programms von der Kommandozeile.

```
using System;

public class Program
{
  static int Main(string[] args)
  {
    System.Console.WriteLine($"How many arguments are given? - {args.Length}");
    foreach (string argument in args)
    {
      System.Console.WriteLine(argument);
    }
    return 0;
  }
}
```

Bei der Vorgabe von Commandline Parametern unterscheidet sich der Ablauf etwas, je nachdem wie Sie Ihren Code “bauen”.

Kommandozeile

1. mono Compiler

```
mcs CmdLineParams.cs
mono CmdLineParams.exe Das ist "ein Test"
```

2. dotnet Umgebung

```
dotnet new console --name dotnet_cmdLinePara --framework net5.0
cd dotnet_cmdLinePara
dotnet run -- -1 -zwei -"drei vier"
```

Entwicklungsumgebung am Beispiel von VS Code

Suchen Sie den zugehörigen `.vscode` Ordner innerhalb Ihres Projektes und öffenen Sie `launch.json`. Ergänzen Sie die Kommandozeilenparameter für den Punkt `args`.

```
"configurations": [
    {
        "name": ".NET Core Launch (console)",
        "args": [], // PUT YOUR ARGUMENTS HERE
         ...
    }
]
```

Auswertung der Kommandozeilenparameter

> **Achtung:** Im folgenden bemühen wir uns selbst um das Parsen der Kommandozeilenargumente. Dies sollte in realen Vorhaben entsprechenden Bibliotheken überlassen werden Using System.CommandLine!

```
using System;

public class Program
{
  static int Main(string[] args)
  {
    System.Console.WriteLine("Geben Sie einen Ganzzahlwert und einen String als Argumente ein!");
    if (args.Length == 0)
    {
        System.Console.WriteLine("Offenbar keine Eingabe - Fehler!");
        return 1;
    }
    if ((args.Length == 1) || (args.Length > 2))
    {
        System.Console.WriteLine("Falsche Zahl von Parametern - Fehler!");
        return 1;
    }
    if (args.Length == 2)  // Erwartete Zahl von Parametern
    {
        // hier müssen wir jetzt die Daten parsen und die Datentypen evaluieren
    }
    return 0;
  }
}
```

Im übernächsten Abschnitt prüfen wir den Datentyp und den Inhalt der Parameter auf Ihre Korrektheit.

Leseoperationen von der Console

Leseoperationen von der Console (oder anderen Streams) werden durch zwei Methoden abgebildet:

```
public static int Read ();
public static string ReadLine ();
```

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    char ch;
    int x;
    Console.WriteLine("Print Unicode-Indizes");
    do
      {
        x = Console.Read();   // Lesen eines Zeichens
        ch = Convert.ToChar(x);
```

```
        Console.Write($"Unicode {x}- Sign {Convert.ToChar(x)}\n");
        // Hier könnte man jetzt eine Filterung realiseren
      } while (ch != '+');
  }
}  //   zu Demonstrationszwecken
```

Das Beispiel zeigt sehr schön, wie verschiedene Zeichensätze auf unterschiedlich lange Codes abgebildet werden. Das chinesische Zeichen, dass vor dem Escape-Zeichen "+" steht generiert einen 2Byte breiten Wert.

Transformation der Eingaben

```
using System;

public class Program
{
  static int Main(string[] args)
  {
    Console.WriteLine("Geben Sie einen Ganzzahlwert und einen String als Argumente ein!");
    string inputstring = Console.ReadLine ();
    string[] param = inputstring.Split(' ');

    foreach(string s in param){
        Console.Write(s + " ");
    }

    if (param.Length == 0)
    {
        Console.WriteLine("Offenbar keine Eingabe - Fehler!");
        return 1;
    }
    if (param.Length == 2)  // Erwartete Zahl von Parametern
    {
        long num1 = long.Parse(param[0]);
        long num2 = System.Convert.ToInt64(param[0]);
        long num3;
        long.TryParse(param[0], out num3);
        Console.WriteLine($"{num1} {num2} {num3}");
        string text = param[1];
        Console.WriteLine($"{text}");
    }

