

Grundlagen der C# Programmierung

Organisation

- Kursablauf
 - Zeiten
 - Pausen
- Räumlichkeiten
- Kursunterlagen
- Slides später Online verfügbar
- Fragen? Fragen!

Ihr Trainer

- Artur Wanner
 - Informatik an der Fachhochschule Münster
 - Softwareentwickler
- ppedv AG, Standort Leipzig
- Kurse: Xamarin, C# Grundlagen, ASP.NET Core
- Kontakt
 - E-Mail: arturw@ppedv.de
 - 0341/59 48 35 - 61

ppedv AG

- Firmensitz in Burghausen
- Schulungszentren
- Schulungen für nahezu alle Microsoft-Technologien
- Konferenzen, Camps, Verlag (VisualStudioOne / VSOOne)
- Website: www.ppedv.de



Agenda

- Tag 1:
 - C# Grundlagen (Variablen, Schleifen, Kontrollstrukturen)
 - Konsolenanwendungen mit Ein-/Ausgabe
 - Funktionen und Parameter
 - Arrays und Aufzählungstypen
- Tag 2:
 - Objektorientierte Programmierung (Klassen/Objekte)
 - Vererbung
 - Interfaces und Polymorphismus
 - Generische Datentypen
- Tag 3 und 4:
 - Event-Handling und Delegates
 - Benutzeroberfläche gestalten mit WindowsForms
 - Exception-Handling
 - Unit-Tests
 - Dateizugriff
 - Abschlussprojekt

Allgemeines zu C#

- Weiterentwicklung von C und C++
 - 1972: C
 - 1979: C++
 - 2001: C#
- Merkmale
 - Case-sensitive
 - Objektorientiert
 - Garbage-Collection
 - CLR - Common Language Runtime (Laufzeitumgebung)
 - JIT – Just in Time Compiler

Allgemeines zu C# und .NET

- Anwendung
 - Microsoft .NET (Windows Desktop)
 - UWP – Windows 10 Apps
 - .NET Core – Windows/Linux/Mac
 - XAMARIN – iOS/Android
 - ASP.NET – Webserver
 - Spiele-Programmierung (Unity)
- Entwicklungsumgebungen
 - Visual Studio (aktuell Version 2017)
 - Xamarin Studio
 - Unity

Sprachgrundlagen

- Befehle:

```
Console.WriteLine(„Hello World“);  
Console.ReadKey();
```

- Zuweisungen

```
int Nummer = 20;  
string Wort = „Wort“;
```

- Kommentare

```
int Nummer = 20; //Einzeiliger Kommentar  
/*Mehrzeiliger  
Kommentar*/
```


Variablen und Datentypen

- Variablen sind Behälter für Werte
- Deklaration und Benutzung von Variablen:

```
int Alter;
```

```
Alter = 20; //Integer mit Wert 20
```

```
int Summe = Alter*2; //Summe bekommt Wert 40
```

```
string Stadt = „Berlin“; //Zeichenkette „Berlin“
```

```
Console.WriteLine(Stadt); //Berlin in Konsole schreiben
```

Datentypen

Schlüsselwort	Typ	
<code>char</code>	<code>System.Char</code>	Einzelnes Textzeichen
<code>string</code>	<code>System.String</code>	Text (max. 4GB)
<code>short</code>	<code>System.Short</code>	16-Bit-Integer
<code>int</code>	<code>System.Int32</code>	32-Bit-Integer
<code>long</code>	<code>System.Int64</code>	64-Bit-Integer
<code>float</code>	<code>System.Single</code>	16-Bit-Gleitkommazahl
<code>double</code>	<code>System.Double</code>	32-Bit-Gleitkommazahl
<code>decimal</code>	<code>System.Decimal</code>	128-Bit-Gleitkommazahl
<code>bool</code>	<code>System.Boolean</code>	true oder false

Standardgemäß:

