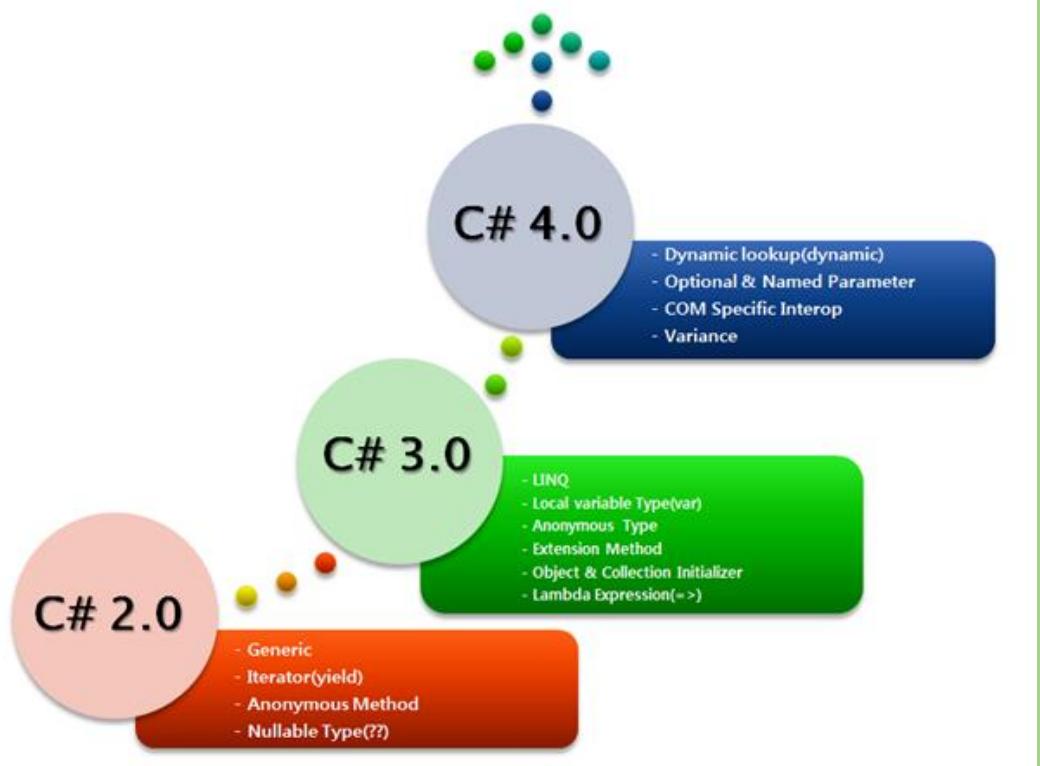




2017

C#-Grundlagen



Karl Steinam

Klara-Oppenheimer-Schule

23.9.2017

Inhalt

Compiler- / Interpretersprachen	5
.NET-Framework und Betriebssystem	7
Aufgaben	8
Sprachgrundlagen	9
Bezeichner	9
Reservierte Worte	9
Anweisungen/Anweisungsblöcke/Ausdruck	10
Variable/Konstante/Datentyp im Überblick	10
Aufgaben	11
Datentypen	15
Wert- und Verweistypen	15
Numerische Datentypen	16
Zeichen-Datentypen	17
Datentyp DateTime	18
Boolesche Datentypen	18
Typumwandlungen (Casten)	18
Operatoren	20
Arithmetische Operatoren	21
Logische Operatoren	24
Verschiebeoperatoren	26
Vergleichsoperatoren	26
Verkettungsoperator	27
Zuweisungsoperator	28
Inkrement- und Dekrementoperatoren	29
Operatorenvorrang	29
Fragen	30
Kontrollstrukturen	37
Anweisung	37
Einseitige/Zweiseitige Auswahl	38
Mehrseitige Auswahl	41
Wiederholung	43
Aufgaben	49
Weitergehende Aufgaben	54
Methoden und Funktionen	56

Grundlagen	56
Rückgabewert einer Methode.....	58
Anwendung Umsatzsteuerberechnung.....	58
Verbesserung der Lösung.....	58
Parameter einer Methode.....	60
Nebeneffekte bei Methodenaufrufen.....	60
Übergabe als Kopie (by val).....	61
Übergabe als Referenz (by ref).....	61
Überladen von Methoden.....	61
Parameter-Variationen: ref / out	63
Auswertungsreihenfolge bei Methoden	66
Sichtbarkeit von Methoden.....	68
Zusammenfassung Methode.....	68
Aufgaben	69
Komplexe Datentypen.....	74
Arrays.....	74
Zugriff auf ArrayElemente	76
Wichtige Methoden/Eigenschaften im Zusammenhang mit Arrays	77
ArrayLists.....	78
Hashtables	79
Arbeit mit Listen – (Durchlaufen_Sortieren_Suchen)	81
Die Schnittstelle »IEnumarable«	81
Die Schnittstelle IList	82
Queues	87
Stacks.....	87
Aufgaben	88
Arrays	88
Aufgaben zu ArrayLists.....	93
Aufgaben zu Hashtable	94
Aufgaben zu Stacks	95
Weiterführende Aufgaben	96
Weitergehende Fragen	101
Objektorientierte Programmierung	102
Grundlagen	103
Enumerations	103
Structures	106

Klassen und Instanzen	107
Felder/Attribute	108
Methoden	108
Properties	109
Konstruktoren und Destruktoren	110
Statische Member	111
Sichtbarkeit von Klassen und deren Membern	112
Die finalize-Methode	113
Aufgaben	114
Weitergehende Aufgaben	118
Vererbung	120
Einfache Vererbung	120
Mehrfachvererbung	129
Abstrakte Klasse	130
Interface	133
Polymorphie / Late vs. Early Binding	134
Beziehungen	136
Assoziation	136
Multiplizität	136
Navigierbarkeit	137
Rollen	137
Aggregation/Komposition	139
Aufgabe	140
Statements und Exceptions	143
Konzept	144
Implementierung	145
Weiterreichen/Erzeugen von Exceptions per throw	148
Exception-Hierarchie	150
Welche Informationen sollten erfasst werden	152
Aufgaben	154
Weitergehende Aufgaben	159
Referate	159
Softwareentwicklung	160
Lernziele	160
Unterricht	160
Code And Fix	160

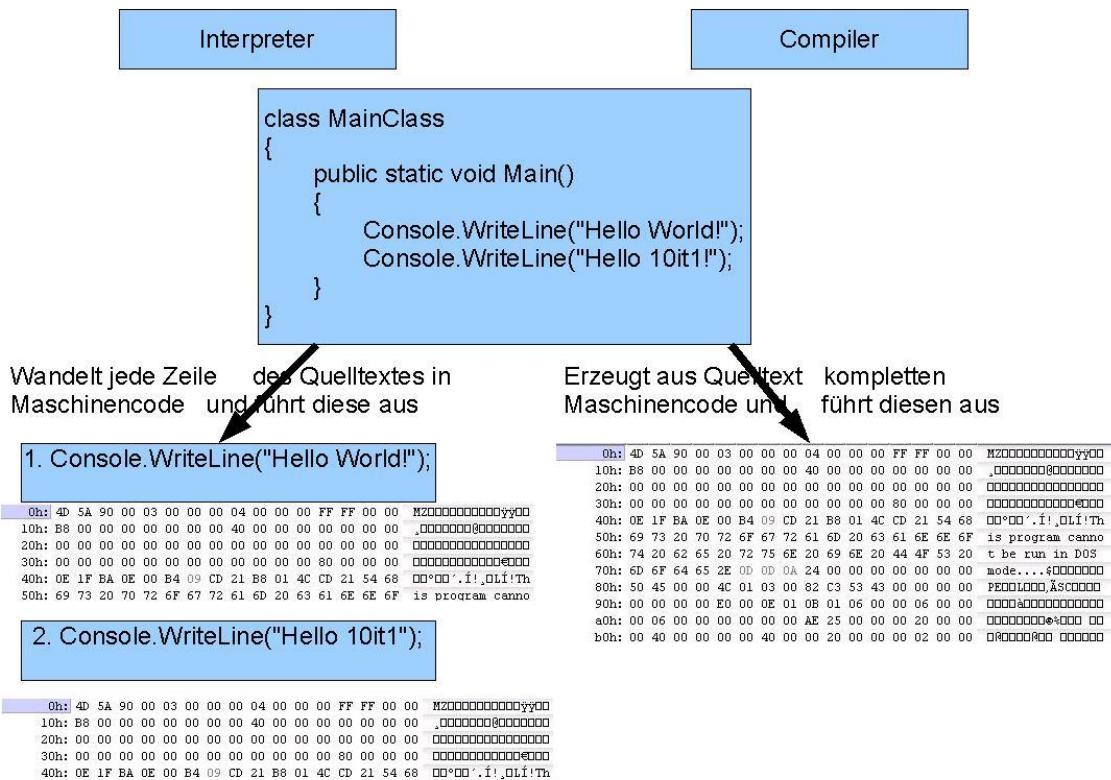
Wasserfallmodell.....	161
V-Modell	162
Prototypmodell	163
Spiralmodell	166
Unified Process	167
XP (Extreme Programming)	168
CMM - Capability Maturity Model	168
Aufgaben.....	170



Compiler- / Interpretersprachen

Üblicherweise werden Programme in höheren Programmiersprachen geschrieben und mit Hilfe von **Interpretern** oder **Compilern** in die Maschinensprache der jeweiligen Zielplattform übersetzt.

Compiler und Interpreter haben dabei unterschiedliche Funktionsprinzipien

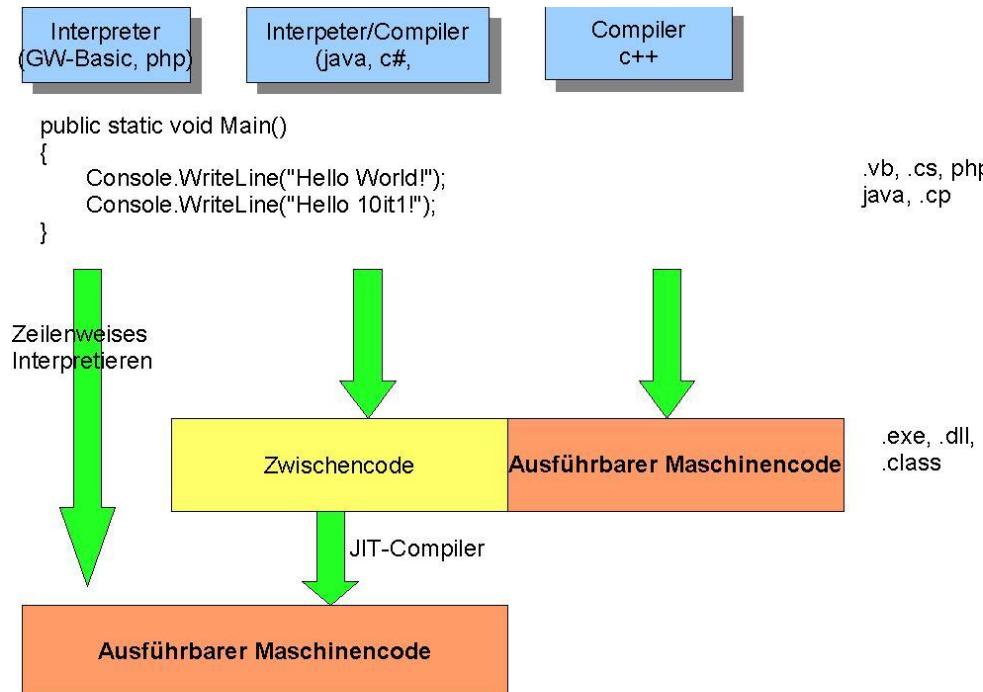


Interpreter	Compiler
Vorteil	Nachteil
Einfache Fehlersuche, da zeilenweises Arbeiten möglich	Hohe Geschwindigkeit des Codes
Vergleichsweise langsam	Zum Testen muss Programm immer compiliert werden (langwierig)

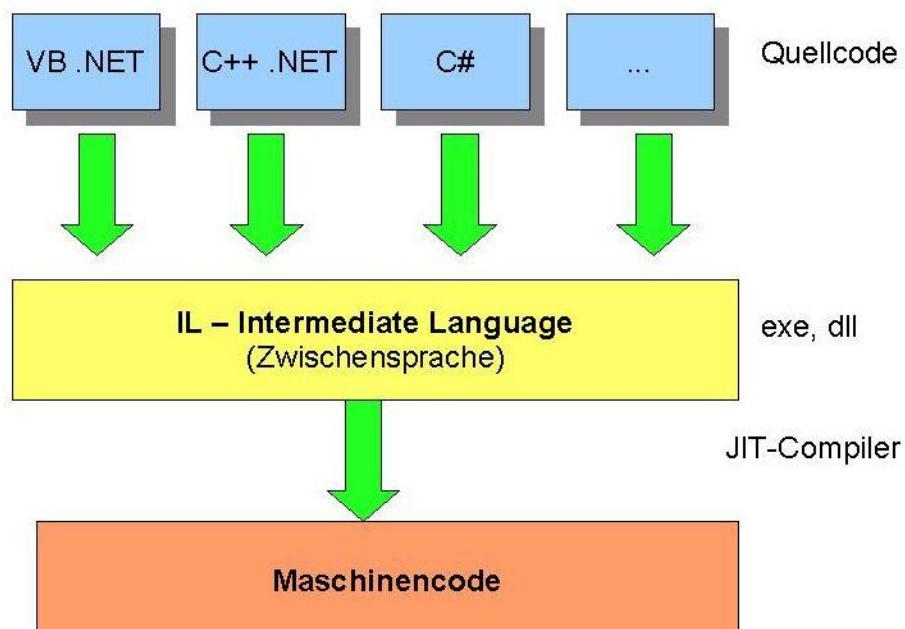
Vor- und Nachteile von Interpreter- und Compilersprachen

Ein Nachteil dieser Programmentwicklung ist, dass kaum eine Programmentwicklung möglich ist, die sowohl betriebssystem- als auch plattformübergreifend funktioniert.

Aus diesem Grunde gewann die Idee, eine Bytecode zu erzeugen, der erst auf einer jeweiligen Zielplattform von einer virtuellen Maschine interpretiert wird, immer mehr an Bedeutung (Java).



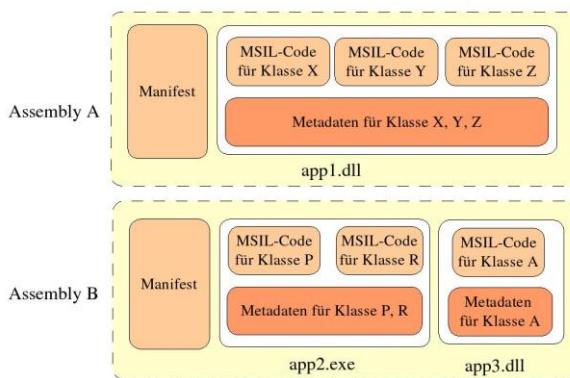
Neben dieser **Plattformunabhängigkeit** wird durch die .NET-Technologie auch eine sprachübergreifende Entwicklung möglich.



.NET-Framework und Betriebssystem

.NET bedeutet nicht nur Programmiersprachen, sondern es ist eine umfangreiche Sammlung von Klassen (.NET 2.0 = ca. 8000 Klassen), die den Programmierer in seiner täglichen Arbeit unterstützen.

- Umfangreiche Klassenbibliothek
Laufzeitumgebung (CLR - Common Language RunTime)
- Einheitliche Zwischensprachen
Quellcode jeder Programmiersprache wird mit Hilfe entsprechender Compiler in eine einheitliche Zwischensprache (MSIL) portiert. MSIL-Code ist sprach- und plattformunabhängig. Damit die Programmiersprachen diesen **gemeinsamen Nenner** generieren können, müssen sie bestimmten Richtlinien entsprechen (CLS - Common Language Specification, CTS - Common Type System)
- Zwischencode wird durch einen JIT (JustInTime)-Compiler in den Maschinencode des entsprechenden Prozessors übersetzt. Dabei wird nur der Programmteil übersetzt, der zur Zeit wirklich gebraucht wird (Geschwindigkeit). Bereits übersetzter Code wird wieder verwendet.
- Laufzeitumgebung übernimmt Sicherheits- und Versionsprüfung sowie Fehler- und Speicherverwaltung (**Managed Code** vs. **Unmanaged Code**)
- .NET-Anwendungen == Assemblies



Sobald eine Quellcodedatei kompiliert wurde, gehört es zu einer sog. **Assembly**. Sie ist ein Container für ein oder mehrere Module. Sie enthält neben dem Programmcode die Metadaten und das Manifest zur Beschreibung der gesamten Assembly.

- Assembly <> Registry/DLL-Hölle

Die Assembly-Technik macht das Registrieren von Komponenten und Versionsnummern in der Registry überflüssig. Es können nebeneinander verschiedene Versionen einer Datei existieren, ohne dass es zur sog. DLL-Hölle kommt.

- Siehe dazu: <http://msdn.microsoft.com/de-de/library/bb979301.aspx> und <http://msdn.microsoft.com/de-de/library/cc405537.aspx>

Aufgaben

- Erläutern Sie den Begriff DLL-Hölle an einem anschaulichen Beispiel
- Was ist der Unterschied zwischen einem JIT-Compiler und einem AOT-Compiler
- Warum heißt der Java-Compiler **Hotspot**-Compiler
- Compiliert ein Compiler immer den kompletten Quellcode ?
- Kann ein kompiliertes Java-Programm schneller sein als ein kompiliertes C++ - Programm.
- Wie muss man sich vorstellen, dass zur Laufzeit nur die Methoden kompiliert werden, die gerade benötigt werden ? Was passiert mit den nicht benötigten Methoden ?
- Welche Vorteile haben dynamische Kompilierung im Vergleich zu statischer Kompilierung
- Eine Optimierung zur Erhöhung der Geschwindigkeit ist das sog. *Inlining*. Was versteht man darunter ?
- Welche Aufgabe übernimmt das Tool ngen.exe im .net-Framework
- Bringen Sie die folgende Begriffe des .NET-Frameworks “Quellcode - CIL-Code und CLR” in einen Zusammenhang
- Erläutern Sie den Begriff **Assembly** sowie DLL-Hölle



Sprachgrundlagen

Bezeichner

Unter Bezeichner versteht man die Namen, die für Konstanten, Variablen, Klassen, Methoden und Ereignisse stehen. Dabei müssen folgende Regeln beachtet werden.