    return 0;
  }
}
```

Dabei nutzt das obige Beispiel 3 Formen der Interpretation der Daten. In den beiden ersten Fällen ist der Entwickler für das Abfangen der *Exceptions* verantwortlich. Die letzte Variante kapselt dies intern und gibt die möglicherweise eingetretene Ausnahme über die Rückgabewerte aus. Der Code auf Seiten der Anwendung wird kompakter.

1. `long.Parse` parst die Eingabe in einen ganzzahligen Wert, wirft jedoch eine Ausnahme aus, wenn dies nicht möglich ist, wenn die bereitgestellten Daten nicht numerisch sind.
2. `Convert.ToInt64()` konvertiert die Zeichenkettendaten in einen korrekten echten int64-Wert und wirft eine Ausnahme aus, wenn der Wert nicht konvertiert werden kann.
3. Einen alternativen Weg schlägt `int.TryParse()` ein. Die TryParse-Methode ist wie die `Parse`-Methode, außer dass die `TryParse`-Methode keine Ausnahme auslöst, wenn die Konvertierung fehlschlägt.

Auf die Verwendung der Ausnahmen wird im folgenden Abschnitt eingegangen.

Ausnahmebehandlungen

Ausnahmen sind Fehler, die während der Ausführung einer Anwendung auftreten. Die C#-Funktionen zur Ausnahmebehandlung unterstützen bei der Handhabung von diesen unerwarteten oder außergewöhnlichen Situationen, die beim Ausführen von Programmen auftreten.

```
using System;
using System.Globalization;

public class Program
{
  static void Main(string[] args)
  {
      // Beispiel 1: Zugriff auf das Filesystem eines Rechners aus dem Netz
      System.IO.FileStream file = null;
      //System.IO.FileInfo fileInfo = new System.IO.FileInfo(@"NoPermission.txt");
      // Beispiel 2: Division durch Null
      int a = 0, b = 5;
      //a = b / a;
      //long num = long.Parse("NN");
  }
}
```

Merke: Wenn für eine spezifische Ausnahme kein Ausnahmehandler existiert, beendet sich das Programm mit einer Fehlermeldung.

Schlüsselwort	Bedeutung
`try`	Ein `try`-Block wird verwendet, um einen Bereich des Codes zu kapseln. Wenn ein Code innerhalb dieses `try`-Blocks eine Ausnahme auslöst, wird diese durch den zugehörigen `catch`-Block behandelt.
`catch`	Hier haben Sie die Möglichkeit, die (spezifische) Ausnahme zu behandeln, zu protokollieren oder zu ignorieren.
`finally`	Der `finally`-Block ermöglicht es Ihnen, bestimmten Code auszuführen, wenn eine Ausnahme geworfen wird oder nicht. Zum Beispiel das Entsorgen eines Objekts aus dem Speicher.
`throw`	Das `throw`-Schlüsselwort wird verwendet, um eine neue Ausnahme zu erzeugen, die in einem `try-catch-finally`-Block aufgefangen wird.

```
try
{
   // Statement which can cause an exception.
}
catch(Type x)
{
   // Statements for handling the exception
}
finally
{
   //Any cleanup code
}
```

Dabei gelten folgende Regeln für den Umgang mit Ausnahmen:

- Alle Ausnahmen sind von `System.Exception` abgeleitet und enthalten detaillierte Informationen über den Fehler, z.B. den Zustand der Aufrufliste und eine Textbeschreibung des Fehlers.
- Ausnahmen, die innerhalb eines `try`-Blocks auftreten, werden auf einen Ausnahmehandler, der mit dem Schlüsselwort `catch` gekennzeichnet ist, umgeleitet.
- Ausnahmen werden durch die CLR ausgelöst oder in Software mit dem `throw` Befehl.
- ein `finally`-Block wird im Anschluss an die Aktivierung eines `catch` Blockes ausgeführt, wenn eine Ausnahme ausgelöst wurde. Hier werden Ressourcen freizugeben, beispielsweise ein Stream geschlossen.