- Ohne Nachkommastellen -> `int`
- Mit Nachkommastellen -> `double`

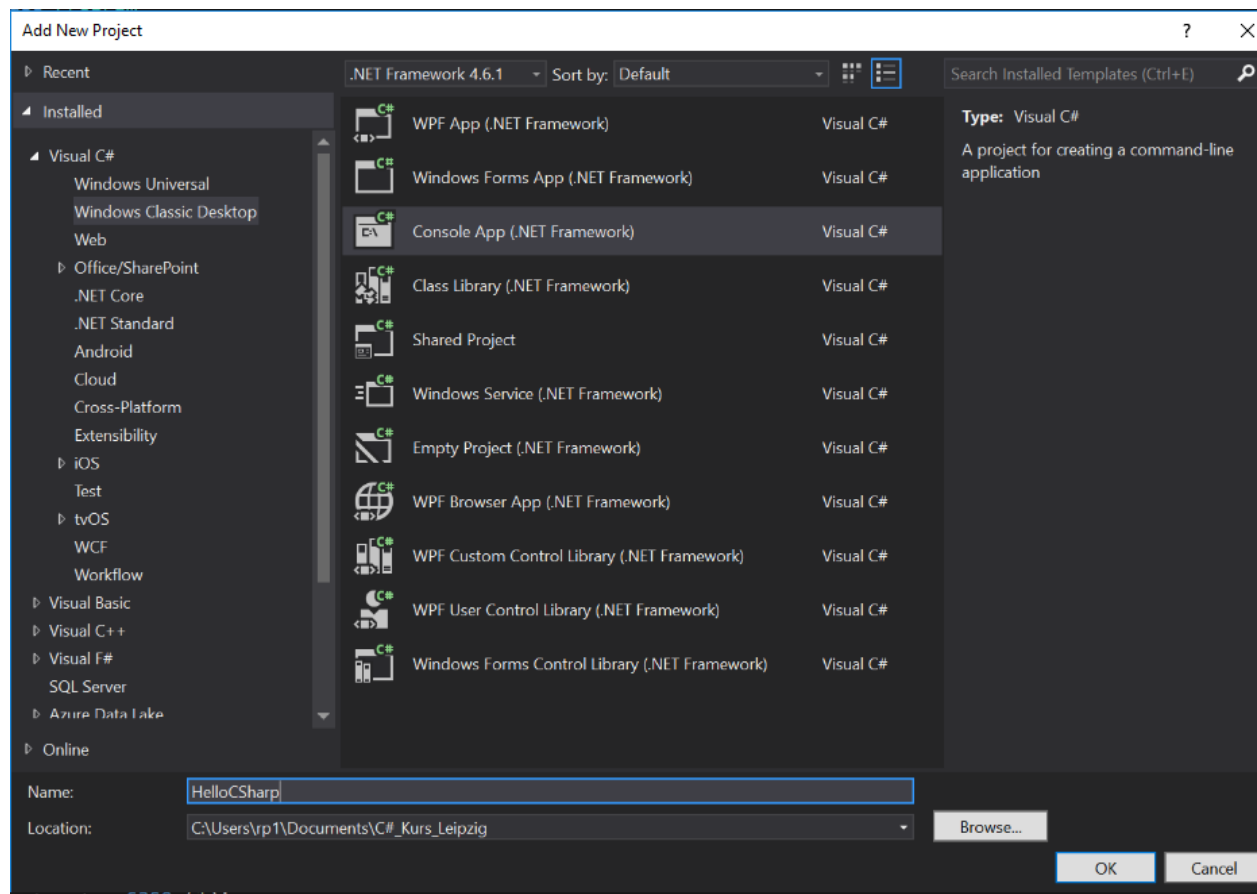
Ausgabe von Strings auf der Konsole

- 3 verschiedene Schreibweisen:

```
int alterVonMax = 20;  
Console.WriteLine("Alter von Max: " + alterVonMax);  
Console.WriteLine(string.Format("Max ist {0} Jahre alt.", alterVonMax));  
Console.WriteLine($"Max ist {alterVonMax} Jahre alt."); // Seit C#6
```

Hello C# Projekt

- Konsolenanwendung in Visual Studio
- Variablen deklarieren und Text ausgeben



Übung

Konsoleneingaben verarbeiten

- Einlesen von Strings

```
string stringEingabe = Console.ReadLine();  
int intEingabe = int.Parse(Console.ReadLine());  
double doubleEingabe = double.Parse(Console.ReadLine());
```

- Einlesen von einzelнем Zeichen (Character)

```
Console.ReadKey();  
Console.WriteLine("Zum Beenden beliebige Taste drücken: ");
```

Konvertierung von Datentypen (Casting)

- Zahl => String

- Implizit:

```
string satz = "Wert der Zahl: " + zahl;
```

- Explizit:

```
string satz = zahl.ToString();
```

- String => Zahl

- Nur per Hilfsfunktion möglich:

```
int zahl = int.Parse(Console.ReadLine());
```

```
double zahl2 = double.Parse(Console.ReadLine());
```

- Zahl => Zahl

```
int ganzzahl = 25;
```

```
double kommazahl = ganzzahl;
```

```
ganzzahl = (int)kommazahl;
```

Mathematische Operatoren und Funktionen

- $a + b$ Plus
- $a - b$ Minus
- $a * b$ Multiplikation
- a / b Division
- $a \% b$ Modulo (Rest der Division)
- $a++/a--$ $a = a + 1$ / $a = a - 1$
- $a += 2$ $a = a + 2$
- $b /= 2$ $b = b / 2$
- `Math.Round(2.6)` Gerundete Zahl, bei .6 runde auf nächste gerade Zahl
- `Math.Max(a,b)` Ermittle Maximum von beiden Zahlen
- `Math.Min(a,b)` Ermittle Minimum von beiden Zahlen

Fragen

- Was ist die Aufgabe der Garbage Collection?
- Welchen Datentypen kann man für Zeichenketten (Text) verwenden?
- Eine Variable wurde mit `int x = 2;` initialisiert, welchen Wert hat sie nach folgenden Operationen:
 - `x+=2;`
 - `x++;`
 - `x % 2`
- Welcher Datentyp hat das Ergebnis der Division eines Integer-Werts mit einem Double-Wert?
- Mit welcher Funktion kann man einen String in einen Double konvertieren?

Kontrollstrukturen

Logische Operatoren

- Jede Operation ergibt entweder true oder false
- $a == b$ a gleich b
- $a != b$ a ungleich b
- $a < b$ a kleiner b
- $a > b$ a größer b
- $a <= b$ a kleiner oder gleich b
- $a \&\& b$ a und b sind wahr
- $a || b$ a oder b ist wahr
- $a \wedge b$ entweder a oder b
- $!a$ true wird zu false, false zu true

Bedingungen

```
if (Bedingung A)
{
    //Anweisungen wenn Bedingung A wahr
}
else if (Bedingung B)
{
    //Anweisungen wenn Bedingung A falsch und B wahr
}
else
{
    //Anweisungen wenn Bedingung A und B falsch sind
}
```

Zahlenratespiel (Teil 1)

- Zufallszahl ermitteln

```
//Initialisiere Zufallsgenerator  
Random generator = new Random();  
//Erzeuge Zahl zwischen 1 und 5  
int zufallszahl = generator.Next(1, 5);
```