- Keine Verwendung von reservierten Worten (s.u.)
- Maximale Länge 512 Zeichen
- Buchstaben, Ziffern und Unterstrich dürfen verwendet werden
- Bezeichner muss mit Unterstrich oder Buchstaben beginnen.
- Groß und Kleinschreibung wird unterschieden.
- Bezeichner müssen in ihrem Gültigkeitsbereich einen eindeutigen Namen besitzen.
- Benutze **PascalCasing** in zusammenhängenden Worten (z.B. int EineSehrLangeVariable)

Reservierte Worte

Reservierte Worte (Schlüsselworte) sind feste, vorgegebene Worte, die zum Sprachumfang von C# gehören. Sie haben eine feste Bedeutung und dürfen vom Programmierer nicht für eigene Bezeichner verwendet werden.

C# ist case-sensitiv, d.h. die Groß- und Kleinschreibung ist grundsätzlich zu beachten.

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort	using
virtual	volatile	void	while	stackalloc	static	string

Reservierte Worte in C#

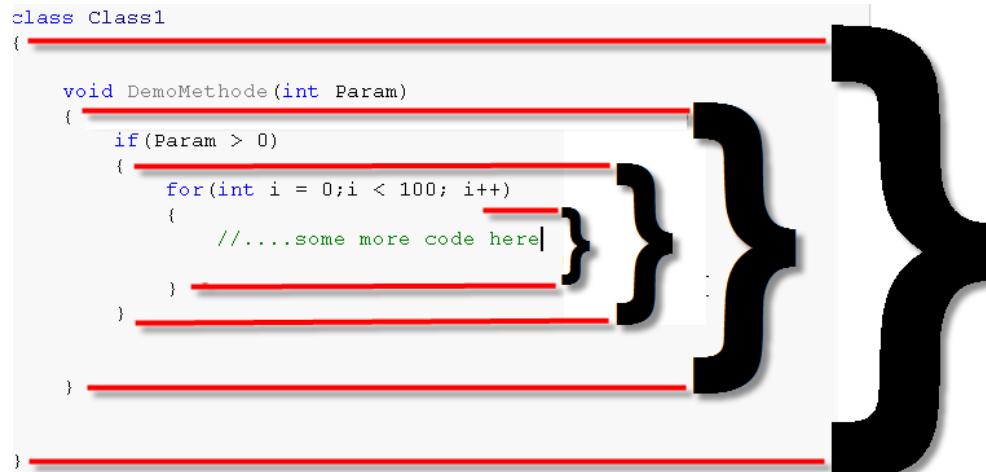


Anweisungen/Anweisungsblöcke/Ausdruck

Eine Anweisung wird in C# durch ein Semikolon abgeschlossen.

Ein Anweisungsblock besteht aus einer Reihe von Anweisungen, die sich innerhalb eines geschweiften Klammerblocks befinden. Anweisungsblöcke können auch ineinander verschachtelt werden.

```
class Class1
{
    void DemoMethode (int Param)
    {
        if(Param > 0)
        {
            for(int i = 0;i < 100; i++)
            {
                //....some more code here
            }
        }
    }
}
```



Alle Bezeichner sind grundsätzlich nur in dem Bereich des Anwendungsblockes gültig, in dem sie deklariert wurden.

Ein Ausdruck ist eine gültige Kombination von Operatoren und Bezeichnern, die ein Ergebnis liefert. Es werden die üblichen mathematischen Regeln angewandt.

Variable/Konstante/Datentyp im Überblick

Jedes Programm braucht für seine Arbeit Speicherplatz, auf dem es Zahlen, Zeichen, Zeichenketten oder andere Objekte ablegen kann. .NET kennt prinzipiell zwei Typen von Variablen, nämlich die sog. **Value Types** und **Reference Types**

Variable werden bei der Deklaration mit einem Namen und einem Datentyp vereinbart. Eine Variable muss vor ihrer Verwendung deklariert werden. Die Namensgebung richtet sich nach den Regeln für Bezeichner.

Aufgaben

- Spot the disallowed variable names
 - int 12count;
 - char \$diskPrice;;
 - char middleInitial;
 - float this;
 - int __identifier;
- Explizite Konvertierung durch casting

Vergleichen Sie die unten abgebildet Quellcodezeilen. Welche Version würden Sie lieber in Ihrem Programm einsetzen ?

```
class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        //Teil1
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue, intValue);

        //Teil2
        checked
        {
            long longValue1 = Int64.MaxValue;
            int intValue1 = (int) longValue1;
            Console.WriteLine("(int){0}={1}", longValue1, intValue1);
        }
    }
}
```

- Value/Reference-Types

Welche Unterschiede gibt es zwischen den sog. Value-Types und den sog. Reference Types

Was ist der Unterschied zwischen Stack und Heap ?

In welchem Speicherbereich werden enum- und struct-Datentypen verwaltet.

- Gültigkeit/Sichtbarkeit von Variablen

Was passiert, wenn sie folgenden Code compilieren wollen?

```
class BlockTest
{
    public void zugriff()
    {
        int aussen = 7;
        if (aussen == 7)
        {
            int innen = 8;
            Console.WriteLine(aussen);
            Console.WriteLine(innen);
        }
        Console.WriteLine(innen);
    }

    public static void Main (String[] args)
    {
        BlockTest ref = new BlockTest();
        ref.zugriff();
    }
}
```

- Welche Ausgabe erzeugt untenstehendes Programm ?

```
class Sichtbar
{
    int x;
    public void zugriff()
    {
        int x = 7;
        Console.WriteLine(x);
        Console.WriteLine(this.x);
    }

    public static void Main (String[] args)
    {
        Sichtbar sicht = new Sichtbar();
        sicht.zugriff();
    }
}
```



- Welchen Gültigkeitsbereich haben die unten dargestellten Variablen ?

```
01 {  
02     ...  
03     int y;  
04     {  
05         int x;  
06         {  
07             int z;  
08  
09         }  
10     }  
11 }
```

Lösung:

Die Variable z darf nicht in x umbenannt werden, denn für die lokalen Variablen gilt, dass deren Namen nicht die Namen von weiter außen definierten Objekten verdecken dürfen. Der Name einer Variable in einem { - } - Block darf aber den Namen einer Variable tragen, die auf der Ebene der Klasse definiert wurde



- Welche Ausgabe erzeugt folgendes Programm. Für die Wirkung der Operatoren informieren sie sich in der MSDN. Versuchen Sie das Ergebnis herauszufinden, ohne das Programm zu starten :-).

```
class Ausdruck
{
    public static void Main (String[] args)
    {
        Console.WriteLine(4+5);

        int a = 4;
        int b = 5;
        int c = 0;

        Console.WriteLine(a + (b*c));
        Console.WriteLine(((a > 100) && (b<1000)) || (c>200));
    }
}
```

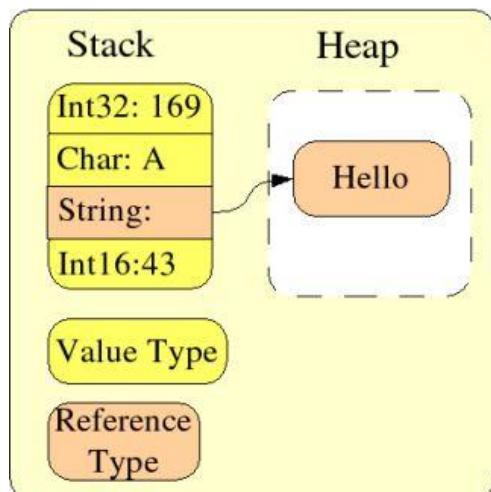
- Erstellen Sie eine Konsolenanwendung. Lesen Sie einen int-, double-, string-Wert mithilfe der Tastatur ein (Console.ReadLine()) und speichern Sie die Werte in entsprechenden Variablen. Geben Sie anschließend die Werte der drei Variablen auf dem Bildschirm aus.

Datentypen

- <http://www.codeproject.com/Articles/18714/Comparing-Values-for-Equality-in-NET-Identity-and>
- <http://www.codeproject.com/Articles/544990/Understand-how-bitwise-operators-work-Csharp-and-V>

Wert- und Verweistypen

Moderne Programmiersprachen unterscheiden zwei Arten von Datentypen.



1. Werttypen

Variable selbst hat einen eigenen Speicherbereich fÃ¼r den Wert; beide liegen auf dem sog. **Stack**

2. Verweistyp

Er enthÃ¤lt lediglich die Adresse eines anderen Speicherortes, der die eigentlichen Daten enthÃ¤lt.

Werttyp	Verweistyp
sbyte, byte, char,	string, <code>object</code>
ushort, uint, <code>int</code> ,	Arrays
ulong, long, <code>float</code> ,	
double, decimal	class, interface, delegate
<code>bool</code> , enum, struct	

Tabelle: Wert- und Verweistypen

Numerische Datentypen

Datentyp	Wertebereich	Speichergröße
sByte	-128 - 127	1 Byte
byte	0-255	1 Byte
short	-32.768 - 32.767	2 Bytes
ushort	0-65535	2 Bytes
int	-2.147.483.648 - 2.147.483.647	4 Bytes
uint	0 - 4.294.967.295	4 Bytes
long	-9.223.372.036.854.775.808 - 9.223.372.036.854.775.807	8 Bytes
ulong	0 - 18.446.744.073.709.551.615	8 Bytes
decimal	0 - +/- 79.228.162.514.264.337.593.543.950.335	16 Bytes

Tabelle: Ganzzahl-Datentypen

Datentyp	Wertebereich	Genauigkeit	Speichergröße
float	ca. 1,4 * 10 ^ 45 bis 3,4 * 10 ^38	7-8 Stellen	4 Bytes
double	ca. 4,9 * 10 ^ - 324 bis 1,8 * 10 ^308	15 - 16 Stellen	8 Bytes
decimal	0 - +/- 79.228.162.514.264.337.593.543.950.335	28 Stellen	16 Bytes

Tabelle: Gleitkomma-Datentypen

Decimal lässt sich sowohl für Ganzzahlen als auch für Nachkommazahlen einsetzen. Wenn Sie einer Variablen vom Typ decimal eine Wert zuweisen, müssen Sie ein **M** oder **m** hinzufügen.

```
decimal x = 99.456m
```

Zeichen-Datentypen

Zeichen-Datentypen enthalten beliebige Zeichen oder Zeichenketten des **Unicode**-Zeichensatzes. Es lassen sich damit die Zeichen vieler Schriftsprachen abdecken.

Datentyp	Wertebereich	Speichergröße
char	65536 verschiedene Unicode-Zeichen	2 Bytes
string	variable Anzahl von Unicode-Zeichen	max. 4 GBytes

Tabelle: Zeichen-Datentypen

- Zeichen werden durch Apostrophe ('A'), Zeichenketten durch Anführungszeichen ("Hallo") eingeschlossen.
- Ein Anführungszeichen lässt sich durch Voranstellen eines \ darstellen

```
s = "Ein sogenannter \"String\"";
```

- Auf einzelnen Zeichen eines Strings kann per Index(0-basiert) zugegriffen werden.

```
string s = "Ein sogenannter \"String\""; char a = s[1]; //gibt i zurück
```

- Ein **Zeilenwechsel** lässt sich durch ein "\n", ein **Wagenrücklauf** durch ein "\r", ein **horizontaler Tabulator** durch ein "\t" darstellen.

```
Console.WriteLine("123" + "\n" + "4");
Console.WriteLine("123" + "\r" + "4");
Console.WriteLine("123" + "\t" + "4");
```

- Sogenannte **verbatim-Strings** werden durch das @-Zeichen eingeleitet. Sie erlauben es, Steuerzeichen zu benutzen, die man sonst maskieren müsste

```
string Pfad = @"C:\Windows\Desktop";
```



Datentyp DateTime

Dieser Datentyp dient zur Speicherung von Datums- und Zeitwerten.

Datentyp	Wertebereich	Speichergröße
DateTime	1. Januar 1 bis 31. Dezember 9999.	
	Uhrzeiten von 0:00.00 bis 23:59:59.	8 Bytes
	Die kleinste Einheit zum Messen sind 100 Nanosekunden (Tick)	

Tabelle: Datentyp DateTime

Ein Datumswert kann auf verschiedene Arten eingegeben werden, z.B. mit Hilfe der Methoden `System.Convert.ToDateTime` oder `System.DateTime.Parse`.

Boolesche Datentypen

Datentyp	Wertebereich	Speichergröße
<code>bool</code>	Es können nur die Werte <code>false (0)</code> und <code>true (ungleich 0)</code> angenommen werden.	2 Bytes

Tabelle: Datentyp `bool`

Typumwandlungen (Casten)

Häufig ist es notwendig, einen Datentyp in einen anderen umzuwandeln. Solange man dabei einen in der Speichergröße kleinere Datentyp in einen größeren Datentyp wandelt, sind keine Probleme zu erwarten. Der Compiler wird dies ohne Fehlermeldung durchführen (**implizites Casten**).

Im umgekehrten Fall kann es zu Problemen kommen, da die Speichergröße der kleineren Datentyps evtl. den jeweiligen Wert nicht aufnehmen kann. In diese Falle würde der Compiler

eine Fehlermeldung generieren. Man benötigt deshalb eine explizite Konvertierung (**explizites Casten**), bei der der gewünschte Zieldatentyp in Klammern vorangestellt wird.

```
//implizites Casten
long d;
int i;
i = 10;
d = i;

//explizites Casten;
int d;
long i;

i=10;
d = (int) i;
```

Die Klasse Convert

Diese Klasse kann ebenfalls für Typkonvertierungen genutzt werden.

```
System.Int32 i;
System.String s = "65656";
i = System.Convert.ToInt32(s);
```

Die Methoden ToString und Parse

Mit diesen Methoden der meisten Standarddatentypen können auf einfache Weise Zahlen in Zeichenketten und umgekehrt konvertiert werden

```
string s;
int i = 10;
s = i.ToString();
i = Int32.Parse(s);
```

Operatoren

Mit Operatoren lassen sich Werte, auch Operanden genannt, miteinander verknüpfen. Operatoren können dabei nach der Anzahl der Operanden in drei Kategorien eingeteilt werden.

- Einstellige Operatoren haben einen,
- Zweistellige Operatoren haben zwei
- Dreistellige Operatoren haben drei Operanden.

Man bezeichnet die Operatoren synonym dazu auch als **unär**, **binär**, **ternär** bzw. **monadisch**, **dyadisch**, **triadisch**.

Des Weiteren muss geklärt werden, in welcher Reihenfolge Operatoren und ihre Operanden in Programmen geschrieben werden. Man spricht in diesem Zusammenhang auch von der Notation der Operatoren.

Der einzige dreistellige Operator ?:, benutzt ebenfalls die Infix-Notation, also

```
<Operand> ? <Operand> : <Operand>
```

- Die meisten einstelligen Operatoren werden in C# in der Präfix-Notation verwendet. Eine Ausnahme davon bilden die Inkrement- und Dekrementoperatoren, die sowohl in der Präfix- als auch in der Postfix-Notation verwendet werden können.

```
<Operand> <Operator> oder  
<Operator> <Operand>
```

- Zweistellige Operatoren verwenden stets die Infix-Notation, in der der Operator zwischen seinen beiden Operanden steht, also

```
<Operand> <Operator> <Operand>
```

Operand	Beispiel	Wirkung
+	+a	keine explizite Funktion
-	-a	Negiert a

Tabelle: Einstellige arithmetische Operatoren

Operand	Beispiel	Wirkung
+	a + b	addiert a und b
-	a - b	Subtrahiert b von a
*	a * b	Multipliziert a mit b
/	a / b	Dividiert a durch b
%	a % b	Restwert einer Division

Tabelle: Zweistellige arithmetische Operatoren

Arithmetische Operatoren

Operator	Beispiel	Beschreibung
+	10 + 11 => 21	Die Operanden werden addiert
	10.1 + 10.2 => 20.3	
-	10 - 2 => 8	Subtraktion
	2.1 - 0.89 => 1.21	
	y = -x	Negation numerischer Werte
*	10 * 12 => 120	
	1.1 * 1.1 => 1.21	Operanden werden multipliziert
/	10 / 12 => 0.8333	Die Operanden werden dividiert.
	10.1 / 10.2 => 0.9901	Wenn Sie diese Division mit Ganzzahl-Datentypen durchführen, werden diese intern in den Datentyp Double umgewandelt.
%	45 % 13 => 6	Modulo-Operator, über den der Restwert einer Division bestimmt werden kann.

Tabelle Arithmetische Operatoren

Arithmetische Operatoren sind Operatoren, die Zahlen, also Werte vom Typ byte, short, int, long, float, double, char als Operanden erwarten.

Die Operatoren + und - können sowohl als einstellige als auch zweistellige Operatoren benutzt werden. Weiterhin kann der + - Operator für das Verketten zweier Zeichenketten verwendet werden.

Eine weitere Besonderheit stellt der Ergebnistyp arithmetischer Operationen dar, der durchaus nicht mit dem Typ beider Operanden übereinstimmen muss.

Aufgabe: Bestimmen Sie das Ergebnis folgender Programmzeilen:

```
public static void main (String args[])
{
    short a = 1;
    short b = 2;
    short c = a + b;
}
```

The screenshot shows the SharpDevelop IDE interface. On the left is a code editor with the following C# code:

```
13     class MainClass
14 {
15     public static void Main(string[] args)
16     {
17         short a = 1;
18         short b = 2;
19         short c = a + b;
20     }

```

To the right of the code editor is an "Ausgabe" (Output) window titled "Erstellen". It displays the following error message:

```
c:\Dokumente und Einstellungen\Administrator\Eigene Dateien\SharpDevelop Projects\tst\Main.cs
(19,14): error CS0029: Implizite Konvertierung des Typs 'int' zu 'short' nicht möglich.

Abgeschlossen -- 1 Fehler, 0 Warnungen
```

Versuchen Sie das Verhalten des Compilers zu erklären. Benutzen Sie dazu die MSDN.