```
using System;

public class Program
```

```
{
  static int Berechnung(int a, int b)
  {
    int c=0;
    try
    {
        checked {c = a + b;}        // Fall 1
        // c = a / b;                // Fall 2
    }
    catch (OverflowException e)
    {
      Console.WriteLine("[Overflow] " + e.Message);
      //throw;
    }
    finally
    {
        Console.WriteLine("Finally Block");
    }
    Console.WriteLine("Hier sind wir am Ende der Funktion!");
    return c;
}

static void Main(string[] args)
{
    // try
    // {
      Berechnung(int.MaxValue, 1);     // Fall 1
      //Berechnung(2, 0);              // Fall 2
    // }
    /*
    catch (DivideByZeroException e)
    {
      Console.WriteLine("[DivideByZero] " + e.Message);
    }
    catch (Exception e)
    {
        Console.WriteLine("[GeneralExept] " + e.Message);
    }
    */
    Console.WriteLine("Hier sind wir am Ende des Hauptprogrammes!");
  }
}
```

Best Practice

Die folgende Darstellung geht auf die umfangreiche Sammlung von Hinweisen zum Thema Exceptions unter https://docs.microsoft.com/de-de/dotnet/standard/exceptions/best-practices-for-exceptions zurück.

- Differenzieren Sie zwischen Ausnahmevermeidung und Ausnahmebehandlung anhand der erwarteten Häufigkeit und der avisierten “Signalwirkung”

```
// Ausnahmevermeidung
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}

// Ausnahmebehandlung
try
{
    conn.Close();
}
```

```
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

- Nutzen Sie die Möglichkeite von Ausnahmen statt einen Fehlercode zurückzugeben
- Verwenden Sie dafür die vordefinierten .NET-Ausnahmetypen!
- Selbst definierte Ausnahmen enden auf das Wort *Exception*!
- Vermeiden Sie unklare Ausgaben für den Fall einer Ausnahmebehandlung, stellen Sie alle Informationen bereit, die für die Analyse des Fehlers nötig sind
- Stellen Sie den Status einer Methode wieder her, die von einer Ausnahmebehandlung betroffen war (Beispiel: Code für Banküberweisungen, Abbuchen von einem Konto und Einzahlung auf ein anderes. Scheitert die zweite Aktion muss auch die erste zurückgefahren werden.)
- Testen Sie Ihre Ausnahmebehandlungsstrategie!

Beispiel Exception-Handling

Schreiben Sie die Einträge eines Arrays in eine Datei!

Lösung unter ExceptionHandling.cs

Schritt	Fragestellungen
1	Welche Fehler können überhaupt auftreten? Welche Fehler werden durch die Implementierung abgefangen?
2	Wo sollen die Fehler abgefangen werden?
3	Gibt es Prioritäten bei der Abarbeitung?
4	Sind abschließende “Arbeiten” notwendig?

Link auf die Dokumentation der StreamWriter Klasse

Aufgaben

- [ ] Entwickeln Sie ein Programm, dass als Kommandozeilen-Parameter eine Funktionsnamen und eine Ganzzahl übernimmt und die entsprechende Ausführung realisiert. Als Funktionen sollen dabei `Square` und `Reciprocal` dienen. Der Aufruf erfolgt also mit

```
mono Calculator Square 7
mono Calculator Reciprocal 9
```

Welche Varianten der Eingaben müssen Sie prüfen? Erproben Sie Ihre Lösung mit einem besonders “böswilligen” Nutzer :-)

```
using System;

class MainClass
{
  static double Square(int num) => num * num;
  static double Reciprocal (int num) => 1f / num;
  static void Main(string[] args)
  {
    bool Error = false;
    double result = 0;
    int num = 1;
    if (args.Length == 2)
    {
      // Hier geht es weiter, welche Fälle müssen Sie bedenken?
      // int.TryParse(args[1], out num) erlaubt ein fehlertolerantes Parsen
      // eines strings
    }
    else Error = true;
    if (Error)
```

```
    {
      Console.WriteLine("Please enter a function and a numeric argument.");
      Console.WriteLine("Usage: Square    <int> or\n       Reciprocal <int>");
    }
    else
    {
      Console.WriteLine("{0} Operation on {1} generates {2}", args[0], num, result );
    }
  }
}
```