- Erzeuge eine Zufallszahl zwischen 1 und 5
- Prüfe Nutzereingabe und gebe aus ob seine Zahl größer, gleich oder kleiner der Zufallszahl ist

Übung

Schleifen

Kopfgesteuert

while (Bedingung)

{

 //wiederholt solange bis Bedingung falsch ist

}

Fußgesteuert

do

{

 //führt mindestens 1 Mal aus

} **while** (Bedingung);

Zahlenratespiel (Teil 2)

- Der Nutzer soll solange neue Zahlen eingeben können bis er die richtige Zahl erraten hat
- Am Ende des Spiels die Anzahl der Versuche ausgeben

Übung

Arrays

Aufzählung von gleichartigen Datentypen

```
int[] zahlen = new int[] { 2, 4, 5 };  
string[] buchstaben = new string[] { "A", "B", "C" };  
double[] kommazahlen = new double[10];
```

Zugriff über Index

```
zahlen[2]; //5  
buchstaben[1]; //"B"  
zahlen[2] = 4; //setzt 3. Element im Array auf Wert 4
```

Zähl-Schleifen

`for(Initialisierung;Bedingung;Anweisung)`

Hochzählen(Inkrementierung):

```
for (int i = 0; i<10; i++)  
{  
    //Zähle von 0 bis 9  
}
```

Runterzählen(Dekrementierung):

```
for (int i = 10; i>0; i--)
```


Array durchlaufen

```
for(int i=0;i<zahlen.Length;i++)  
{  
    Console.WriteLine(zahlen[i]);  
}
```

Oder

```
foreach(var item in zahlen)  
{  
    Console.WriteLine(item);  
}
```

Nützliche Funktionen und Eigenschaften von Arrays

```
int[] numbers = new int[] { 2, 9, 5 };
```

Funktionsname	Beschreibung	Beispiel
<code>Contains(int value)</code>	Prüft ob Array ein Element enthält	<code>numbers.Contains(5); // true</code>
<code>Max()</code>	Gibt den größten Wert im Array zurück	<code>numbers.Max(); //9</code>
<code>Min()</code>	Gibt den niedrigsten Wert im Array zurück	<code>numbers.Min(); //2</code>
<code>Length</code>	Gibt die Anzahl der Elemente im Array zurück	<code>numbers.Length; //3</code>
<code>First()</code>	Gibt das erste Element im Array zurück	<code>numbers.First(); //2</code>
<code>Last()</code>	Gibt das letzte Element im Array zurück	<code>Numbers.Last(); //5</code>
<code>Sum()</code>	Gibt die Summe aller Elemente zurück	<code>Numbers.Sum(); //16</code>

Fragen

- Bool a = true und bool b = false, welchen Wahrheitswert ergeben folgende Ausdrücke:
 - a == b
 - a || b
 - !(a && b)
 - !b
 - a ^ b
- Welche der folgenden Array-Initialisierungen ist syntaktisch falsch?
 - `int[] a = new int[20];`
 - `Int[] b = new int[] {2, 5, 8}`
 - `Int[] a = new int[];`
- Wie oft wird eine do-while-Schleife mindestens durchlaufen?
- Was ist der Unterschied zwischen break und continue in einer Schleife?

Enumeratoren

- Eigenen Datentyp definieren, der nur ganz bestimmte Werte zulässt

```
enum Wochentag { Mo=1, Di=2, Mi, Do, Fr, Sa, So};
```

- Vorteil: bessere Lesbarkeit des Codes

```
Wochentag tag = Wochentag.Do;  
if (tag == Wochentag.Mi) {  
    //...  
}
```

Switch

```
switch (tag)
{
    case Wochentag.Mo:
        Anweisung A;
        break;
    case Wochentag.Di:
    case Wochentag.Mi:
        Anweisung B;
        break;
    default:
        Anweisung C;
        break;
}
```

Funktionen

```
public static int addiere(int a, int b)
{
    int c = a + b;
    return c;
}
```

Modifier Rückgabewert Name(Parameterliste)

Aufruf

```
int summe = addiere(2, 4);
```

Spezielle Parametertypen: out/ref

- Normalerweise werden Werttypen wie int, bool, string oder double in Funktionen als Wert übergeben
- Mittels out/ref können sie als Referenz übergeben werden

```
public void changeAlter(out int wert)
{
    wert = 6;
}
```

```
//Aufruf
int alter;
changeAlter(out alter); //alter ist jetzt 6
```

- Out behandelt die Variable so als hätte sie noch keinen Wert (nicht initialisiert)
- Bei Ref muss eine bereits initialisierte Variable übergeben werden

Spezielle Parametertypen: params

- ermöglicht variable Anzahl von Parametern von einem Typ

```
public static int BildeSumme(params int[] summanden)
{
    int summe = 0;
    foreach (var item in summanden)
    {
        summe += item;
    }
    return summe;
}
```

```
BildeSumme(2, 3, 4);           //ergibt 9
BildeSumme();                  //ergibt 0
BildeSumme(new int[]{3, 3, 4}); //ergibt 10
```


Spezielle Parametertypen: optional

- Parameter können schon in der Deklaration mit Defaultwerten initialisiert werden, beim Aufruf ist der Parameter dann optional

```
public string StringCombine(string[] strings, char trenner = '/')
{
    string resultString = strings[0];
    for(int i = 1; i<strings.Length; i++)
    {
        resultString += (trenner + strings[i]);
    }
    return resultString;
}
```

```
StringCombine(new string[] { "A", "B", "C" });           // "A/B/C"
StringCombine(new string[] { "A", "B", "C" }, '-');      // "A-B-C"
```

Fragen

- Was stimmt nicht an folgenden Funktionsdeklaration?
 - `void funktion1(int x, string y = "abc", int z)`
 - `void funktion2(params int x)`
 - `void funktion3(int x, out y, int x, out y)`
- Warum sollte man statt fester String-Werte lieber Enumeratoren verwenden, um Zustände des Programms zu verwalten?