Suchen Sie dazu nach dem Compilerfehler CS0029

Damit ist auch oben aufgetretene Fehlermeldung erklärbar und behebbar. Das Ergebnis der Operation muss explizit auf short gecastet werden.

```
short c = (short) (a + b)
```

Besonderheiten bei Divisionen

Was ist das Ergebnis des folgendes Codes

```
using System;
class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        float ergebnis;
        float nenner;
        float zaehler;
        zaehler = 20;
        nenner = 0;
        ergebnis = zaehler / nenner;
        Console.WriteLine(zaehler + " / " + nenner + " = " + ergebnis);
        zaehler = 0;
        nenner = 0;
        ergebnis = zaehler / nenner;
        Console.WriteLine(zaehler + " / " + nenner + " = " + ergebnis);
        int a = -7;
        int b = 3;
        Console.WriteLine(a/b);

        int c = 0;
        int d = 0;
        Console.WriteLine(c/d);
    }
}
```

Bei der Verwendung von ganzzahligen Operanden ist das Ergebnis wieder eine ganze Zahl.
Der Nachkommateil wird abgeschnitten.

Bei der Division durch 0 wird bei Ganzzahldivision eine ArithmeticException, bei Gleitpunktdivision wird als Ergebnis Infinity incl. Vorzeichen zurückgeliefert.

Ist bei einer ganzzahligen Division der Zähler oder Nenner negativ, so ist das Ergebnis negativ. Das Ergebnis bemisst sich dabei nach den arithmetischen Regeln der IEEE 754

Suchen Sie in der MSDN nach einer Zusammenfassung zu den Divisionsregeln.

Logische Operatoren

Operator	Operandentyp	Beispiel	Beschreibung
<hr/>			
! (NOT)	Boolean	$\text{!true} \Rightarrow \text{false}$ $\text{!false} \Rightarrow \text{true}$	Bei booleschen Datentypen wird die logische Negation des Wertes als Ergebnis zurückgegeben. ``true ``wird zu ``false`` und umgekehrt.
<hr/>			
& (AND)	Ganzzahl	$100 \& 115 \Rightarrow 96$	Bei Ganzzahl-Datentypen wird eine bitweise Konjunktion durchgeführt. 1. Operand 10011 Als Ergebnis ergibt sich 2. Operand 10001 die 1, wenn beide Zahlen Ergebnis 10001 an derselben Stelle eine 1 besitzen.
<hr/>			
& (AND)	Boolean	$\text{true} \& \text{true} \Rightarrow \text{true}$ $\text{true} \& \text{false} \Rightarrow \text{false}$ $\text{false} \& \text{true} \Rightarrow \text{false}$ $\text{false} \& \text{false} \Rightarrow \text{false}$	Bei booleschen Datentypen wird nur dann true zurückgeliefert, wenn beide Operanden den Wert true besitzen. 1. Operand 10011 Es werden immer beide Operanden ausgewertet. 2. Operand 10001 Ergebnis 10001
<hr/>			
(OR)	Ganzzahl	$100 115 \Rightarrow 119$	Bei Ganzzahl-Datentypen wird eine bitweise Verknüpfung durchgeführt. Dabei ergibt sich 1, wenn mindestens eine der beiden Zahlen an der gleichen Stelle eine 1 besitzt.
<hr/>			
(OR)	Boolean	$\text{true} \text{false} \Rightarrow \text{true}$ $\text{false} \text{true} \Rightarrow \text{true}$ $\text{true} \text{true} \Rightarrow \text{true}$ $\text{false} \text{false} \Rightarrow \text{false}$	Bei booleschen Datentypen wird dann true zurückgeliefert, wenn mind. einer der Operanden den Wert true besitzt. Es werden immer beide Operanden ausgewertet.
<hr/>			
^ (XOR)	Ganzzahl	$100 ^ 115 \Rightarrow 23$	Es wird eine bitweise Verknüpfung durchgeführt.

	1. Operand 10111	Dabei ergibt sich eine 1,
	2. Operand 10001	wenn beide Zahlen an
	Ergebnis 00110	derselben Stelle einen
		unterschiedlichen Wert
		besitzen.
<hr/>		
^ (XOR) Boolean	true ^ false => true	Ergibt dann true, wenn
	false ^ true => true	beide Operanden einen
	true ^ true => false	unterschiedlichen Wert
	false ^ false => false	besitzen.
<hr/>		
&& Boolean	true && true => true	Dieser Operator arbeitet
	true && false => false	wie der & - Operator.
	false && => false	Die Auswertung des 2.
	false && => false	Ausdrucks erfolgt jedoch
		nur dann, wenn der erste
		den Wert ``true`` hat.
<hr/>		
Boolean	true => true	Dieser Operator arbeitet
	true => true	wie der - Operator.
	false true => true	Die Auswertung des 2.
	false false => false	Ausdrucks erfolgt jedoch
		nur dann, wenn der erste
		den Wert ``false`` hat.
<hr/>		

Verschiebeoperatoren

Operator	Operandentyp	Beispiel	Beschreibung
<hr/>			
<<	int, uint, long, ulong	13 << 1 => 26 (01101 => 11010)	Es wird ein bitweises Verschieben um die Zahl der angegebenen Stellen nach links durchgeführt
<hr/>			
>>	int, uint, long, ulong	26 >> 1 => 13 (11010 => 01101)	Hier werden alle Bits um die Zahl der angegebenen Stellen nach rechts verschoben
<hr/>			

Vergleichsoperatoren

Operator	Operandentyp	Beispiel	Beschreibung
<hr/>			
==	alle Datentypen	10 == 11 => false 10 == 10 => true	Untersucht zwei Operanden auf Gleichheit. Sind sie gleich, wird ``true``, ansonsten ``false`` als Ergebnis geliefert.
<hr/>			
!=	alle Datentypen	10 != 11 => true 10 != 10 => false	Untersucht zwei Operanden auf Ungleichheit. Sind sie ungleich, wird ``true``, ansonsten ``false`` als Ergebnis geliefert.
<hr/>			
> < <= >=	numerische	10 > 10 => false 11 < 10 => true 11 <= 10 => false 9 > 10 => false	
<hr/>			

Bei der Anwendung der Operatoren `==`, `!=` auf so genannte Verweistypen (Objekte), wird geprüft, ob die Variable auf das gleiche Objekt verweist! Wenn geprüft werden soll, ob die Inhalte von Verweistypen gleich sind, muss man die Methode `equals` verwenden.

```
class Test
{
    public static void Main()
    {
        object o1 = 2;
        object o2 = 2;

        System.Console.WriteLine((o1 == o2).ToString());
        System.Console.WriteLine(o1.Equals(o2).ToString());
    }
}
```

Verkettungsoperator

Operator	Operandentyp	Beispiel	Beschreibung
<code>+</code>	<code>string</code>	<code>"ket" + "te"</code>	Zeichenketten werden aneinandergehängt.

Zuweisungsoperator

Operator	Operandentyp	Beispiel	Beschreibung
=	alle	i = 24;	Einer Variablen wird ein Wert zugewiesen.
+=	numerische Variablen	i += 1	Kurzform für i = i + 1;
	string	s += s2	Kurzform für s = s + s2;
-=, *=,	numerische Variablen	i -= 1, i *= 1;	Kurzform für i = i - 1; i = i * 1;
/=, %=,	Variablen	i /= 1, i \= 1;	i = i / 1; i = i \ 1;
^=		i ^= 1,	i = i \\\ 1; i = i ^ 1;
&=	Ganzzahl, boolean	i &= 12	Kurzform für i = i & 1;
=	Ganzzahl, boolean	i = 12	Kurzform für i = i 1;
<=	int, uint, long, ulong	i <= 1	Kurzform für i = i << 1;
>=	int, uint, long, ulong	i >= 1	Kurzform für i = i >> 1;

Inkrement- und Dekrementoperatoren

Operator	Operandentyp	Beispiel	Beschreibung
<hr/>			
++	numerisch	<code>int i = 1;</code> <code>i++; ++i;</code>	Numerische Variablen werden um den Wert 1 erhöht.
<hr/>			
--	numerisch	<code>int i = 1;</code> <code>i--; --i;</code>	Numerische Variablen werden um den Wert 1 erniedrigt.
<hr/>			

Operatoren zum Erhöhen/Erniedrigen gibt es in der Präfix (++Variable) bzw. in der Postfix (Variable++). Welche Unterschiede gibt es im folgenden Code

```
int i = 1;
Console.WriteLine(i++);

int z = 1;
Console.WriteLine(++i);
```

Operatorenvorrang

Für die Ausführungsreihenfolge der Operationen in einem Ausdruck werden Operatoren der Reihenfolge nach gegliedert. Die Rangstufen sind in der unten abgebildeten Tabelle von oben (höchste Priorität) nach unten geordnet. Innerhalb der verschiedenen Rangstufen gibt es bei den logischen und arithmetischen Operatoren ebenfalls eine Rangordnung (ranghöhere sind zuerst genannt). Mehrere gleichwertige Operatoren werden in der Regel von links nach rechts abgearbeitet. Es gibt aber auch Operatoren (Zuweisungsoperatoren), bei denen die Abarbeitung von rechts nach links erfolgt.

Die Ausführungsreihenfolge kann durch Klammerung beeinflusst werden.

Siehe dazu auch [*MSDN*](#).

[*Zfsfq_DatentypenOperatoren*](#).

[*Operatorenrangfolge*](#).

Fragen

Beachte

Versuchen sie die Fragen erst durch Überlegen zu lösen, bevor Sie die F5-Taste der IDE benutzen.

- Der Datentyp **string** hat als Verweistyp eine besondere Stellung.

Informieren Sie sich in der MSDN über folgende Methoden dieses Datentyps

- Copy()
- Remove()
- Insert()
- Length
- Trim()
- IndexOf()
- ToUpper()
- ToLower()
- Replace()
- Substring()
- Concat()

- Neben dem Datentyp **string** gibt es noch im Namensraum **System.Text** die Klasse **String-Builder**. Informieren Sie sich in der MSDN über die Unterschiede und Einsatzmöglichkeiten dieser Klasse.
- Sind folgende Anweisungen möglich

```
System.DateTime Datum;

Datum = Convert.ToDateTime("24/12/02");
Datum = Convert.ToDateTime("24.12.2002");
Datum = Convert.ToDateTime("24. Dezember 2002");
Datum = Convert.ToDateTime("December 24, 2002");
Datum = Convert.ToDateTime("24/12/02");
Datum = DateTime.Parse("24/12/02");
```

4. Was bedeuten die Begriffe **NaN** und **Infinity/unendlich**?
5. Welche Ausgabe erzeugt folgender Code

```
int i = 1; int z = 2;
Console.WriteLine(i++ * z++);
Console.WriteLine(++i * z++);
Console.WriteLine(i-- * --z);
```

6. Beurteilen Sie ob die Ausgabe aller Zeilen die gleiche ist!

```
int x=1, y=1, z=1 ,a=1 ;
x = (x * 2) + 1;
y = (y << 1) + 1;
z= z * 2 + 1;
a= a << 1 + 1;

Console.WriteLine(x);
Console.WriteLine(y);
Console.WriteLine(z);
Console.WriteLine(a);
```

7. Erzeugen Sie 2 Konsolenprojekte (eines davon in c#, eines davon in visual basic .net). Geben Sie im C#-Projekt folgende Anweisungen ein. Welches Ergebnis erwarten Sie.

```
float f = 10.57f;
int i = (int) (f*100);
Console.WriteLine(i);
=====
float f = 10.57f;
f = f * 100;
int i = (int) f;
Console.WriteLine(i);
```

Geben Sie im Vb-Projekt folgenden Code ein. Welches Ergebnis erwarten Sie. Versuchen Sie den Unterschied im Ergebnis im Vergleich zum ersten c#-Beispiel zu ergründen.

```
Dim f As Single = 10.57
Dim i As Integer = f * 100
Console.WriteLine(i)
```

8. Welche Aussagen sind zum folgenden Code-Fragment richtig?

```
char c= '-' ;
res = +c;

Console.WriteLine(res);
```

- Die Variable res kann vom Typ double sein.
- Die Variable res kann vom Typ char sein.
- Ist die Variable res vom Typ int, ist die Ausgabe eine positive Zahl.
- Der Code führt zu einem Fehler beim Kompilieren.

9. Welche Aussagen sind zum folgenden Code-Fragment richtig?

```
int i=0, j=1;
Console.WriteLine(++i + j++ +" " +i +" " +j);
```

- Die Ausgabe ist: 2 1 1
- Die Ausgabe ist: 1 1 2
- Die Ausgabe ist: 2 1 2
- Der Code führt zu einem Fehler beim Kompilieren.

10. Welches Ergebnis erzeugt folgendes Codefragment?

```
static void Main(string[] args)
{
    char c= '1';
    Console.WriteLine(-5%3 + c);
}
```

- Die Ausgabe ist: ????
- Der Code führt zu einem Fehler beim Kompilieren.



11. Welche Aussagen sind zum folgenden Code-Fragment richtig?

```
byte b = 1;
boolean bo1 = true, bo2 = false;
Console.WriteLine(bo1^!bo2 & false ?++b:b--);

- Die Ausgabe ist: 1
- Die Ausgabe ist: 2
- Die Ausgabe ist: 0
- Der Code führt zu einem Fehler beim Kompilieren.
```

12. Welche Aussagen sind zum folgenden Code-Fragment richtig?

```
int i=-1;
byte b=1;
i= i>>32;
b >>= 20;
Console.WriteLine(i);           // Ausgabe: 0
Console.WriteLine(i>>10>>10>>10); // Ausgabe: -1
Console.WriteLine(i>>10>>10>>10); // Ausgabe: 3
Console.WriteLine(b);          //Ausgabe: -1
//Die Anweisung b>>>= 20; ist gleichbedeutend mit b =
b>>>20;
```

13. Welche Ausgaben sind zu den folgenden Deklarationen richtig?

```
String s1= new String("100"), s2= new String("100");
String s3= null; int i=100;
Console.WriteLine(s1==s2);    //Ausgabe: true
Console.WriteLine(s1+=i);    //Ausgabe: 100100
Console.WriteLine(s2+=s3);    //Ausgabe: 100
Console.WriteLine(i+=s1);    //Ausgabe: 200
s3!=null & s3.length()>0?">0":"0"); Ausgabe: Laufzeit-Fehler (Exception)
```

14. Welche Ausgaben sind zu den folgenden Deklarationen richtig ?

```
int a = -1, b = 8, c = 3;
Console.WriteLine(~a==(a^a));           //Ausgabe: true
Console.WriteLine(~~a==a);             //Ausgabe: true
Console.WriteLine(!a<0 ? a:-a);        //Ausgabe: 1
Console.WriteLine(a>>>1==Integer.MAX_VALUE); //Ausgabe: false
Console.WriteLine((b&c) % (b|c));      //Ausgabe: 0
Console.WriteLine((b|c) % (b&c));      //Ausgabe: 0
```

15. Welche Ausgaben erzeugt true zu den folgenden Deklarationen?

```
double a=0.0, b=1.0;
Console.WriteLine(a/a==0.0/0.0);
Console.WriteLine(b/a > 0.0/0.0);
Console.WriteLine(b/a > Double.MAX_VALUE);
Console.WriteLine((int)(b/a)== Integer.MAX_VALUE);
Console.WriteLine(b/a == 1/0);
```

16. Welche Aussagen sind zu den folgenden Deklarationen richtig?

```
int i= 1; Integer j= new Integer(1), k= new Integer(1);
Console.WriteLine(j);                  //Ausgabe: 1
Console.WriteLine(j==k);              //Ausgabe: true
Console.WriteLine(j.equals(k));       //Ausgabe: true
Console.WriteLine(i==j);              //Fehler beim Kompilieren
Console.WriteLine((i+"").equals(j+"")); //Fehler beim Kompilieren
```

17. Welche Ausgabe erzeugt folgende Quellcode ?

```
public static void Main (String args[])
{
    double x1 = 10^20;
    double x2 = 1223;
    double x3 = 10^18;
    double x4 = 10^15;
    double x5 = 3;
    double x6 = -10^12;
    double y1 = 10^20;
    double y2 = 2;
    double y3 = -10^22;
    double y4 = 10^13;
    double y5 = 2111;
    double y6 = 10^16;
    Console.WriteLine (x1*y1 + x2*y2 + x2*y2 +x3*y3 + x4*y4
+ x5*y5 + x6*y6);
}
```

18. Zu welchen Fehlern führt folgender Ausschnitt aus einem Deklarationsblock?

```
public static void Main (String args[])
{
    int a;
    byte c = 200;
    long y = a + c;
    boolean b1 = c > 200;
    boolean b2 = y;
    short x = 47;
    float a = 42.0;
}
```

19. Welche Werte liefern folgende Ausdrücke (alle Variable seien vom Typ int)?

```
public static void Main (String args[])
{
    48 / 6 / 2;
    48 / ( 6 / 2 );
    x = 10;
    | y = "x";
    z = x;
    x = 10;
    y = x++;
    z = x;
    x = 10;
    a = x++ * x++;
    ( x > 10 ) \ -11 : 10;
    5 << 2;
    100 >> 3;
    100 >>> 3;
    -1 >> 1;
    -1 >>> 1;
    ( 1 << 30 ) * ( 1 << 2 )
}
```



Kontrollstrukturen

Download:

- Siehe **Übung**.
- Siehe **PAP**.
- Siehe **GOTO**.
- Siehe: <http://www.jamesshore.com/Articles/Quality-With-a-Name.html>
- Video **MSDN_Teil3**.

Ein Programm besteht aus einer Folge von Anweisungen, die häufig nicht nur sequenziell(hintereinander) ablaufen, sondern die wiederholt oder evtl. auch gar nicht ausgeführt werden sollen. Viele Problemstellungen sind umfangreich und kompliziert und erfordern deshalb eine systematische Vorarbeit. Beim Entwurf werden grafische Darstellungsmittel für die Logik des Programmablaufes verwendet.

Die Logik eines Programmablaufes kann mit den untenstehenden Konzepten realisiert werden.

- Anweisung
- Auswahl
- Wiederholung

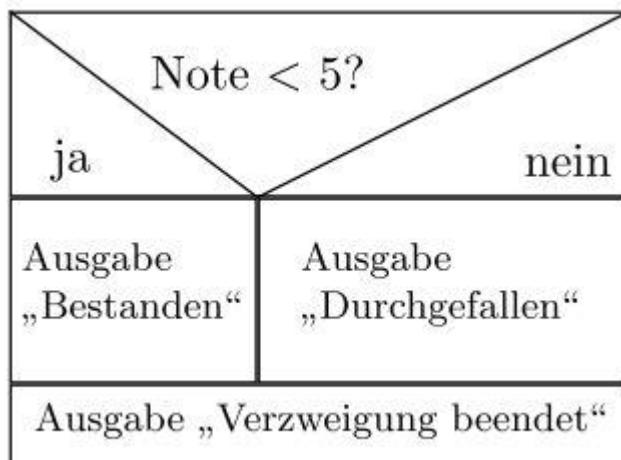
Anweisung

Die Anweisung ist der grundlegende Befehl innerhalb einer Programmiersprache. Im Struktogramm wird er durch ein Rechteck abgebildet. Durch die Folge von Anweisungen entsteht ein sequenzieller Ablauf eines Programms.

Einseitige/Zweiseitige Auswahl

Ein Programm kann, abhängig von einem Kriterium, eine Entweder-oder-Entscheidung treffen, d.h. entweder einen bestimmten Programmteil "A" ausführen oder einen anderen bestimmten Programmteil "B". Der nötige Code lässt sich sehr intuitiv verstehen. Sehen wir uns als Beispiel den Code an, der, abhängig vom Wert der Variablen note, den Text "Bestanden" oder "Durchgefallen" ausgibt:

```
if (note<5) {
    Console.WriteLine("Bestanden");
} else {
    Console.WriteLine("Durchgefallen");
}
Console.WriteLine("Verzweigung beendet");
```



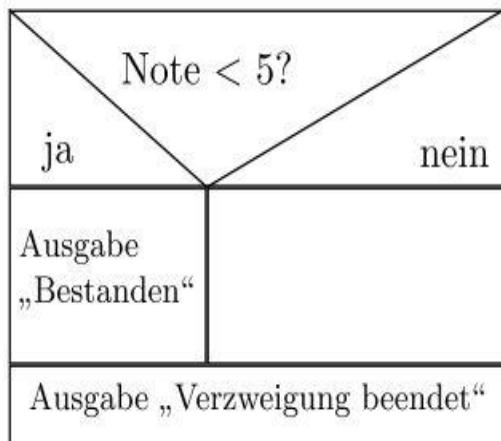
Zuvorsteht ein neues Schlüsselwort: if (wenn). Dann folgt in Klammern das Kriterium: "Wenn note kleiner als 5 ist". Wenn das der Fall ist, werden alle Anweisungen in dem folgenden, von geschweiften Klammern eingeschlossenen Block ausgeführt. Dann folgt ein weiteres Schlüsselwort else (sonst), gefolgt von einem weiteren Block, der ausgeführt wird, wenn note eben nicht kleiner als 5 ist. Danach ist die Verzweigung beendet, das

heißt, der folgende Code wird auf jeden Fall ausgeführt und hängt nicht mehr von der if-Bedingung ab. Im Struktogramm sieht das so aus:

Man kann auch den else-Teil komplett weglassen. Wenn man zum Beispiel die Meldung "Durchgefallen" nicht braucht, kann man schreiben:

```
if (note < 5) {
    Console.WriteLine("Bestanden");
}
Console.WriteLine("Verzweigung beendet");
```

Das ist die sogenannte Einseitige Auswahl im Gegensatz zur Zweiseitigen Auswahl, die wir eben kennengelernt haben. Das Struktogramm hat jetzt folgendes Aussehen:



Wenn wir anders herum den "ja-Zweig" (korrekt würde man sagen: if-Zweig) weglassen und nur den "nein-Zweig" (korrekt: else-Zweig) implementieren wollen, wird es etwas schwieriger. Wir wandeln unser Beispiel so ab, dass nur der Text "Durchgefallen" ausgegeben werden soll. Möglich wäre folgendes:

```

if (note<5) { //kein guter Stil
} else {
    Console.WriteLine("Durchgefallen");
}
  
```

Das läuft zwar, ist aber kein guter Stil. Besser ist es, die Bedingung umzudrehen:

```

if (note >= 5) {
    Console.WriteLine("Durchgefallen");
}
  
```

Kommen wir schließlich noch zur Bedingung, die in den Klammern steht. Was darin steht, ist das Ergebnis der Rechenoperation **note<5**. Die Rechenoperation ist eine sogenannte Vergleichsoperation und das Ergebnis ist ein Wahrheitswert. Ein Wahrheitswert kann nur die beiden Werte wahr und falsch annehmen.

Die öffnende und die schließende geschweifte Klammer darf in der if-Anweisung weggelassen werden. Dies ist jedoch immer schlechter Stil und kann leicht zu schwer auffindbaren Fehlern führen. Werden die Klammern weggelassen, besteht der if-Block aus der Zeile, die der if-Anweisung folgt und der else-Block besteht aus der Zeile, die der else-Anweisung folgt. Beispiel:

```

if (wert%2==0)
    Console.WriteLine("Die Zahl ist gerade");
  
```

Die Gefahr darin zeigt sich in folgendem Codeausschnitt:

```
if (wert%2==0)
    Console.WriteLine("Die Zahl ist gerade");
    Console.WriteLine("Die Zahl ist durch 2 teilbar");
```

Entgegen dem Anschein wird die zweite WriteLine-Zeile auch bei ungeraden Zahlen ausgeführt, denn Java verwendet wegen der fehlenden geschweiften Klammern nur die erste WriteLine-Zeile für den if-Block. Dies ist ein nachträglich schwer zu findender Fehler, der von vornherein vermieden werden kann, wenn man konsequent Klammern für den if und den else-Block setzt. Ein weiteres schwer auffindbarer Fehler ist:

```
if (wert%2==0);
    Console.WriteLine("Die Zahl ist gerade");
```

Diesen Code muss man folgendermaßen interpretieren: Falls die Bedingung wahr ist, werden die Anweisungen bis zum nächsten Semikolon ausgeführt, also bis zum Semikolon am Ende der if-Zeile. Anschließend ist die if-Verweigung zu Ende. Die WriteLine-Anweisung wird also immer ausgeführt, gleichgültig ob wert gerade oder ungerade ist.

if-else-Kaskaden

In einem Sonderfall lässt man teilweise die geschweiften Klammern aber doch weg. Manchmal gibt es mehr als zwei Fälle, die unterschiedlich behandelt werden müssen. In diesem Fall schachtelt man mehrere if-Anweisungen ineinander und erhält die sogenannte if-else-Kaskade. Im nachfolgenden Beispiel bauen wir die Ausgabe einer Schulnote so weit aus, dass für jede Note ein individueller Text ausgegeben wird.

```
if (note==1) {
    Console.WriteLine("sehr gut");
} else if (note==2) {
    Console.WriteLine("gut");
} else if (note==3) {
    Console.WriteLine("befriedigend");
} else if (note==4) {
    Console.WriteLine("ausreichend");
} else if (note==5) {
    Console.WriteLine("mangelhaft");
} else {
    Console.WriteLine("Fehler im Programm");
}
```

Mehrseitige Auswahl

In allen neueren Sprachen, gibt es eine Verzweigung, die, abhängig von einer Integer-Variablen, einen von mehreren Programmblöcken anspringt. Das Notenprogramm, das im vorigen Kapitel mit einer if-else-Kaskade gelöst wurde, ist ein gutes Beispiel dafür. Das Struktogramm dazu ist:

1	2	3	4	5	note	sonst
Ausgabe „sehr gut“	Ausgabe „gut“	Ausgabe „befriedigend“	Ausgabe „ausreichend“	Ausgabe „mangelhaft“	Ausgabe „Programmfehler“	

switch-Anweisung

Die entsprechende Anweisung heißt in C# switch-Anweisung. In anderen Sprachen ist sie als case- oder select-Anweisung bekannt. Sie beginnt mit einer Zeile switch, gefolgt von der Variablen, deren Wert für die Verzweigung herangezogen wird:

```
switch (note) {
```

Dann folgt für jede Note ein sogenannte case-Block. Die erste Zeile eines case- Blocks wird eingeleitet durch das Schlüsselwort case, gefolgt von dem Wert, für den der Block ausgeführt werden soll und einem Doppelpunkt:

case 1:

Dann kommen die Anweisungen für den Block. Ein Block wird mit dem Befehl break abgeschlossen.

```
switch(note) {
    case 1: {
        Console.WriteLine("sehr gut");
        break;
    }
    case 2: {
        Console.WriteLine("gut");
        break;
    }
    case 3: {
```

```

        Console.WriteLine("befriedigend");
        break;
    }
case 4: {
    Console.WriteLine("ausreichend");
    break;
}
case 5: {
    Console.WriteLine("mangelhaft");
    break;
}
default: {
    Console.WriteLine("Fehler.");
}
}//switch

```

Am Ende der switch-Anweisung darf man noch einen sogenannten default-Block unterbringen, der immer dann ausgeführt wird, wenn keiner der vorigen case- Blöcke zutreffend war. Man kann ihn auch weglassen. Dann wird statt dessen der switch-Block komplett übersprungen. Die switch-Anweisung hat ihre Tücken; **so darf man das break am Ende nicht vergessen.** Man kann das so verstehen: Die case- Zeilen sind Ansprungstellen. Das heißt, wenn jetzt zum Beispiel die Note gleich 2 ist, wird die Zeile mit case 2 angesprungen. Dann läuft das Programm Zeile für Zeile weiter. Wird ein break erreicht, springt das Programm aus dem switch-Block heraus. Haben wir jetzt beispielsweise das break nach case 2 vergessen, dann läuft das Programm einfach Zeile für Zeile weiter, bis der switch-Block zu Ende ist oder ein break erreicht wurde. In unserem Beispiel würde dann

gut befriedigend ausgegeben.

Diesen Effekt kann man durch geschickte Programmierung auch ausnutzen und Zweige für ganze Bereiche definieren. Im folgenden Beispiel wird bei den Noten 1-4 der Text “bestanden” ausgegeben.

```

switch(note) {
case 1:
case 2:
case 3:
case 4: Console.WriteLine("bestanden");
break;
case 5: Console.WriteLine("mangelhaft");
break;
default: Console.WriteLine("Fehler.");
} //switch

```

Eine andere Möglichkeit, Bereiche bei switch-Anweisungen anzugeben, gibt es in C# (anders als z.B. in Pascal) leider nicht. Wenn man das folgende Beispiel mit einer switch-Verzweigung umsetzen wollte, bräuchte man mehrere hundert case-Anweisungen. In diesem Fall greift man besser wieder auf if-else-Kaskade zurück.

```
if (windgeschwindigkeit<=2) {
    Console.WriteLine("Windstille");
} else if (windgeschwindigkeit<=45) {
    Console.WriteLine("schwacher Wind");
} else if (windgeschwindigkeit<=75) {
    Console.WriteLine("starker Wind");
} else if (windgeschwindigkeit<=120) {
    Console.WriteLine("Sturm");
} else if (windgeschwindigkeit<=200) {
    Console.WriteLine("Orkan");
} else {
    Console.WriteLine("Messgeraet kaputt, weil Wind zu
stark");
}
```

Wiederholung

Zählschleife

for Zähler = start to endwert step
Schrittweite

Anweisungsblock

Die Zählschleife ist eine Schleifenart, bei der von Anfang an feststeht, wieviele Wiederholungen der Schleife ausgeführt werden. Es gibt einen Zähler (Laufvariable) der von einem Anfangswert bis zu einem Endwert läuft und sich bei jedem Durchlauf um einen festen Betrag ändert. Das Struktogramm der Zählschleife ist:

for Zähler = start to endwert step
Schrittweite

Anweisungsblock

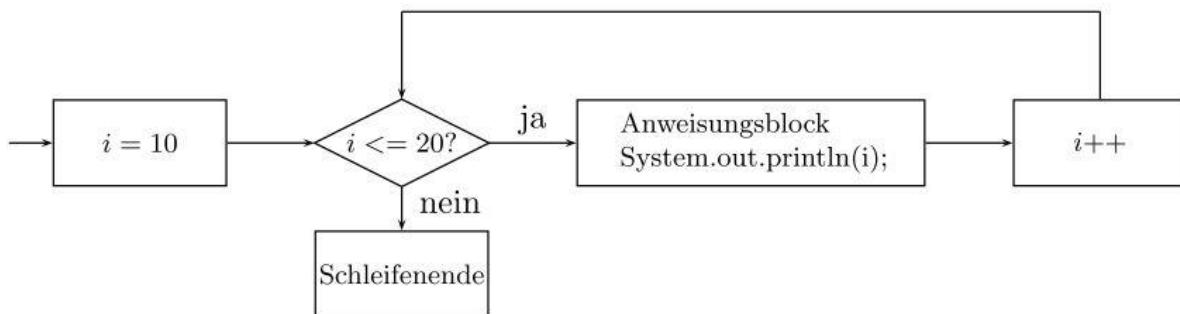
Zählschleifen werden in C# mit dem Schlüsselwort *for* eingeleitet.

```
for (int i=10; i<=20; i++) {  
  
    Console.WriteLine(i);  
  
}
```

Dem Schlüsselwort folgt eine Parameterliste, die in einer runden Klammer zusammengefasst ist und aus 3 Teilen besteht, die jeweils durch ein Semikolon getrennt sind. Die 3 Teile sind:

1. Initialisierung der Laufvariablen.
2. Abbruchbedingung (kein Abbruch, solange die Bedingung erfüllt ist).
3. Veränderung der Laufvariablen.

Die Reihenfolge, in der die Teile abgearbeitet werden, veranschaulicht das folgende Flussdiagramm:



Änderung der Laufvariablen im Schleifenkörper

Es ist möglich, die Laufvariable im Schleifenkörper, also zwischen den geschweiften Klammern, zu ändern. Ein Beispiel dafür ist:

```
for (int i=10; i>=0; i++) {  
    i=i-2;  
    Console.WriteLine(i);  
}
```

Damit ist die Schleife aber keine reine Zählschleife mehr. In anderen Sprachen (z.B. Pascal) ist das auch verboten.

Deklaration der Laufvariablen in der Schleife

Es ist möglich, die Laufvariable im Schleifenkopf zu deklarieren:

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

Dann ist die Laufvariable nur in der Schleife gültig und kann nach Beendigung der Schleife nicht mehr angesprochen werden. Diese Form ist die üblichste Form einer for-Schleife.

Weitere Besonderheiten:

```
for (int i=10; i>=0; i--) ;           #leere Schleife
for(;;)                                #Endlosschleife
for(;;);                               #Leere Endlosschleife
```

Schleife mit Anfangsabfrage (Kopfgesteuerte Schleife)

Bei manchen Schleifen steht zu Anfang die Anzahl der Durchläufe noch nicht fest. Es kann sein, dass es mehrere Abbruchbedingungen gibt oder dass die Laufvariable ihre Werte unvorhersehbar verändern kann. Hier benutzt man entweder die kopfgesteuerte oder die fußgesteuerte Schleife. Die kopfgesteuerte Schleife hat das Aussehen

```
while (Bedingung) {
    Anweisungs-Block
}
```

while Bedingung

Anweisungsblock

Das bedeutet, dass der Anweisungsblock ausgeführt wird, solange die Bedingung in der while-Zeile den Wert true ergibt. Das entsprechende Struktogramm hat das Aussehen:

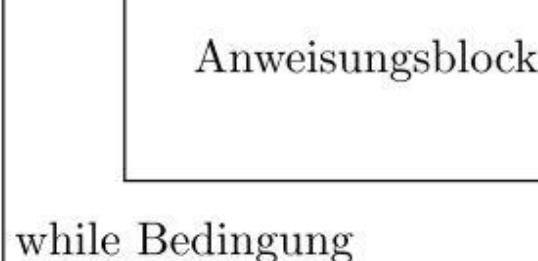
Wir nehmen ein Countdown-Programm, dass von 100 rückwärts bis 10 zählt, aber alle durch 7 teilbaren Zahlen auslässt:

```
int i=100;
while (i>=10) {
    Console.WriteLine(i);
    i--;
    if (i%7==0) {
        i--; //Durch 7 teilbare Zahlen überspringen
    }
}
```

Schleife mit Endabfrage (Fußgesteuerte Schleife)

Die Schleife mit Endabfrage ähnelt der Schleife mit Anfangsabfrage. Allerdings wird die fußgesteuerte Schleife mindestens einmal durchlaufen, während demgegenüber die kopfgesteuerte Schleife gar nicht durchlaufen wird, wenn die Anfangsbedingung vor dem 1. Durchlauf nicht erfüllt ist. Die Schleife mit Endabfrage hat folgendes Aussehen:

```
do {
    //Anweisungsblock
} while (Bedingung)
```



Das Countdown-Beispiel hat hier folgendes Aussehen:

```
int i=10;
do {
    Console.WriteLine(i);
    i--;
} while (i>=0);
```

In C und Java wird diese Schleife auch do-while-Schleife genannt, in Unterscheidung zur kopfgesteuerten while-Schleife. In Pascal spricht man stattdessen von einer repeat-until-Schleife. Da die fußgesteuerte Schleife sich immer mit einer kopfgesteuerten nachbilden lässt, besitzen manche Sprachen (z.B. Fortran, Python) keine fußgesteuerte Schleife.

continue

Innerhalb einer Schleife kann mit dem Befehl continue der aktuelle Schleifen- durchlauf abgebrochen werden, d.h. das Programm wird mit dem Beginn des nächsten Schleifendurchlaufs fortgesetzt. Die continue-Anweisung kann in allen Schleifenvarianten benutzt werden. Das folgende Countdown-Programm ist mit continue so abgewandelt, dass die Zahl 3 ausgelassen wird.

```
for (int i=10; i>=0; i--) {
    if (i==3) {
        continue;
    }
    Console.WriteLine(i);
}
```

break

Der break-Befehl bewirkt, dass eine Schleife komplett abgebrochen wird. Das Programm zählt nur bis zur Zahl 4 herunter.

```
for (int i=10; i>=0; i--) {
    if (i==3) {
        break;
    }
    Console.WriteLine(i);
}
```

break und continue wirken sich nur auf die jeweilige Schleife aus, in der sie definiert wurden.