Objektorientierte Programmierung

- Vorteile:
 - Code wiederverwendbar
 - Bessere Strukturierung und Lesbarkeit
 - Sicherheit (public/private)
- Klassen: beschreiben Struktur von Objekten
- Aus einer Klasse beliebig viele Objekte erzeugbar:
 - `Random random1 = new Random();`
 - `Random random2 = new Random();`
- Klassen bestehen aus
 - Feldern (Membervariablen)
 - Eigenschaften (Properties)
 - Funktionen (Methoden)
 - Konstruktor/Destruktor

Felder und Methoden

```
public class Person
{
    //Felder
    private string vorname;
    private string nachname;
    //Methoden
    public void SetVorname(string vorname)
    {
        this.vorname = vorname;
    }
    public string GetVorname()
    {
        return this.vorname;
    }
}
```

Properties

- Kurzschreibweise um Zugriff auf private Membervariablen über Methoden zu definieren

```
public class Person
{
    //Properties
    public string Vorname { get; private set; }
    public string Nachname { get; private set; }
    //Konstruktor
    public Person(string vorname, string nachname)
    {
        this.Vorname = vorname;
        this.Nachname = nachname;
    }
}
```

Varianten von Properties

```
//öffentliches Lesen und Schreiben erlauben  
public string Vorname { get; set; }
```

```
//Variable kann nur noch intern gesetzt werden  
public string Nachname { get; private set; }
```

```
//Variable kann von Außen nur überschrieben werden  
public string Geheim { private get; set; }
```

Zugriff auf Property genau steuern

```
private int alter; //Membervariable
public int Alter //dazugehörige Property
{
    get
    {
        return alter;
    }
    set
    {
        if(value > 0)
        {
            alter = value;
        }
    }
}
```

Modifizier

- Können für Properties, Klassen und Methoden verwendet werden

Modifizier	Zugriff
<code>public</code>	Von Außerhalb der Klasse
<code>private</code>	Nur innerhalb der Klasse
<code>protected</code>	Innerhalb der Klasse und in allen abgeleiteten Klassen
<code>internal</code>	Zugriff nur innerhalb der aktuellen Assembly/des selben Namespaces

Konstruktor/Destruktoren

- Legt den Startzustand des Objekts nach Initialisierung fest
- Mehrere Konstruktoren pro Klasse möglich (verschiedene Parameter)

```
public Person(string vorname, string nachname)
{
    this.Vorname = vorname;
    this.Nachname = nachname;
}
```

- Destruktor wird bei Zerstörung des Objektes aufgerufen

```
~Person()
{
    Console.WriteLine("Destruktor");
}
```

Statische Member (Methoden/Eigenschaften)

- Als statisch markierte Methoden oder Eigenschaften gelten für die Klasse selbst und nicht für Objekte der Klasse
- Der Zugriff erfolgt über den Klassennamen
- Jede Eigenschaft existiert nur 1 Mal für die jeweilige Klasse
- Innerhalb von statischen Methoden darf nicht auf nicht-statische Methoden/Eigenschaften der Klasse zugegriffen werden
- WriteLine() und ReadLine() sind statische Methoden der Klasse Console, um auf sie zuzugreifen muss zuvor keine Instanz der Klasse Console erzeugt werden

Fragen

- Was ist der Unterschied zwischen private und protected?
- Welche Zugriffsmöglichkeiten innerhalb und außerhalb der Klasse bieten die folgenden Eigenschaften:

- `public int Property1 { get; set; }`
- `public int Property2 { get; private set; }`
- `private int Property3 { get; set; }`
- `public int Property4 { get; }`

- Woran erkennt man eine Konstruktor-Deklaration?

Vererbung von Klassen

- Alle Eigenschaften und Methoden von Basisklasse übernehmen

```
class CustomRandom : Random
{
    public int NextInclusive(int min, int max)
    {
        return base.Next(min, max+1);
    }
}
```

Konstrukturen in vererbten Klassen

```
public class Lebewesen
```

```
{
```

```
    public int Alter { get; set; }
```

```
    public Lebewesen(int alter)
```

```
    {
```

```
        this.Alter = alter;
```

```
    }
```

```
}
```

```
public class Mensch : Lebewesen
```

```
{
```

```
    public string Wohnort { get; set; }
```

```
    public Mensch(int alter, string wohnort) : base(alter)
```

```
    {
```

```
        this.Wohnort = wohnort;
```

```
    }
```

```
}
```

Ruft zuerst Basiskonstruktor auf und
danach Konstruktor der Kindklasse

Virtuelle Methoden

```
public class Fahrzeug
{
    public virtual string BeschreibeMich() {
        return "Ich bin ein Fahrzeug";
    }
}
public class Auto : Fahrzeug
{
    public override string BeschreibeMich()
    {
        return base.BeschreibeMich() + " habe 4 Räder";
    }
}
```

Abstrakte Methoden und Klassen

```
abstract class Fahrzeug
{
    public abstract bool IstFahrbereit();
}
```

```
public class Fahrrad : Fahrzeug
{
    public int Luft { get; set; }
    public override bool IstFahrbereit()
    {
        return (Luft > 0);
    }
}
```

Fragen

- Von wie vielen Klassen kann eine Klasse maximal erben?
- Was ist der Unterschied zwischen abstrakten und virtuellen Methoden?