Mehrere verschachtelte Schleifen

Mehrere verschachtelte Schleifen sind möglich, wie am nachfolgenden Beispiel mehrerer verschachtelter for-Schleifen zu sehen ist. Es gibt ein Dreieck aus Sternen aus:

```
*  
**  
***  
****  
*****  
  
int max=5;  
for (int i=0; i<max; i++) {  
    for (int j=0; j<=i; j++) {  
        System.out.print("*");  
    }  
    System.out.println();  
} //for i
```

Aufgaben

1. Es soll folgendes Spiel durch ein Programm simuliert werden:

Der Spieler zahlt für jedes Spiel einen Einsatz von 1 €. Er darf einmal mit 3 Würfeln würfeln. Je nach gewürfelter Punktzahl bekommt der Spieler einen Gewinn entsprechend der Tabelle ausgezahlt.

Punktzahl	Gewinn in €
3-14	0
15	2
16	5
17	10
18	100

2. Das nachfolgende Programm ist syntaktisch korrekt und könnte somit übersetzt und ausgeführt werden. Es enthält jedoch vier Beispiele für logische Fehler, die bei einem Programmablauf teilweise zu einem Abbruch führen würden. Finden und korrigieren Sie die Fehler ohne Benutzung des Compilers.

```

public class Falsch
{
    public static void Main()
    {
        int x = 0, y = 4;
        //Beispiel A
        if (x < 5)
            if (x < 0)
                Console.WriteLine("x < 0");
        else
            Console.WriteLine("x >=5");

        // Beispiel B
        if (x > 0)
            Console.WriteLine("ok! x > 0");
            Console.WriteLine("1/x = " + (1/x));

        //Beispiel C
        if (x > 0);
            Console.WriteLine("1/x = " + (1/x));

        //Beispiel D
        if (y < x)
        { //vertausche x und y
    
```

```

        x = y;
        y = x;
    }
    Console.WriteLine("x = " + x + " y = " + y);
}
} // Falsch

```

3. Warum ist nachfolgender Code ein Beispiel für schlechten Programmierstil.

Erstellen Sie einen Schreibtischtest für die Ausgabe.

```

public class badSchleife
{
    public static void Main()
    {
        int i, j;
        for (i=1; i<=10; i++)
        { // Schleife A
            Console.WriteLine("A1: i = " + i);
            i = 5;
            Console.WriteLine("A2: i = " + i);
            for (i = 7; i<=20; i++)
            { // Schleife B
                Console.WriteLine("B1: i " + i);
                i = i + 2;
                Console.WriteLine("B2: i " + i);
            }
        }
    }
}

```

4. Welches Ergebnis erzeugt untenstehender Code

```

class SwitchTest
{
    const int EINS = 1;

    public void testSwitch (int zahl)
    {
        switch (zahl)
        {
            case EINS:
            {
                Console.WriteLine("Testergebnis: "+EINS);
                break;
            }
            case 2:
            {
                Console.WriteLine("Testergebnis: "+2);
                break;
            }
        }
    }

    public static void Main (String[] args)
    {
        SwitchTest test = new SwitchTest();
        test.testSwitch (1);
        test.testSwitch (2);
        test.testSwitch (EINS);
    }
}

```

5. Der Algorithmus

1. Lies den Wert von n ein.
2. Setze i auf 3.
3. Solange $i < 2n$, wiederhole
 - Erhöhe i um 1.
 - Gib $\{1\}/\{2i+1\}$ aus

soll auf drei verschiedene Arten implementiert werden.

Schreiben Sie jeweils ein C#-Programmstück, das diesen Algorithmus als **while**, als **for** und als **do while** - Schleife realisiert. Sämtliche Programmstücke sollten die gleiche Ausgabe erzeugen

6. Erstellen Sie ein Programm, das Ganzzahlen beliebigen Wertes annimmt und in der binären Darstellung ausgibt. Benutzen Sie die **do-while**-Schleife zur Ermittlung der Binär-Zahl.
7. Ein Programm soll einen Automaten zur Geldrückgabe simulieren. Das Programm erhält einen Betrag in Form einer Gleitkommazahl als Parameter in der Kommandozeile. Dann soll der Betrag mit möglichst wenig Münzen (Euro) ausgegeben werden.
8. In einer Fabrik für Fahrrad-Nummernschlösser soll der neue Computer die Schließnummern festlegen. Jedes Schloss wird mit einer dreistelligen Nummer geöffnet.

Der Computer soll alle Nummern ausgeben. Fahrradschlösser mit zwei oder drei gleichen Ziffern nimmt die Kundschaft nicht ab, also darf der Computer sie nicht aufschreiben. Damit nicht so viel Platz verbraucht wird, soll der Computer jeweils 10 Nummern nebeneinander schreiben, etwa so.

```
012 013 014 015 016 017 018 019 021 023
024 025 026 027 028 029 031 032 034 035
036 037 038 039 041 042 043 045 046 047
048 049 051 052 053 054 056 057 058 059
061 062 063 064 065 067 068 069 071 072
073 074 075 076 078 079 081 082 083 084
085 086 087 089 091 092 093 094 095 096
097 098 102 103 104 105 106 107 108 109
120 123 124 125 126 127 128 129 130 132
```

9. Der Text von Edsger W. Dijkstra **Go To Statement Considered Harmful** ist ein 'Klassiker' der EDV-Literatur:
 1. Welche Aussagen trifft der Text
 2. Wie muss man sich den von Dijkstra kritisierte Programmiercode vorstellen. Finden Sie entsprechende Beispiele.
 3. Finden Sie im Web entsprechende Gegenmeinungen zu Dijkstra.

10. Hier Fragen aus der Zwischenprüfung Frühjahr 2007

3.6

Der Lieferant gibt auf einige Artikel, die als Haussortiment gekennzeichnet sind, Rabatte von 10 %. Des Weiteren gibt es ab einem Bestellwert von 200,00 € zusätzlich 15 % bzw. ab einem Bestellwert von über 100,00 € 5 % Rabatt. Lediglich bei einem Bestellwert von weniger als 20,00 € wird eine Pauschale von 3,50 € für Verpackung und Versand berechnet.

Die Daten einer Bestellung werden in einer Tabelle gespeichert, deren Struktur dem abgebildeten Auszug entspricht:

Auszug aus der Tabelle Bestellung

Bestellposition	Artikelbezeichnung	Haussortiment	Preis	Menge
010	Radiergummi	True	0.85	2

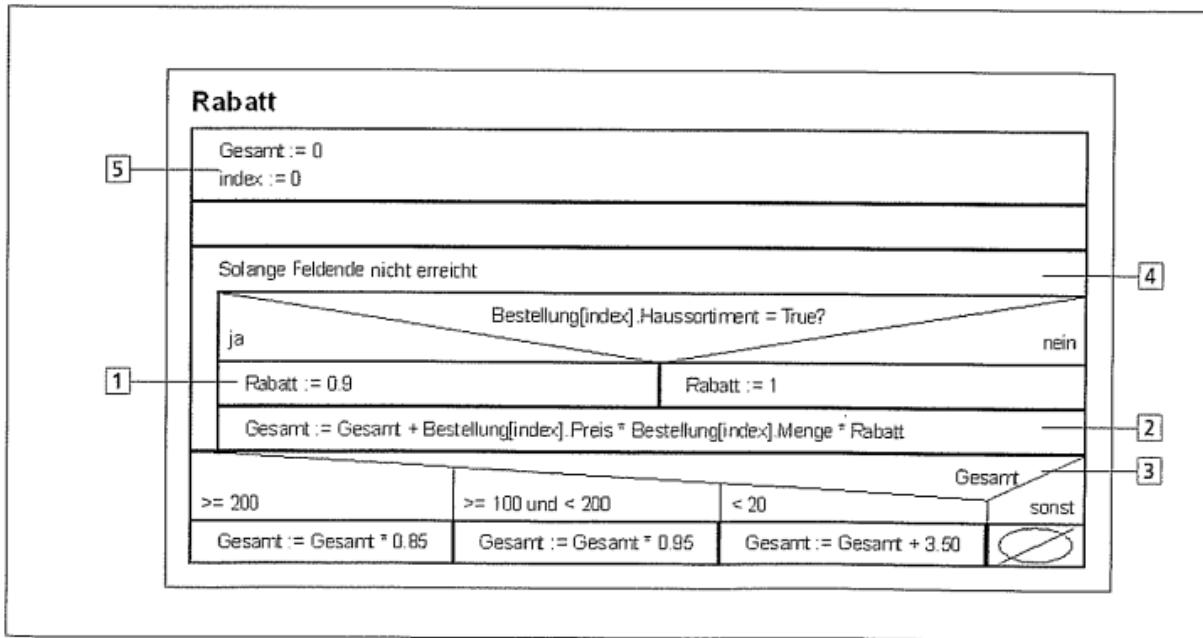
Die Tabellenelemente werden wie folgt angesprochen:

Bestellung[index].spaltenname

Der Index startet mit 0.

Sie prüfen das abgebildete Struktogramm zur Ermittlung des Bestellwerts auf seine Korrektheit. Bestimmen Sie die Position des logischen Fehlers!

1. Der angegebene Rabattsatz entspricht nicht den Vorgabewerten aus der Ist-Analyse.
2. Die Berechnung des Bestellwerts ist fehlerhaft, daher stimmt auch die Gesamtsumme nicht.
3. Es werden nicht alle Fälle bei der Mehrfachauswahl berücksichtigt.
4. Es werden nicht alle Datensätze verarbeitet, da die Index-Variable nicht erhöht wird.
5. Die Initialisierung der Variablen Gesamt ist fehlerhaft.



Frühjahr 2006

Um die Ausführungsgeschwindigkeit des Programms zu erhöhen, halten Sie die aktuellen Daten komplett im Arbeitsspeicher. Sie verwenden dazu ein Array. Um ein bestimmtes Datum zu finden, müssen Sie eine Suche programmieren. Sie verwenden die binäre Suche, da diese bei einem bereits sortierten Feld gut genutzt werden kann. Das Prinzip der binären Suche beruht auf folgender Vorgehensweise:

- Zuerst wird das mittlere Element der Datenmenge untersucht. Ist dieses größer als das gesuchte Element, muss nur noch in der unteren Hälfte gesucht werden, andernfalls in der oberen.
- Bei jedem Schritt halbiert sich die Menge der Elemente, die betrachtet werden muss.
- Die Suche ist beendet, wenn das Element gefunden wurde, oder nur noch ein Element übrig ist. Entweder ist dieses Element das gesuchte, oder das gesuchte Element kommt nicht vor.

Weitergehende Aufgaben

1. Übungsaufgabe Kammerprüfung

Sommer 2005, HS 5

Die Webanwendung der Media HO GmbH speichert nach jeder Sitzung auf dem PC des Benutzers einen Cookie, der die Artikelnummern der Artikel enthält, die der Benutzer zuletzt angesehen hat. Die Artikelnummern sind in einer Zeichenkette gespeichert.

Bsp: 2105607105535_2105607105538_2105607105537

Zu Sitzungsbeginn wird der Cookie gelesen.

Die Systemsoft GmbH soll einen Algorithmus erstellen, der alle Artikelnummern aus dem Cookie extrahiert und die zu den Artikelnummern gehörigen Artikelobjekte in eine Artikeliste speichert.

Vorgaben:

In der Zeichenkette "favoriten" wird der Wert eines Cookies gespeichert

Die Variable list enthält die Referenz auf ein Artikellisten-Objekt

Folgende Methoden sollen verwendet werden.

Klasse	Methode	
String	indexOf(String s)	- Sucht in einem String nach dem Teilstring s und liefert die Position, an der s gefunden wurde - Wird der Teilstring s nicht gefunden, wird -1 zurückgegeben
	indexOf(Integer pos, String s)	- Sucht in einem String nach dem Teilstring s und liefert die Position, an der s gefunden wurde - Beginnt die Suche an der Stelle pos - Wird der Teilstring s nicht gefunden, wird -1 zurückgegeben
	subString(Integer p1, Integer p2)	- Liefert einen Teilstring von der Position p1 bis zur Position p2 (exklusiv)
DBTool	getArtikel(String artikelnummer)	- Statische Methode - Erstellt zur übergebenen Artikelnummer ein Objekt vom Typ Artikel - Liefert eine Referenz auf dieses Artikelobjekt
Artikelliste	add(Artikel a)	- Fügt einer Artikelliste das übergebene Artikelobjekt a hinzu

5. Handlungsschritt (20 Punkte)

Variable	Datentyp	Beschreibung
pos1, pos2	Integer	Variablen, in denen die Positionen eines „_“-Zeichen und des nächst folgenden „_“-Zeichen gespeichert wird
artikelnummer	String	Eine im Cookie enthaltene Artikelnummer
a	Artikel	Referenz auf einen Artikel
favoriten	String	Wert des Cookies
liste	Artikelliste	Referenz auf ein bereits erstelltes Artikelliste-Objekt

```
pos1 := 0
pos2 := favoriten.indexOf("_")
Solang (pos2 >= 0)
  artikelnummer := favoriten.substring(pos1, pos2)
  Artikel a := DBTool.getArtikel(artikelnummer)
  liste.add(a)
  pos1 := pos2 + 1
  pos2 := favoriten.indexOf(pos1, "_")
```

Sommer 2005, HS 6

6. Handlungsschritt (20 Punkte)

Alle Artikel der Media-HO GmbH werden mit der Europäischen Artikel Nummer (EAN) gekennzeichnet. Die Systemsoft GmbH soll für eine Kontrollroutine eine Funktion schreiben, die die Prüfziffer berechnet.

Aufbau des EAN-Code

Stellen 1 bis 12: Artikelnummer

Stelle 13: Prüfziffer

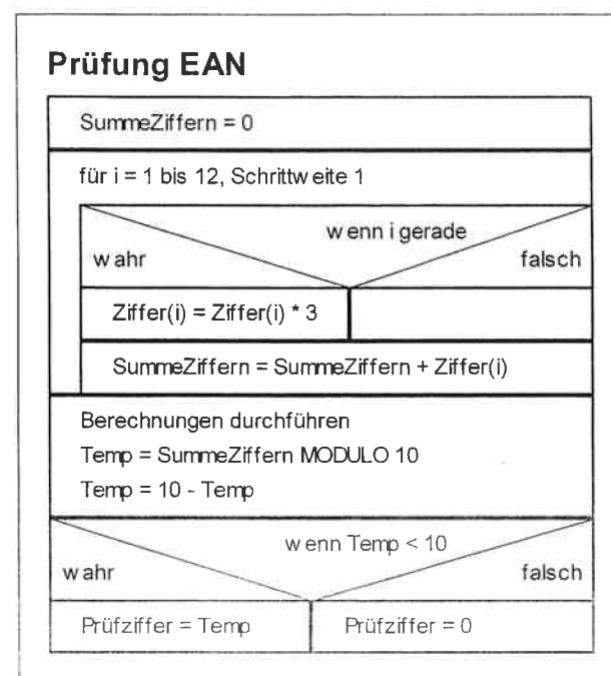
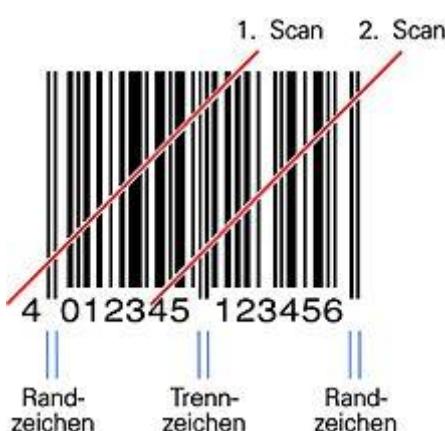
Berechnung der Prüfziffer

Die zwölf Ziffern der Artikelnummer werden von links nach rechts addiert. Vor der Addition werden die Ziffern an den geraden Stellen mit 3 multipliziert. Die Summe wird durch 10 dividiert. Der Rest wird als ganze Zahl von 10 subtrahiert. Die Einerstelle der Differenz ist die Prüfziffer.

Ein zu kontrollierender EAN-Code ist in der eindimensionalen Tabelle „Ziffer“ gespeichert. Jedes Tabellenelement ist mit einer EAN-Code // Ziffer belegt.

Stellen Sie die Logik zur Berechnung der Prüfziffer in einem Struktogramm dar.

6. Handlungsschritt (20 Punkte)



Methoden und Funktionen

Download:

- Siehe **Übung**.
- Siehe: <http://www.jamesshore.com/Articles/Quality-With-a-Name.html>

Grundlagen

Programme bestehen fast nie aus einem einzigen Programmblöck, sondern die Kunst des Programmierens besteht darin, sinnvolle Untereinheiten zu bilden, die insgesamt gesehen das Programmierproblem lösen.

Aufgabe: Was ist unter diesem Aspekt am untenstehenden Beispiel nicht optimal?

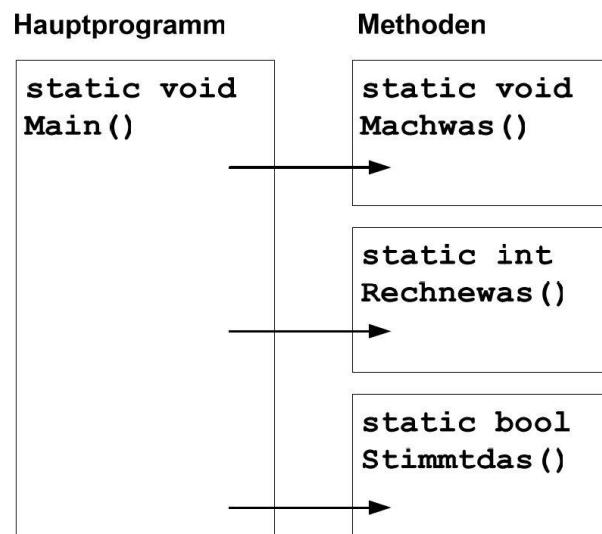
```
static void Main(string[] args)
{
    int zahl1 = int.Parse(Console.ReadLine());
    int zahl2 = int.Parse(Console.ReadLine());
    int zahl3 = int.Parse(Console.ReadLine());
    int zahl4 = int.Parse(Console.ReadLine());
    int zahl5 = int.Parse(Console.ReadLine());
    int Summe = zahl1 + zahl2 + zahl3 + zahl4 + zahl5;

    Console.WriteLine("Vorher:" + Summe);
    int Umsatzsteuer = Summe*16/100;
    int GesamtSumme = Summe + Umsatzsteuer;

    Console.WriteLine("Gesamtsumme gleich " + GesamtSumme);
}
```

- Auslagern von Funktionalität in Form von
 - Unterprogrammen
 - Funktionen

- Gründe (Auswahl)
 - Wiederverwendbarkeit
 - Wartbarkeit
 - Übersichtlichkeit
 - Kapselung
 - Modularisierung
 - ...



Die Aufteilung von Aufgaben wird auf der untersten Ebene mit Hilfe von **Methoden** vorgenommen. Die sinnvolle Kombination und die Zusammenarbeit unter den Methoden sorgt für den Programmfluss.

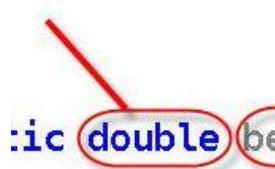
Der grundsätzliche Aufbau einer Methode orientiert sich dabei an mindestens drei verschiedenen Gesichtspunkten

```

static double berechneUmsatz(double Nettowert, double Prozentsatz)
{
}
  
```

Rückgabewert einer Methode

Methoden können an ihre Nutzer Ergebnisse von Berechnungen **zurückliefern oder auch nicht**. Man spricht vom sog. *Rückgabewert* einer Methode.

 Für den Nutzer der Methode ist dabei natürlich wichtig, von welchem Datentyp der Rückgabewert ist, weil er dafür eine Variable bereitstellen muss. Die Methode stellt aus diesem Grund den Datentyp des Rückgabewertes vor ihren Methodennamen.

Will die Methode keinen Wert an ihre Nutzer zurückgeben, so schreibt man vor den Methodennamen das Schlüsselwort **void**.

Anwendung Umsatzsteuerberechnung

Um die Umsatzsteuerberechnung als eigene Methode auszulagern, kann man nun wie folgt vorgehen.

- Schreiben einer Methode berechneUmsatzSteuer
- Rückgabewert der Methode soll vom Typ **void** sein
- Als Ausgangswert erhält die Methode einen Parameter vom Typ **double**.
- berechneUmsatzSteuer() soll den Wert berechnen und anschließend selbst die Gesamtsumme ausgeben.

```
static void berechneUmsatz(double Nettowert)
{
    double Umsatzsteuer = Nettowert*16/100;
    double GesamtSumme = Nettowert + Umsatzsteuer;

    Console.WriteLine("Gesamtsumme gleich " + GesamtSumme);
}
```

Verbesserung der Lösung

Die vorherige Lösung nimmt zwar Quellcode aus der Methode main() heraus, führt aber ein Wartungsproblem hinzu. Welches????

Antwort

Die Methode berechneUmsatzsteuer() macht immer noch 2 Dinge, nämlich das Berechnen der Umsatzsteuer und die Ausgabe des Gesamtbetrages.

Sie ist damit nicht beliebig einsetzbar

Die Methode berechneUmsatzsteuer() darf nur die Umsatzsteuer berechnen und das Ergebnis zurückgeben. Die Ausgabe soll wiederum in eine eigene Methode ausgelagert werden.

- Die Methode **berechneUmsatzsteuer** erhält einen Rückgabewert vom Typ double. Sie liefert nur die tatsächliche Umsatzsteuer an main() zurück.
- main() berechnet dann die Gesamtsumme und ruft dann ausgabeGesamtsumme() auf.

Als Parameter wird der Wert von Gesamtsumme übergeben.

```
//Auslagern der Umsatzsteuerberechnung
//und Ausgabe der Berechnung

    }

    static double berechneUmsatzsteuer(double Nettowert)
    {
        double Umsatzsteuer = Nettowert*16/100;
        return Umsatzsteuer;
    }

    static void ausgabeGesamtSumme(double GesSumme)
    {
        Console.WriteLine("Gesamtsumme gleich " + GesSumme);
    }
}
```

Parameter einer Methode

Neben dem bereits angesprochenen Rückgabewert kann eine Methode eine Liste an Werten aufnehmen, mit denen es dann innerhalb der Methode weiterarbeiten kann. Die Werte werden innerhalb der Klammer in der Form **Datentyp Variablenname** übergeben. Mehrere Parameter werden durch Komma getrennt **Parameterliste**.

```
static double berechneUmsatz(double Nettowert, double Prozentsatz)
{
}
```

Die Variablen innerhalb der Klammer sind **lokal** und stehen nur innerhalb der Methode zur Verfügung.

Die Werte für diese lokalen Variablen stehen häufig in Variablen der aufrufenden Methoden. Deshalb stellt sich die Frage, wie eine Veränderung in der ausgelagerten Methode die Werte in der Nutzermethode beeinflusst.

Nebeneffekte bei Methodenaufrufen

Was ist das Ergebnis des folgenden Quellcodes ?

```
public static void Main()
{
    double Summe = 50.00;
    //Ein Array mit 5 Feldern von 0 bis 4 mit double-Werten
    double[] SummenArray = new double[]{1,2,3,4,5};

    Console.WriteLine("Summe = " + Summe);
    Console.WriteLine("SummenArray[0] = " + SummenArray[0]);

    berechneUmsatzsteuer(Summe, SummenArray);

    Console.WriteLine("Summe = " + Summe);
    Console.WriteLine("SummenArray[0] = " + SummenArray[0]);
}

public static void berechneUmsatzsteuer(double Umsatz, double[] Summe)
{
    Umsatz = Umsatz * 2;
    Summe[0] = Umsatz;
}
```

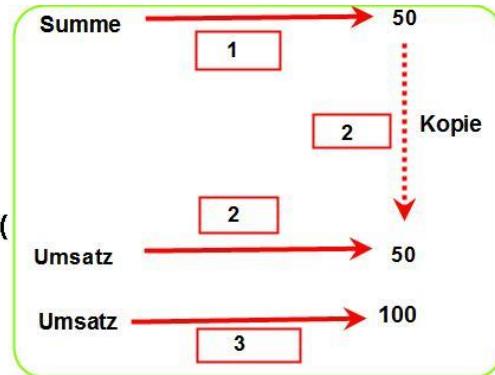
Während sich der Wert der Variable Summe nicht verändert hat, hat sich der Wert des Arrays an der Stelle 0 verändert. Dies ist umso bemerkenswerter, da beide offensichtlich den gleichen Datentyp besitzen.

Übergabe als Kopie (by val)

Einfache Datentypen (auch string) werden beim Aufruf einer Methode per default als Kopie übergeben.

```
double Summe = 50.00;    1
berechneUmsatzsteuer(Summe);
}

public static void berechneUmsatzsteuer(
    double Umsatz)
{
    Umsatz = Umsatz * 2;    3
}
```

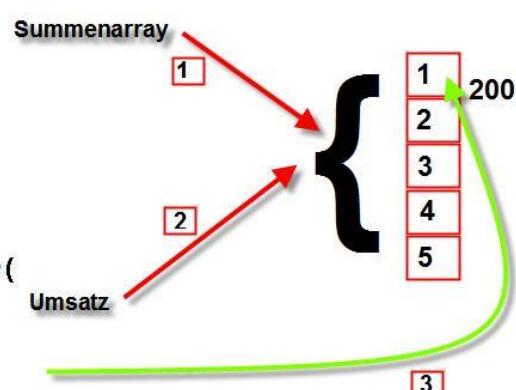


Übergabe als Referenz (by ref)

Komplexe Datentypen (Objekte) werden **nicht** kopiert und dann an die Methode übergeben, sondern beide Variablen zeigen weiterhin auf den gleichen Wertebereich im Speicher. Deshalb verändert eine Manipulation der 2. Variable auch den Wert der 1. Variable. Dies war beim Array der Fall.

```
double[] Summenarray =
    new double[]{1,2,3,4,5};
berechneUmsatzsteuer(Summenarray);
}

public static void berechneUmsatzsteuer(
    double[] Umsatz)
{
    Umsatz[0] = 200;
}
```



Überladen von Methoden

Häufig ist es sinnvoll, dass eine Methode durch einen Nutzer unter dem gleichen Namen, aber mit verschiedenen Wertetypen aufgerufen werden kann. Alle Frameworks machen sehr häufig Gebrauch davon, z.B.

```
Console.WriteLine()
```

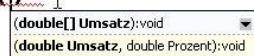
Man spricht dabei von einer **überladenen Methode**

Praktisch wirkt sich das so aus, dass im Quellcode mehrere Male der gleiche Methodenkopf erstellt und ausprogrammiert wird.

```
double[] Summenarray =
    new double[]{1,2,3,4,5};
berechneUmsatzsteuer(Summenarray);
berechneUmsatzsteuer()
}

public static void berechneUmsatzsteuer( double[] Umsatz )
{
    Umsatz[0] = 200;
}

public static void berechneUmsatzsteuer(double Umsatz, double Prozent)
{
    double Steuer = Umsatz/16 * 100;
}
```



Jeder Methodenname kann beliebig überschrieben und damit an die Bedürfnisse der Nutzer angepasst werden. Allerdings sind einige Bedingungen einzuhalten

- Jede Methode muss sich in der Zahl und/oder der Art des Datentyps unterscheiden
- Unterschiedliche Variablennamen sind nicht ausreichend

```

    berechneUmsatzsteuer()
}

public static void berechneUmsatzsteuer(double[] Umsatz)
{
    Umsatz[0] = 200;
}

public static void berechneUmsatzsteuer(double Umsatz, double Prozent)
{
    double Steuer = Umsatz/16 * 100;
}

public static void berechneUmsatzsteuer(double[] Umsatz, double Prozent)
{
}

public static void berechneUmsatzsteuer(double Umsatz)
{
}

public static void berechneUmsatzsteuer(double Netto)
{
}

public static void berechneUmsatzsteuer(int Umsatz, double Prozent)
{
    double Steuer = Umsatz/16 * 100;
}

public static void berechneUmsatzsteuer(int Umsatz, int Prozent)
{
    double Steuer = Umsatz/16 * 100;
}

```

Parameter-Variationen: ref / out

Wie bereits erwähnt, werden die Werte einfacher Datentypen beim Aufruf einer Methode kopiert. Die Originalwerte bleiben deshalb immer erhalten.

Dieses Verhalten kann mit Hilfe der Schlüsselworte **ref** und **out** beeinflusst werden.

Parameterübergabe durch **ref**

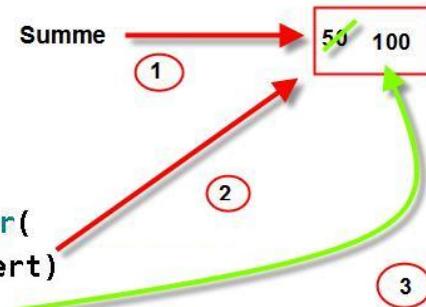
Das Festlegen des **ref**-Schlüsselworts für Methodenparameter führt dazu, dass eine Methode auf dieselbe Variable verweist, die in die Methode übergeben wurde. Alle Änderungen des Methodenparameters werden in diese Variable übernommen, sobald die Steuerung wieder an die aufrufende Methode übergeben wird.

```

        double Summe = 50.00;
        berechneUmsatzsteuer(ref Summe);
        Console.WriteLine(Summe);
    }

    public static void berechneUmsatzsteuer(
        ref double GesamtWert)
    {
        GesamtWert = 100;
    }

```



Parameterübergabe durch out

Das Festlegen des out-Schlüsselworts für Methodenparameter führt dazu, dass eine Methode auf dieselbe Variable verweist, die in die Methode übergeben wurde. Alle Änderungen des Methodenparameters werden in diese Variable übernommen, sobald die Steuerung wieder an die aufrufende Methode übergeben wird.

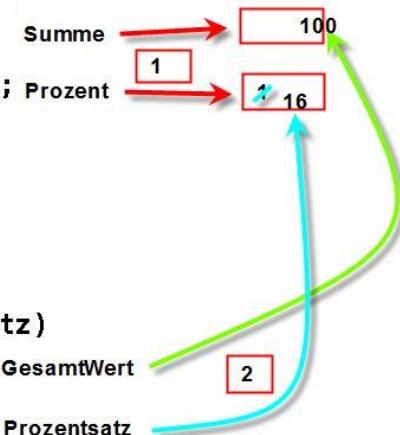
Das Deklarieren einer out-Methode ist nützlich, wenn eine Methode mehrere Werte zurückgeben soll. Eine Methode, die einen out-Parameter verwendet, kann immer noch einen Wert zurückgeben. Eine Methode kann mehr als einen out-Parameter enthalten.

Um einen out-Parameter verwenden zu können, muss das Argument explizit als out-Argument an die Methode übergeben werden. Der Wert des out-Arguments wird nicht an den out-Parameter übergeben.

Eine Variable, die als out-Argument übergeben wurde, muss nicht initialisiert sein. Dem out-Parameter muss jedoch ein Wert zugewiesen werden, bevor die Methode zurückgegeben wird.

```
//keine Initialisierung der Variable notwendig,
double Summe;
int Prozent=1; // aber möglich
berechneUmsatzsteuer(out Summe, out Prozent);
Console.WriteLine(Summe);
Console.WriteLine(Prozent);
}
```

```
//Rückgabe mehrerer Werte per out
public static void berechneUmsatzsteuer(
    out double GesamtWert, out int Prozentsatz)
{
    GesamtWert = 100;
    Prozentsatz = 16;
}
```



Auswertungsreihenfolge bei Methoden

Was ist das Ergebnis des folgenden Quellcodes

```
class Auswertung
{
    public static void Main (String[] args)
    {
        int aktuell = 1;
        methode (aktuell++, aktuell);
        Console.WriteLine ("Nach Methodenaufruf:");
        Console.WriteLine ("Wert von aktuell: " + aktuell);
    }

    public static void methode(int formalA, int formalB)
    {
        Console.WriteLine ("Innerhalb der Methode:");
        Console.WriteLine ("Wert von formalA: " + formalA);
        Console.WriteLine ("Wert von formalB: " + formalB);
    }
}
```

```
1 using System;
2
3 public class Class2
4 {
5     public static void Main (String[] args)
6     {
7         int aktuell = 1;
8         methode (aktuell++, aktuell);
9         Console.WriteLine ("Nach Methodenaufruf:");
10        Console.WriteLine ("Wert von aktuell: " + aktuell);
11    }
12
13     public static void methode(int formalA, int formalB)
14     {
15         Console.WriteLine ("Innerhalb der Methode:");
16         Console.WriteLine ("Wert von formalA: " + formalA);
17         Console.WriteLine ("Wert von formalB: " + formalB);
18     }
19 }
```

Lösung

Beim Aufruf der Methode `methode()` laufen folgende Zuweisungen ab:

```
formalA = aktuell++;
formalB = aktuell;
```

Als aktuelle Werte werden die Rückgabewerte der Ausdrücke `aktuell++` und `aktuell` an die formalen Parameter der Methode `methode()` zugewiesen. In C# werden die aktuellen Parameter von links nach rechts ausgewertet.

Zuerst wird also der erste aktuelle Parameter ausgewertet. Der Rückgabewert 1 des Ausdrucks `aktuell++` wird dem ersten formalen Parameter zugewiesen.

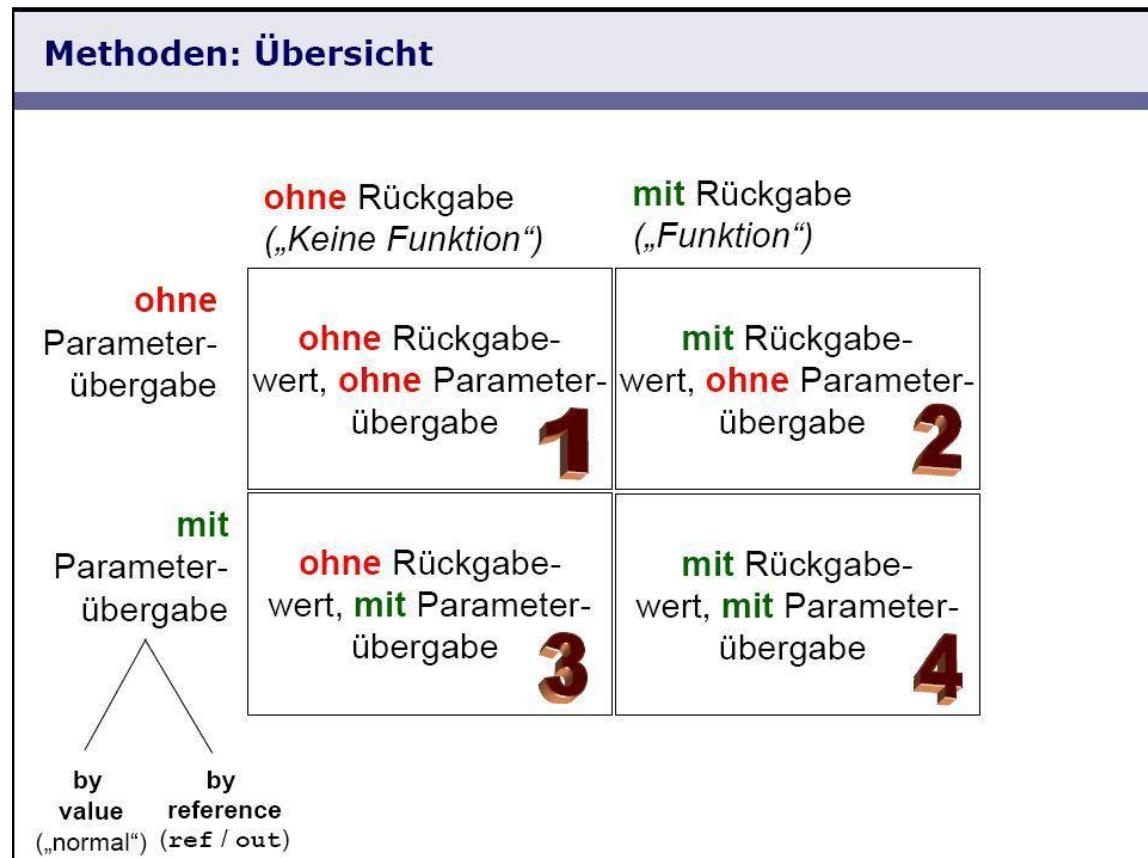
Nach der Bewertung des ersten aktuellen Parameters hat die Variable `aktuell` den Wert 2. Dieser Wert wird dem zweiten formalen Parameter zugewiesen.