Interfaces

- Geben Schnittstellen für Methoden und Eigenschaften vor ohne diese zu implementieren
- Unterschied zu abstrakten Klassen: dort können einzelne Methoden bereits implementiert sein
- Eine Klasse kann beliebig viele Interfaces gleichzeitig implementieren

```
interface IBewegbar
{
    int Räderanzahl { get; set; }

    void Bewegen();
}
```

Interface implementieren

```
class Fahrrad : Fahrzeug, IBewegbar
{
    //explizite Schnittstellenimplementierung
    void IBewegbar.Bewegen()
    {
        Console.WriteLine("Bewegen Explizit");
    }
}

class Fahrrad : Fahrzeug, IBewegbar
{
    //normale Implementierung
    public void Bewegen()
    {
        Console.WriteLine("Bewegen Implizit");
    }
}
```

Polymorphismus

- Objekte einer bestimmten Klasse können entweder als ihr eigener Typ, als beliebiger Basistyp (Klasse von der sie abgeleitet sind) oder als Typ einer Schnittstelle, die sie implementieren, verwendet werden

```
class Fahrrad : Fahrzeug, IBewegbar { ... }
```

```
Fahrrad fahrrad = new Fahrrad();
```

```
Fahrzeug fahrzeug = fahrrad;
```

```
IBewegbar bewegbar = fahrrad;
```

```
Variablentyp bezeichner = new Laufzeittyp();
```

Typ eines Objektes prüfen

- GetType() ermittelt den Typ des Objektes, auf den eine Variable referenziert

```
Fahrzeug fahrzeug = new Fahrrad();  
if(fahrzeug.GetType() == typeof(Fahrrad)) // true  
if(fahrzeug.GetType() == typeof(Fahrzeug)) // false
```

- is prüft ob ein Objekt Typ einer bestimmten Klasse ist, ein Interface implementiert oder Kind einer Basisklasse ist (fahrzeug steht für Fahrzeug-Objekt und Fahrrad für Fahrrad-Objekt)

```
if(fahrzeug is Fahrrad) // false  
if(fahrzeug is Fahrzeug) // true  
if(fahrrad is IBewegbar) // true  
if(fahrrad is Fahrzeug) // true  
if(fahrrad is Object) // true
```

Fragen

- Wie viele Interfaces kann eine Klasse implementieren?
- Von welcher Klasse erben alle anderen Klassen automatisch mit?
- Welche der Folgenden Bedingungen ergibt true?

```
class Fahrzeug {...}  
class Fahrrad : Fahrzeug{...}  
Fahrzeug fz = new Fahrzeug();  
Fahrzeug fr = new Fahrrad();  
if (fz is Fahrzeug){...}  
if (fz is Fahrrad){...}  
if (fr is Fahrzeug){...}  
if (fr.GetType() == typeof(Fahrzeug)){...}
```

Generische Datentypen

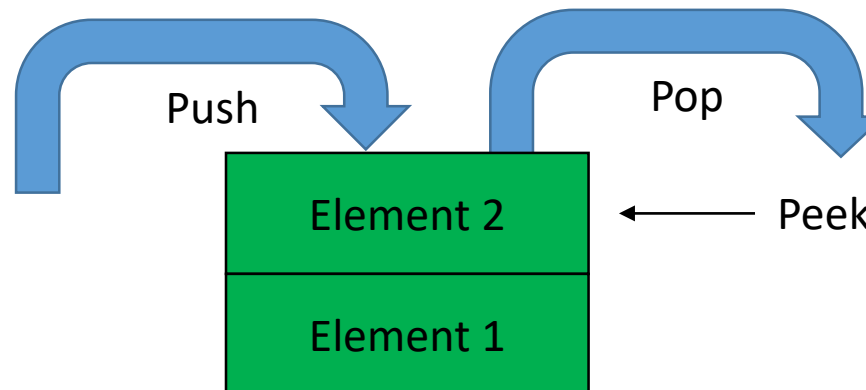
```
List<string> StringListe = new List<string>();  
StringListe.Add("1. Eintrag");  
foreach(var item in StringListe)  
{  
    Console.WriteLine(item);  
}
```

T steht für einen beliebigen Datentypen

Spezielle Arten von Listen: Stack

- LIFO: Last In First Out

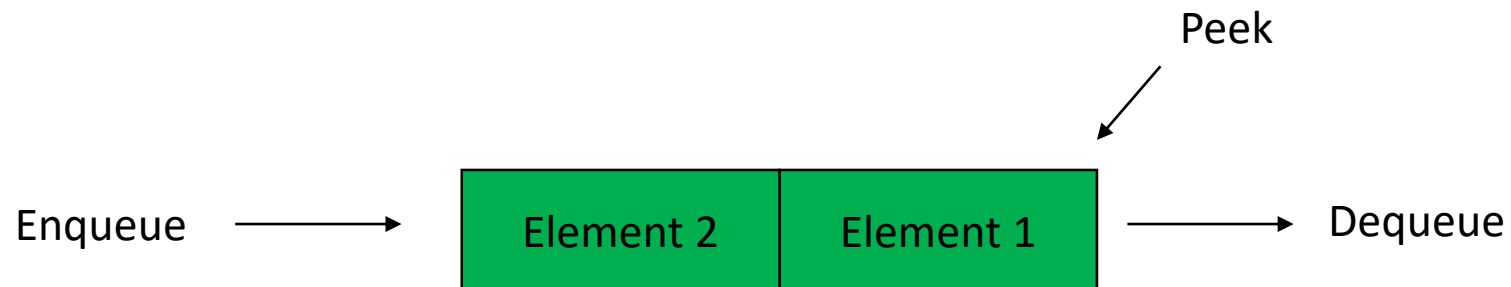
```
Stack<string> stringStack = new Stack<string>();  
stringStack.Push("Element 1");  
stringStack.Push("Element 2");  
stringStack.Peek() //=> "Element 2"  
stringStack.Pop()  //=> Element 2 herunternehmen
```