Sichtbarkeit von Methoden

Als letztes Element kann innerhalb eines Methodenkopfes die Sichtbarkeit definiert werden.

- public: Öffentliche Methoden, die von überall aufgerufen werden können
- privat: Methoden, die nur innerhalb einer Klasse benutzbar sind
- # protected: Methoden, die innerhalb von Vererbung aufrufbar sind bzw. innerhalb des gleichen Namensraumes genutzt werden können.

Zusammenfassung Methode



Aufgaben

1. Übersetzen Sie die folgenden Aussagen !

The programmer can write methods to define specific tasks that may be used at many points in a program. These methods sometimes are referred to as programmer-defined methods.

The actual statements defining the method are written only once and are hidden from other methods.

Methods are called by writing the name of the method (sometimes preceded by the class name and a dot operator), followed by a left parenthesis, the method's argument (or a comma-separated list of arguments) and a right parenthesis.

All variables declared in method definitions are local variables— they are known only in the method in which they are defined.

Packaging code as a method allows that code to be executed from several locations in a program when the method is called.

The return statement in a method passes the results of the method back to the calling method.

The first line of a method definition is sometimes known as the method header. The attributes and modifiers in the method header are used to specify information about the method.

The method return-value-type is the data type of the result that is returned from the method to the caller. Methods can return one value at most.

The parameter-list is a comma-separated list containing the declarations of the parameters received by the called method. There must be one argument in the method call for each parameter in the method definition.

The declarations and statements within the braces that follow the method header form the method body.

Variables can be declared in any block, and blocks can be nested.

A method cannot be defined inside another method.

In many cases, an argument value that does not correspond precisely to the parameter types in the method definition is converted to the proper type before the method is called.

When an argument is passed by value, a copy of the argument's value is made and passed to the called method.

With pass-by-reference, the caller enables the called method to access the caller's data directly and to modify that data if the called method chooses

Several methods can have the same name, as long as these methods have different sets of parameters, in terms of number of parameters, types of the parameters and order of the parameters. This is called method overloading.

Method overloading commonly is used to create several methods with the same name that perform similar tasks, but on different data types.

2. Erstellen Sie eine Applikation, die Ihnen das Ergebnis der Multiplikation einer Zahl mit sich selbst zurückliefert. Die Applikation soll dies sowohl für double- als auch für integer-Zahlen ermöglichen.

Lagern Sie die Multiplikation in eine eigene Methode aus. Rufen Sie dann die Methode aus der Main-Methode heraus auf. Schreiben Sie überlagerte Multiplikationsmethoden für jeden Datentyp.

3. Finden Sie den Fehler in jedem der folgenden Programmsegmente und erklären Sie, wie der Fehler korrigiert werden kann.

```
int g() {
    Console.WriteLine( "Inside method g" );
    int h() {
        Console.WriteLine( "Inside method h" );
    }
}
```

```
int sum( int x, int y ) {
    int result;
    result = x + y;
}
```

```
int sum( int n ) {
    if ( n == 0 )
        return 0;
    else
        n + sum( n - 1 );
}
```

```
void f( float a ); {  
    float a;  
    Console.WriteLine( a );  
}
```

```
void product() {  
    int a = 6, b = 5, c = 4, result;  
    result = a * b * c;  
    Console.WriteLine( "Result is " + result );  
    return result;  
}
```

4. Welchen Parametertyp/Methodentyp würden Sie für folgende Problemstellungen wählen

- Anruf bei der Telefonauskunft:
“Können Sie mir bitte die Postleitzahl für die Garystraße 21 in Berlin geben?”
- Der Vater zu seinem Sohn:
„Hier, Junge, da hast Du mein Auto, kannst du das mal bitte waschen und tanken?“
- Auf einer Party: “ey, bring mir mal Bier und Chips mit!”



5. Lagern Sie folgende lange Methode in mehrere kleinere Methoden aus.

```

static void Wochentag()
{
    int t, m, j, MerkeMonat;
    Console.WriteLine("Geben Sie den Tag ein");
    t = int.Parse(Console.ReadLine());
    Console.WriteLine("Geben Sie den Monat ein");
    m = int.Parse(Console.ReadLine());
    MerkeMonat = m;
    Console.WriteLine("Geben Sie das Jahr ein");
    j = int.Parse(Console.ReadLine());
    if(m <= 2)
    {
        m += 10;
        j -= 1;
    }
    else
    {
        m -= 2;
    }
    int c = j / 100;
    int y = j % 100;
    int h = (((26 * m - 2) / 10) + t + y + y / 4 + c / 4 - 2 * c) % 7;
    if(h < 0)
        h += 7;
    string Tag = "";
    switch(h)
    {
        case 0:
            Tag = "Sonntag";
            break;
        case 1:
            Tag = "Montag";
            break;
        case 2:
            Tag = "Dienstag";
            break;
        case 3:
            Tag = "Mittwoch";
            break;
        case 4:
            Tag = "Donnerstag";
            break;
        case 5:
            Tag = "Freitag";
            break;
    }
}

```

```
case 6:  
    Tag = "Samstag";  
    break;  
}  
Console.WriteLine("Der " + t + "." + MerkeMonat + "." + j + " ist  
ein " + Tag);  
}
```

Komplexe Datentypen

[MSDN Teil6](#)

Arrays

Werte	Index
34	0
81	1
106	2
3	3
56	4
76	5
1234	6
343	7
65	8
42	9

```
int[] Werte = new int[10];
Console.WriteLine(Werte[7].ToString());
```



Arrays sind in der Lage, mehrere Werte eines Datentyps unter einem Bezeichner zu speichern. Sie haben in C# eine feste Größe. Die Elemente werden über einen Index angesprochen.

Der Datentyp eines Arrays kann beliebig sein, ein einmal gewählter Typ legt aber den Datentyp für den kompletten Array fest. Ein Array kann auch aus structs oder Klassen gebildet werden. Unabhängig welcher Datentyp gewählt wird, ist der Array immer ein Speicher von Daten des Typs object, d.h. er ist selbst ein sogenannter **reference type**. Alle Verweise auf Array-Elemente arbeiten damit immer auf dem Original.

Syntax einer Arraydeklaration

```
int[] Arrayname1;
int[] Arrayname2 = {5,7,9,3};
int[] Arrayname3 = new int[9];
int[] Arrayname4 = new int[]{1,4,7,3,1};
int[,] Arrayname5;
int[,] Arrayname6 = {{3,1},{1,2}};
int[,] Arrayname7 = new int[1,1];
int[,] Arrayname8 = new int[,]{{1,2},{3,4},{5,6},{7,8}};
```

- Grundsätzlich können Arrays **ohne oder mit new**-Operator deklariert werden.
- Der Deklaration können Zugriffsmodifizierer (public, private, etc) vorangestellt werden.
- Die Arraydeklaration unterscheidet sich von der einfachen Variablen-deklaration durch das Hinzufügen der eckigen Klammern
- Innerhalb der eckigen Klammer kann die Angabe der Arraylänge erfolgen.
- Bei der Deklaration mehrdimensionaler Arrays werden die Dimensionen durch Kommas getrennt

Vollziehen Sie die untenstehenden Anweisungen schrittweise nach; welche Werte sind jeweils vorhanden

```
int i = 2;  
  
int[] A;  
-----  
  
A = new int [4];  
-----  
A [0] = 8; A [1] = 7;  
  
A [i] = 9; A [3] = 6;  
-----  
  
A = new int [3];  
-----  
int [] B;  
  
B = A;  
-----  
A[0] = 6;  
  
B [1] = 7;  
  
B [2] = B [0] + 2;  
-----  
i = B [0];  
  
A = new int [5];  
-----  
A [i - 2] = B [1];  
  
B [i - 4] = A.length;
```



Zugriff auf ArrayElemente

Um auf ein Element zuzugreifen verwendet man den Namen des Arrays und die jeweilige Indexstelle in eckigen Klammern.

```
int[] vArray = new int[3];
vArray[2] = 5;
Console.WriteLine(vArray[0].ToString());
```

Den Durchlauf durch alle Elemente kann man einerseits wie folgt erreichen

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

Anderseits bietet .net für sog. Collections das foreach-Konstrukt an, mit der auf einfache Art und Weise jede Liste durchlaufen werden kann.

```
using System;

public class Primzahl
{
    public static void Main()
    {
        int[] Primzahl = new int[99];
        int i;
        int j;
        bool IstPrimzahl;

        foreach(int Zahl in Primzahl)
        {
            if(Zahl == 2 || Zahl == 3)
            {
                Console.Write("{0} ", Zahl);
            }
        }
    }
}
```

Die Syntax der foreach-Schleife lautet also foreach(Datentyp Variablenname in Arrayname) { Anweisungsblock; }

Wichtige Methoden/Eigenschaften im Zusammenhang mit Arrays

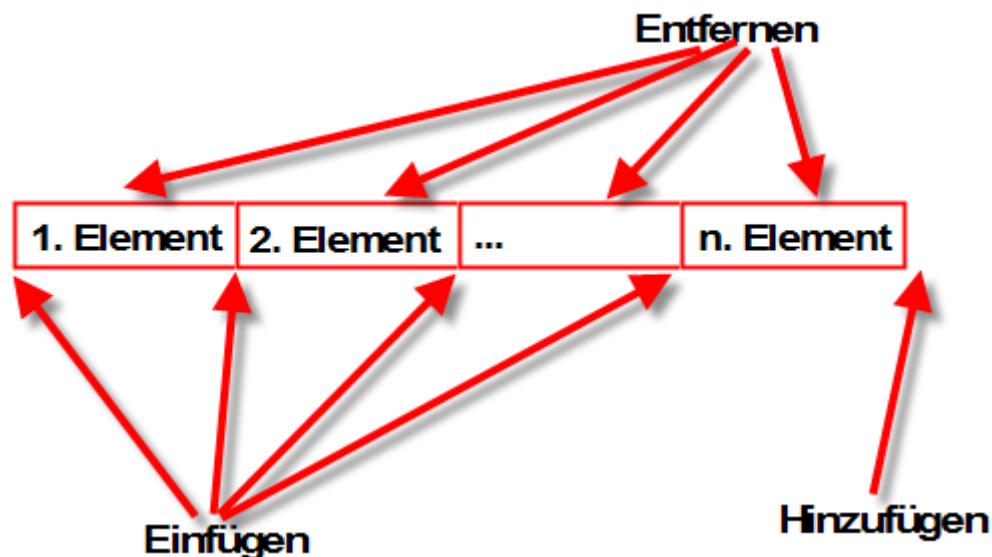
ar.GetType();	
i = ar.Rank();	
GetUpperBound() GetLowerBound()	
Length() GetLength()	
Clear(); System.Array.Clear(ar, 0,23);	
Reverse() System.Array.reverse(ar);	
SetValue() ar.SetValue(123,2)	
Sort() System.Array.Sort(ar);	
int i = System.Array.BinarySearch(ar, 123)	

ArrayLists

ArrayLists sind sogenannte **Collections**, die wie der Array auch eine Vielzahl von Objekten unter einem gemeinsamen Namen verwalten können.

Im Gegensatz zum Array muss die Aufnahmekapazität der Liste jedoch nicht zu Beginn bekannt sein, sondern sie kann sich dynamisch den Erfordernissen anpassen.

Weiterhin können einer ArrayList Elemente an beliebigen Stellen entnommen werden, ohne dass Leerstellen entstehen. Die verbleibenden Elemente rücken wieder zusammen.

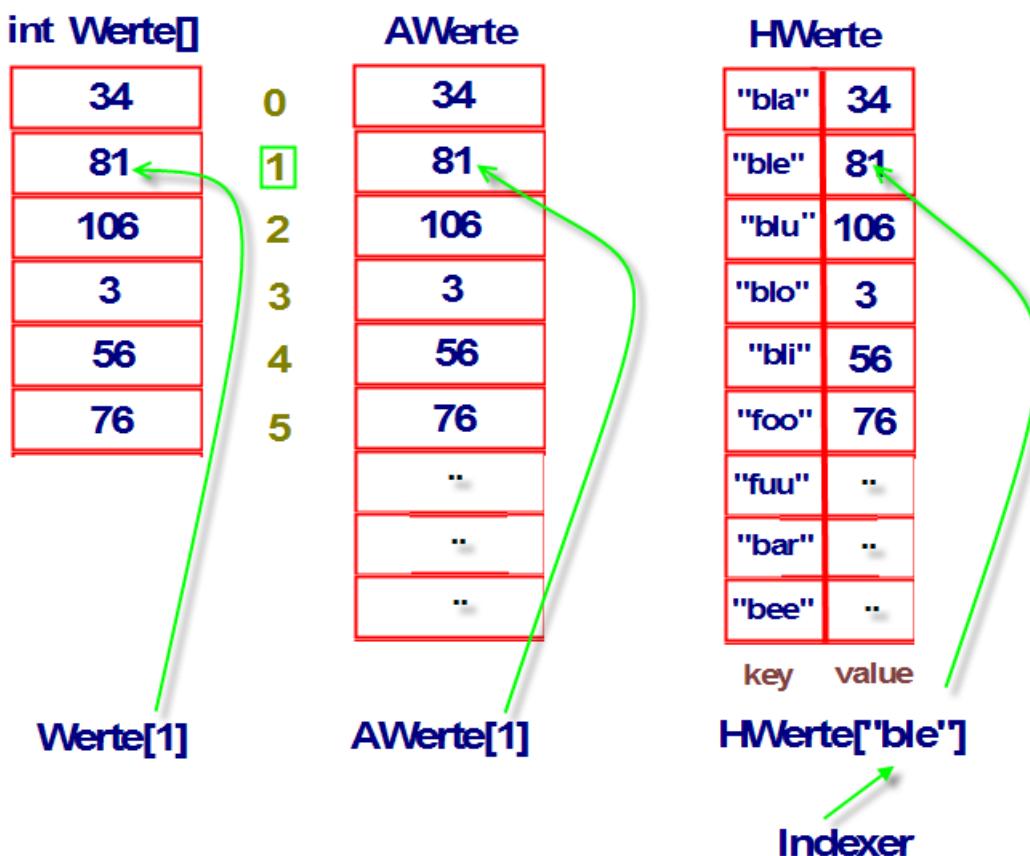


Count()	
Capacity()	
Add()	
Clear()	
Insert()	
Remove()	
RemoveAt()	

Hashtables

Hashtable sind eine besondere Art von **Collections**, nämlich in der Tatsache, wie auf die Elemente in der Liste zugegriffen werden kann.

Um ein Element einer Liste hinzufügen, muss nicht nur das Element, sondern auch ein sog. **Key (Schlüssel)** mit angegeben werden, unter dem das Element später gefunden werden kann. Dies erspart das u. U. aufwendige Durchlaufen der gesamten Liste beim Suchen, da über den Schlüssel direkt auf das gesuchte Element zugegriffen werden kann.



Das .NET-Framework implementiert mehrere Arten von Hashtabellen, nämlich die sog. Hashtable (.NET 1.0) bzw. das sog Dictionary (.Net 2.0). Die Unterschiede liegen in der Typsicherheit (Non-generic vs. Generic) und dem Weglassen von Cast-Vorgängen beim Lesen.

Beispiel:

<http://stackoverflow.com/questions/301371/why-is-dictionary-preferred-over-hashtable>

Coming to difference between `HashTable` & `Dictionary`, `Dictionary` is generic where as `Hashtable` is not Generic. We can add any type of object to `HashTable`, but while retrieving we need to Cast it to the required Type. So, it is not type safe. But to `Dictionary`, while declaring itself we can specify the type of Key & Value, so no need to cast while retrieving. Let's take an Example,

HashTable

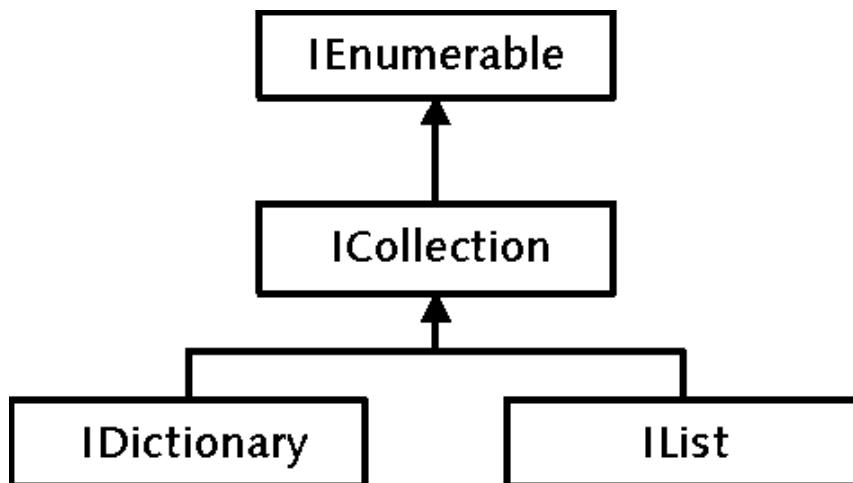
```
class HashTableProgram
{
    static void Main(string[] args)
    {
        Hashtable ht = new Hashtable();
        ht.Add(1, "One");
        ht.Add(2, "Two");
        ht.Add(3, "Three");
        foreach (DictionaryEntry de in ht)
        {
            int Key = (int)de.Key; //Casting
            string value = de.Value.ToString(); //Casting
            Console.WriteLine(Key + " " + value);
        }
    }
}
```

Dictionary

```
class DictionaryProgram
{
    static void Main(string[] args)
    {
        Dictionary<int, string> dt = new Dictionary<int, string>();
        dt.Add(1, "One");
        dt.Add(2, "Two");
        dt.Add(3, "Three");
        foreach (KeyValuePair<int, String> kv in dt)
        {
            Console.WriteLine(kv.Key + " " + kv.Value);
        }
    }
}
```

Arbeit mit Listen – (Durchlaufen_Sortieren_Suchen)

Die Grundfunktionalität aller Auflistungen lässt sich auf elementare Methoden zurückführen. Es ist deshalb nicht verwunderlich, dass die Gemeinsamkeiten durch Schnittstellen beschrieben werden, die von den Klassen implementiert werden. Dabei handelt es sich um die folgenden:



Diese Schnittstellen bilden eine Hierarchie, deren Wurzel **IEnumerable** ist, aus der **ICollection** abgeleitet wird. Beide Schnittstellen sind charakteristisch für Auflistungsklassen, denn sie stellen die wichtigsten Grundfunktionalitäten bereit. **IDictionary** und **IList** leiten sich zudem aus **ICollection** ab und spalten die Auflistungsklassen in zwei Gruppen:

1. Klassen, die das Interface **IList** implementieren, beschreiben Objektauflistungen, auf deren Einträge über einen Index zugegriffen wird.
2. Klassen, die das Interface **IDictionary** implementieren, verwalten ihre Einträge über eine Schlüssel-Wert-Kombination.