Spezielle Arten von Listen: Queue

- FIFO: First In First Out

```
Queue<string> stringQueue = new Queue<string>();  
stringQueue.Enqueue("Element 1");  
stringQueue.Enqueue("Element 2");  
stringQueue.Peek()    //=> "Element 1"  
stringQueue.Dequeue() //=> Element 1 herausnehmen
```



Spezielle Arten von Listen: Dictionary

- Ordnet einem Wert (**Key**) einen anderen Wert zu (**Value**)

```
var stringDictionary = new Dictionary<int, string>();  
stringDictionary.Add(5, "Element 1");  
stringDictionary.Add(10, "Element 2");  
stringDictionary[10]; // "Element 2"  
stringDictionary.ContainsKey(5) //true  
stringDictionary.ContainsValue("Element 3") //false
```

Key vom Typ Integer	Value vom Typ String
5	„Element 1“
10	„Element 2“

Spezielle Arten von Listen: Hashtable

- Ähnlich wie Dictionary, aber Key und Value können einen beliebigen Wertetyp oder Referenztypen haben (List, Stack, eigene Klasse)
- Beim Versuch auf einen nicht-existent Index zuzugreifen, gibt es keine Fehlermeldung

```
Hashtable ht = new Hashtable();  
ht.Add("key1", "value1");  
ht.Add(20, 100);  
ht.Add(10.5, new DateTime(2012, 1, 1));  
  
if(ht[10.5]?.GetType() == typeof(DateTime)) {  
    Console.WriteLine("Jahr: " + ((DateTime)ht[10.5]).Year);  
}
```

Kopie eines Objektes erstellen: Kopierkonstruktor

Objekte werden standardmäßig immer per Referenz zugewiesen

```
Person person1 = new Person("Alex");
```

```
Person person2 = person1; //beide zeigen auf das gleiche Objekt im Speicher
```

Kopierkonstruktor definieren:

```
public Person(Person person)
{
    this.Name = person.Name;
}
```

```
Person person2 = new Person(person1);
```

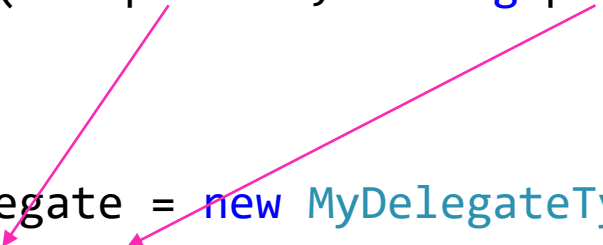
Kopie eines Objektes erstellen: Clone-Funktion

```
public class Person : ICloneable
{
    public string Name { get; set; }
    public Person Ehepartner { get; set; }
    public object Clone()
    {
        Person newPerson = (Person) this.MemberwiseClone();
        newPerson.Ehepartner = (Person) this.Ehepartner?.Clone();
        return newPerson;
    }
}
```

Delegates

- Delegate-Variablen speichern Referenzen auf andere Methoden
- Referenzen können zur Laufzeit hinzugefügt/entfernt werden
- Referenzierte Methoden müssen gleiche Signatur haben wie der Delegate-Typ

```
public delegate int MyDelegateTyp(int param1, string param2);  
public int Beispiel(int params1, string param2) {  
    return 2;  
}  
MyDelegateTyp myDelegate = new MyDelegateTyp(Beispiel);  
myDelegate.Invoke(5, "Test"); //Rückgabewert: 2
```



Vordefinierte Delegate-Typen

Name	Rückgabotyp	Parameter	Beispiel für referenzierbare Methode
Action<int>	Void	1 Integer	<pre>public void myAction(int p1) { Console.Write(p1); }</pre>
Predicate<int>	Bool	1 Integer	<pre>public bool myPredicate(int p1) { return true; }</pre>
Func<int, int, string>	String	2 Integer	<pre>public string myFunc(int p1, int p2) { return (p1 + p2).ToString(); }</pre>
Func<string>	String	Keine	<pre>public string myFunc() { return "Test"; }</pre>

Anonyme Methoden

- Anonym bedeutet: Methode ohne Namen
- Delegate-Variablen Methoden zuweisen, ohne diese vorher zu deklarieren

```
Func<int, int> myDelegate = new MyDelegates(  
    delegate(int p1) {  
        return 5;  
    }  
);
```

- Kurzschreibweise:

```
Func<int, int> myDelegate = delegate(int p1) {  
    return 5;  
};
```

Timer verwenden

```
using System.Timers;

Timer timer = new Timer();
timer.Elapsed += new ElapsedEventHandler(Countdown);
timer.Interval = 1000;
Timer.Start();
```

```
void Countdown(Object sender, ElapsedEventArgs args)
{
    countdownNumber--;
    Console.WriteLine(countdownNumber);
    if(countdownNumber <= 0)
    {
        timer.Stop();
    }
}
```


Fragen

- Welche Methode passt zu welchem Delegate-Typen?

```
void methode1(int x)
```

```
int methode2(string y)
```

```
void methode3(bool x, bool y)
```

```
string methode4(List<string> x)
```

```
void methode5()
```

```
Func<List<string>, string> delegate1;
```

```
Action<bool, bool> delegate2;
```

```
Func<string, int> delegate3;
```

```
Action delegate4;
```

```
Action<int>
```

Exception-Handling

- Mittels try/catch Fehlerfälle in Anwendung abfangen:

```
try
{ int Eingabe = int.Parse(Console.ReadLine());}
catch(FormatException e)
{
    Console.WriteLine("Bitte eine Zahl eingeben.");
    return;
}
catch(Exception e)
{
    Console.WriteLine(e.Message);
    return;
}
finally
{
    Console.WriteLine("Danke");
}
```

Eigene Exception-Klasse mit Fehlermeldung

```
public class MyException : Exception
{
    public MyException(string message) : base(message)
    {
    }
}
```

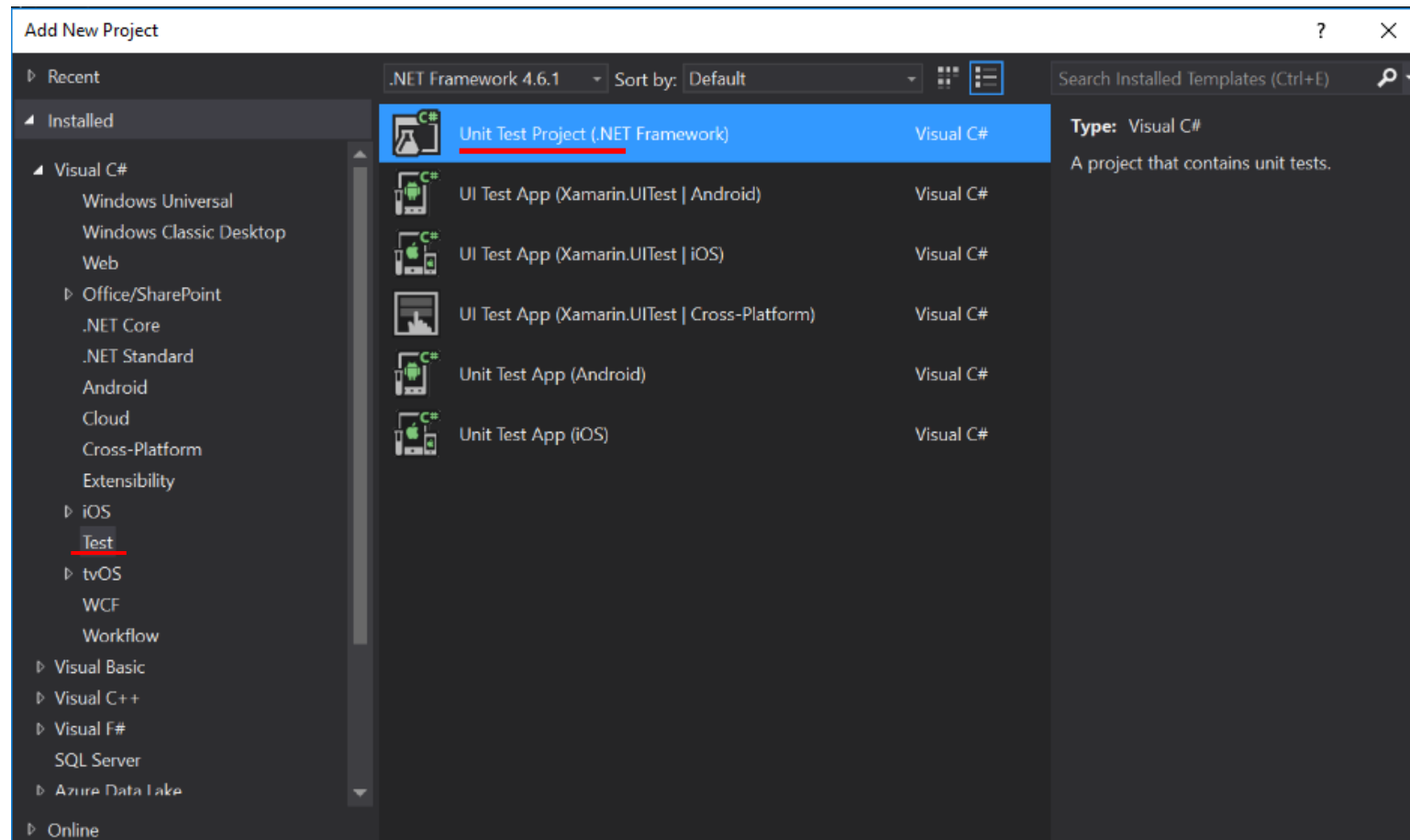
- Fehler werfen:

```
throw new MyException("Errormessage");
```

- Fehler abfangen:

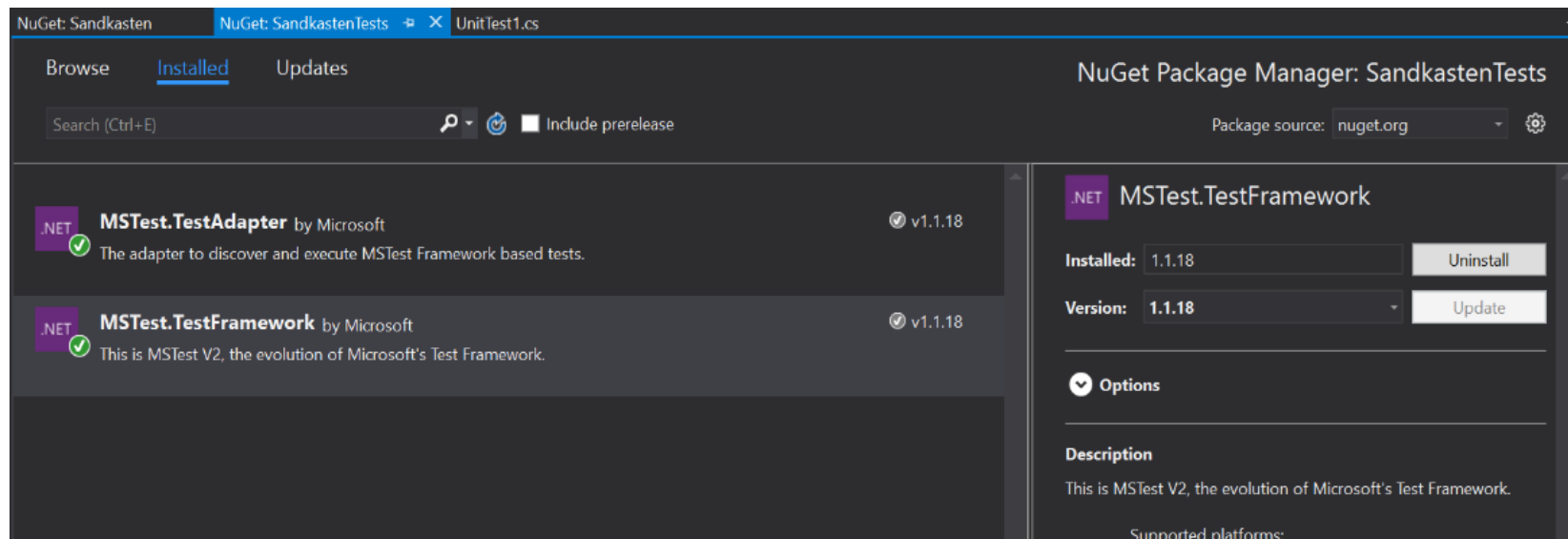
```
catch (MyException e)
{
    Console.WriteLine(e.Message);
}
```

UnitTests - Testprojekt anlegen



UnitTests – NuGet Packete installieren

- MSTest.TestFramework und MSTest.TestAdapter über NuGet Package Manager für das Projekt installieren



- Schließlich noch das zu testende Projekt unter References hinzufügen

UnitTests – Testklasse schreiben

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        KlasseAusProjekt objekt = new KlasseAusProjekt();
        int result = objekt.ZuTestendeMethode(4);
        Assert.IsNotNull(result);
        Assert.AreEqual(2, result);
    }
}
```

- Test laufen lassen über Test->Run->All Tests

Fragen

- Wozu benötigt man den Finally-Block in einem try-catch-Block?
- Was passiert wenn man innerhalb eines catch-Blockes erneut eine Exception wirft?

Dateien schreiben/lesen

```
using System.IO;
```

```
File.WriteAllText("Testdatei.txt", "Hier folgt ein Content");
```

```
File.ReadAllText("Testdatei.txt");
```


Dateidialog anzeigen

- Dialog zum Datei öffnen bzw. speichern
- `using` Microsoft.Win32;

```
SaveFileDialog saveDialog = new SaveFileDialog();  
saveDialog.FileName = "Personen.pl";  
saveDialog.Filter = "Personenliste|*.pl|Textdokument|*.txt|Alle|*.*";
```

```
if(saveDialog.ShowDialog() == true){...}
```

- `FileName`: ausgewählte (bzw. vorbelegter) Dateiname
- `Filter`: welche Dateitypen sollen auswählbar sein
- zum Datei Öffnen analog die Klasse OpenFileDialog verwenden

Objekte als JSON serialisieren

- Serialisierung: Beliebiges Objekt in einen String konvertieren
- Vorteil: Der Zustand eines Objektes kann als String in eine Datei geschrieben und später wieder ausgelesen und zurückkonvertiert (deserialisiert) werden

```
using Newtonsoft.Json; //vorher per NuGet installieren
var daten = new List<string> { „Wert1“, „Wert2“, „Wert3“ };
//.NET-Objekt in JSON-String konvertieren
string result = JsonConvert.SerializeObject(daten);
sw.Write(result);
//JSON-String in .NET-Objekt umwandeln
objekt = JsonConvert.DeserializeObject<List<string>>(sr.ReadToEnd());
```

Fragen

- Warum muss man beim Deserialisieren das Format, in das deserialisiert werden soll, mit angeben aber beim Serialisieren nicht?
- Wie kann man beim OpenFileDialog die Auswahl auf Text-Dateien einschränken?

Nützliche Links

- <http://blog.ppedv.de/>
- <https://studios.ppedv.de/>
- <https://stackoverflow.com/>
 - Für Fragen aller Art
- http://openbook.rheinwerk-verlag.de/visual_csharp_2012/1997_01_001.html#dodtpf471901b-c92c-4256-b4a7-6c4233f40e97
 - Sehr gutes Online-Lehrbuch von Rheinwerk zu Themen rund um C# und .NET
- <https://msdn.microsoft.com/de-de/library>
 - Offizielle Referenz zu allen .NET Klassen von Microsoft
- <https://www.dotnetperls.com/>
 - Gute Sammlung von Tutorials
- <http://www.entityframeworktutorial.net/>
 - Entity Framework Tutorial