Die Schnittstelle »**IEnumerable**«

Diese Schnittstelle hat nur die Methode `GetEnumerator`, die ein Enumerator-Objekt bereitstellt. Ein Enumerator verfügt über die Fähigkeit, eine Auflistung elementweise zu durchlaufen. Damit gleicht dieses Objekt einem Positionszeiger, dem drei Methoden eigen sind: `Current`, `MoveNext` und `Reset`.

Der Enumerator positioniert sich standardmäßig vor dem ersten Eintrag einer Auflistung. Um ihn auf den ersten Eintrag und anschließend auf alle Folgeeinträge zeigen zu lassen, muss die Methode `MoveNext` ausgeführt werden. Mit `Current` wird auf den Eintrag zugegriffen, auf den der Enumerator aktuell zeigt. `Reset` setzt den Enumerator in seine Ausgangsposition zurück, also vor den ersten Eintrag.

Es gibt eine Situation, in der die Fähigkeit des Enumerators ausgesprochen wichtig ist. Es ist die foreach-Schleife, mit der eine Auflistung vom ersten bis zum letzten Element durchlaufen wird:

Auf die Inhalte von Listen kann mit Hilfe verschiedener Methoden zugegriffen werden:

Die Schnittstelle **IList**

Auflistungen, die **IList** implementieren, zeichnen sich dadurch aus, ihre Elemente über Indizes verwalten zu können. Das beste Beispiel hierfür dürfte die Klasse **ArrayList** sein, aber auch eine große Anzahl weiterer, meist steuerelementspezifischer Auflistungen gehört zu dieser Gruppe.

Methode/Eigenschaft	Beschreibung
Item	Stellt den Indexer für die IList -Klasse dar und dient dem Zugriff auf ein Listenelement.
Add	Hängt ein neues Element an das Ende der Auflistung an.
Clear	Löscht alle Elemente der Auflistung.
Contains	Gibt zurück, ob ein bestimmtes Objekt bereits zu der Auflistung gehört.
IndexOf	Liefert den Index eines bestimmten Objekts.
Insert	Fügt ein Objekt an einer bestimmten Position in die Auflistung ein.
IsFixedSize	Beschreibt mit einem booleschen Wert, ob die Kapazität der Auflistung dynamisch vergrößert werden kann oder nicht.
IsReadOnly	Beschreibt mit einem booleschen Wert, ob die Auflistung schreibgeschützt ist.
Remove	Löscht ein Objekt unter der Angabe der Referenz.
RemoveAt	Löscht ein Objekt unter der Angabe des Index.

Tabelle: Methoden und Eigenschaften von Arrays

Ihre Stärke spielen Schnittstellen aus, wenn sie von mehreren Klassen implementiert werden. Jede Klasse weist dann dieselben Merkmale und Verhaltensweisen auf. Wenn Sie mit einer Klasse gearbeitet haben, die eine oder mehrere gängige Schnittstellen unterstützt, sollten Sie auch mit allen anderen Klassen umgehen können, welche die gleichen Schnittstellen unterstützen. Das trifft insbesondere auf die Schnittstelle **IList** zu, weil sie von sehr vielen Klassen des .NET Frameworks implementiert wird. Es ist daher empfehlenswert, sich insbesondere mit den Eigenschaften und Methoden dieser Schnittstelle vertraut zu machen. Beispiele dazu werden Ihnen im weiteren Verlauf dieses Buchs noch ausgesprochen viele begegnen.

Indexer

Mit C# können Klassen und Strukturen so definiert werden, dass deren Objekte wie ein Array indiziert werden können. Indizierbare Objekte sind in der Regel Objekte, die als Container für andere Objekte dienen – vergleichbar einem Array.

Stellen Sie sich vor, Sie würden die Klasse Fußballmannschaft entwickeln. Eine Mannschaft setzt sich aus vielen Einzelspielern zusammen, die innerhalb der Klasse in einem Array vom Typ Spieler verwaltet werden. Instanziieren Sie die Klasse Fußballmannschaft mit

```
Fußballmannschaft Wacker = new Fußballmannschaft();
```

wäre es doch zweckdienlich, sich von einem bestimmten Spieler beispielsweise mit der Anweisung

```
string name = Wacker[2].Zuname;
```

den Zunamen zu besorgen.

Genau das leistet ein Indexer. Wir übergeben dem Objekt einen Index in eckigen Klammern, der ausgewertet wird und die Referenz auf ein Spieler-Objekt zurückliefert. Darauf können wir mit dem Punktoperator den Zunamen des gewünschten Spielers ermitteln – vorausgesetzt, diese Eigenschaft ist in der Klasse Spieler implementiert.

Und wie muss jetzt die Klasse Fussballmannschaft implementiert sein? Die Lösung sieht wie folgt aus:

```

class Program {
    static void Main(string[] args) {
        Fußballmannschaft Wacker = new Fußballmannschaft();
        // Spieler im Team aufnehmen
        Wacker[0] = new Spieler("Fischer", "Udo", 33, "Stürmer");
        Wacker[1] = new Spieler("Müller", "Peter", 25, "Verteidiger");
        Wacker[2] = new Spieler("Mamic", "Miroslav", 22, "Torhüter");
        Wacker[1] = new Spieler("Meier", "Gert", 25, "Stürmer");
        // einen Spieler ausgeben
        Console.WriteLine("Name: {0}, {1}", Wacker[1].Zuname, Wacker[1].Vorname);
        string name = Wacker[2].Zuname;
        Console.ReadLine();
    }
}

#####
class Fußballmannschaft {
    private Spieler[] team = new Spieler[25];
    // Indexer
    public Spieler this[int index] {
        get { return team[index]; }
        set {
            if (team[index] == null)
                team[index] = value;
            else
                Console.WriteLine("Der Index {0} ist bereits belegt", index);
        }
    }
}

class Spieler {
    public string Zuname, Vorname, Position;
    public int Alter;
    public Spieler(string zuname, string vorname, int alter, string position) {
        this.Zuname = zuname; this.Vorname = vorname;
        this.Alter = alter; this.Position = position;
    }
}

```

Sortieren und Suchen in Listen

Wer jetzt sagt, dass dies eigentlich von Datenbanken erledigt wird, sollte bedenken, dass Datenbankabfragen ebenfalls Zeit kosten. Es ist insbesondere dann ärgerlich, wenn die Daten ja eigentlich schon im Speicher des Rechners sind.

Sortieren und Suchen in Listen sind Standardaufgaben, die immer wieder anfallen. Sortieren und Suchen hängen eng miteinander zusammen. Listen werden häufig sortiert, um anschließend schneller Suchanfragen auf die sortierte Liste stellen zu können.

Collections bieten zum Sortieren bereits eine Sort-Methode an. Diese muss jedoch in der Lage sein, alle möglichen Objekttypen sortieren zu können.

Eine weitere Problematik besteht darin, dass es nur bei einfachen Datentypen relativ klar ist, nach welchem Kriterium sortiert werden soll.

Doch nach welchem Kriterium soll ein komplexer Datentyp sortiert werden?

```
public class Person
{
    private int Alter;
    private string Vorname;
    private string Nachname;

    public Person(string _vorname, string _nachname, int _alter)
    {
        this.Alter = _alter;
        this.Nachname = _nachname;
        this.Vorname = _vorname;
    }

    //Methode zum ausgeben der Felder
    public override string ToString()
    {
        return "" + Vorname + ", " + Nachname + ", " + Alter;
    }
}
```

```
void CboSortierungSelectedIndexChanged(object sender, EventArgs e)
{
    private ArrayList personen = new ArrayList();
    Person steinam = new Person("Karl", "Steinam", 43);
    Person oppenheimer = new Person("Klara", "Oppenheimer", 36);

    personen.Add(steinam);
    personen.Add(oppenheimer);

    personen.Sort(); Nach welchem Kriterium soll  
sortiert werden ?
```



Queues

TODO

Stacks

TODO



Aufgaben

Arrays

1. Erstellen Sie jeweils ein Array, eine ArrayList und eine Hashtable/Dictionary mit 1 mio. unsortierten Einträgen.

Messen Sie die Zeit

- Zum Erstellen der jeweiligen Ausgangsstruktur
- Zum Finden eines beliebigen Wertes. Um Zufallsergebnisse zu vermeiden, führen Sie diesen Vorgang 10-mal hintereinander mit anderen Werten durch.

2. Berechnen Sie die Summe der Zahlen im Array.

```
class Uebung1
{
    public static void Main ( String[] args )
    {
        int[] arr = {0, 1, 2, 3};

        int summe = #Has to be done

        Console.WriteLine( "Summe aller Zahlen = " + summe );
    }
}
```

3. Begutachten Sie das folgende Programm:

```
class Uebung2
{
    public static void Main ( String[] args )
    {
        int[] arr = {13, -4, 82, 17};
        int[] doppelt;

        Console.WriteLine( "Ursprüngliches Array: "
            + arr[0] + " " + arr[1] + " " + arr[2] + " " + arr[3] );

        // Konstruieren Sie ein Arrayobjekt für doppelt.
```

Vervollständigen Sie das Programm, so dass eine neuen Array doppelt konstruiert wird. Kopieren Sie jetzt die Werte von arr nach doppelt, aber machen Sie die Werte in doppelt zweimal so groß als wie sie in arr sind.

4. Begutachten Sie das folgende Programm:

```
class Uebung3
{
    public static void main ( String[] args )
    {
        int[] arrA    = { 13, -22,  82,  17};
        int[] arrB    = {-12,   24, -79, -13};
        int[] summe   = {  0,     0,     0,     0};

        // Addieren Sie die Werte der entsprechenden Slots von arrA und arrB,
        // und stellen Sie das Ergebnis in den entsprechenden Slot von summe.

        Console.WriteLine( "Summe: "
            + summe[0] + " " + summe[1] + " " + summe[2] + " " + summe[3]
        );
    }
}
```

Vervollständigen Sie das Programm mit vier Zuweisungsanweisungen, so dass jeder Slot von summe die Summe der entsprechenden Slots von arrA und arrB enthält. D.h., addieren Sie Slot 0 von arrA mit Slot 0 von arrB und stellen Sie das Ergebnis in den Slot 0 von summe und so weiter.

5. Vervollständigen Sie das folgende Programm, so dass es die Summe aller Elemente des Arrays berechnet. Schreiben Sie das Programm, so dass es sogar dann funktioniert, wenn die Dimensionen der Zeilen und Spalten geändert werden.

```

class ArraySumme
{
    public static void main ( String[] args ) throws IOException
    {
        int[][] data = {
            new int[]{ 3, 2, 5},                  //+10
            new int[]{ 1, 4, 4, 8, 13},           //+30
            new int[]{ 9, 1, 0, 2},               //+12
            new int[]{ 0, 2, 6, 3, -1, -8 } //+ 2
            } ;                                //summe:      54

        // Summe deklarieren

        // Summe berechnen
        for ( int zeile=0; zeile < data.length; zeile++)
        {
            for ( int spalte = 0; spalte < ???; spalte++)
            {

            }
        }

        // Summe ausgeben
        Console.WriteLine( );
    }
}

```

6. Grundsätzlich können Arrays **ohne oder mit new**-Operator deklariert werden. Worin liegt der genaue Unterschied ?
7. Lesen Sie in der MSDN nach, wie die Sort-Eigenschaft von System.Array implementiert wird.

Ist diese Implementierung schnell

Wie kann diese Implementierung auf eigene Datentypen (z.B. eine fiktive Klasse Person angewendet werden)

8. ArrayLists sind sog. doppelt verkettete Listen, bei denen ein Element sowohl den Vorgänger als auch seinen Nachfolger kennt.

Wie würden Sie diese Funktionalität implementieren, wenn Sie für eine Liste von Personen eine eigene verkettete Liste implementieren müssten, ohne eine ArrayList oder sonstige Klasse aus dem Collections-Namespace verwenden dürften.

9. Welche Ausgabe erzeugt folgendes Codeschnipsel?

```
int [] alpha = new int [2];
int [] beta;
beta = alpha;
Console.WriteLine("alpha equals beta ist " + alpha.Equals(beta));
Console.WriteLine("alpha hat " + alpha.Length +" Komponenten");
```

10. Was ist der Unterschied zwischen folgenden Schreibweisen

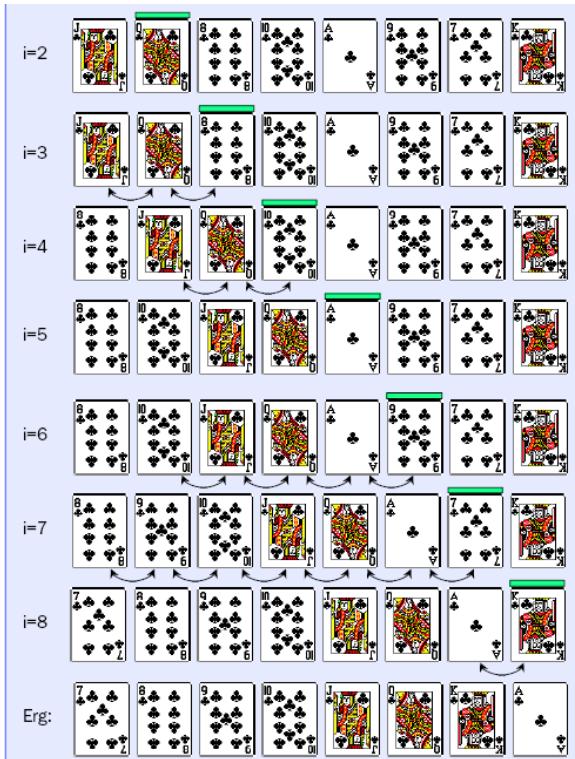
```
int [][] vArray = new int[2][];
int[,] Arrayname8 = new int[,]{{1,2},{3,4},{5,6},{7,8}};
```

11. Der wohl einfachste Sortieralgorithmus trägt den Namen **Selection Sort** und basiert auf zwei ineinander geschachtelten Schleifen, von denen die äußere alle Elemente bis auf das letzte durchläuft. Ausgehend vom aktuellen Element der äußeren Schleife werden in der inneren Schleife alle Elemente bis zum Ende des Arrays durchlaufen, um dort das jeweils kleinste Element zu suchen. Dieses kleinste Element wird dann mit dem aktuellen Element der äußeren Schleife ausgetauscht

Implementieren Sie dieses Sortierverfahren mit Hilfe von C#.

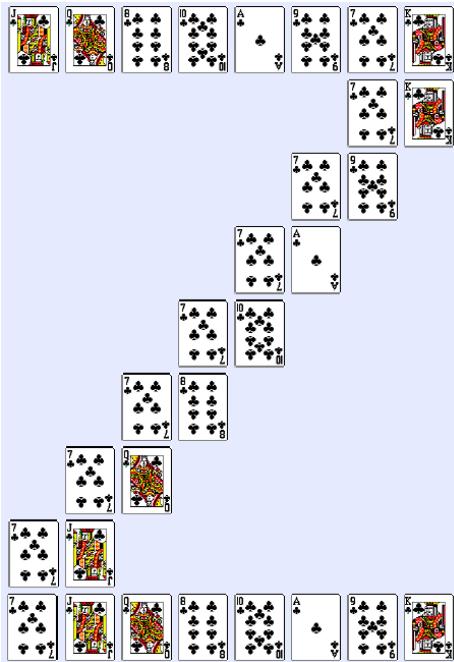
Testen Sie das Verfahren an einem int-Array mit den Werten (123,43,567,54, 89,1,43,7)

12. Beim so genannten „Sortieren durch Einfügen“ (Insertion Sort) werden die Elemente beginnend mit dem zweiten Element durchlaufen.



Bei jedem Element wird überprüft, ob es nicht besser vor eines der Vorgänger-Elemente passt. Ist das aktuelle Element kleiner als eines seiner Vorgänger-Elemente, wird es einfach vor dieses Vorgänger-Element platziert. Dazu muss allerdings Platz im Array geschaffen werden. Jedes Vorgänger-Element, das kleiner als das aktuelle Element ist, wird dazu einfach bei jedem Schritt eine Position nach rechts gerückt.(siehe Abbildung). Die Implementierung dieses Algorithmus sieht dann so aus:

13. Bubble Sort



Dem Insertion Sort sehr ähnlich ist der Bubble Sort-Algorithmus. Während beim Insertion-Sort die Originalposition des Elements und seine spätere Einfügeposition sehr weit auseinander liegen können, werden beim Bubble Sort immer nur zwei direkte Nachbarn ausgetauscht. Seinen Namen verdankt dieser Algorithmus der Tatsache, dass ein Element wie eine Blase im Sprudelwasser an den Beginn des Arrays aufsteigt. Die Abbildung zeigt, wie die Kreuz Sieben an den Anfang des Arrays „blubbert“. Auch beim Bubble Sort-Algorithmus werden zwei ineinander verschachtelte Schleifen eingesetzt: Die äußere bewegt sich vom Anfang des Arrays bis zu dessen Ende, während die innere alle Elemente vom Ende

des Arrays bis zum aktuellen Element der äußeren Schleife durchläuft. In der inneren Schleife werden zwei unmittelbare Nachbarn nur dann vertauscht, wenn ihre Reihenfolge nicht der gewünschten entspricht.

Aufgaben zu ArrayLists

1. Gegeben ist folgender Quellcode:

```

public class ClassA
{
    public int Prop;

    public ClassA(int x) {
        Prop = x;
    }
}

#####
ClassA obj1 = new ClassA(1);
ClassA obj2 = new ClassA(2);
ArrayList col = new ArrayList();
col.Add(obj1);
col.Add(obj2);

#####
ClassA obj3 = new ClassA(3);
col.Insert(1, obj3);

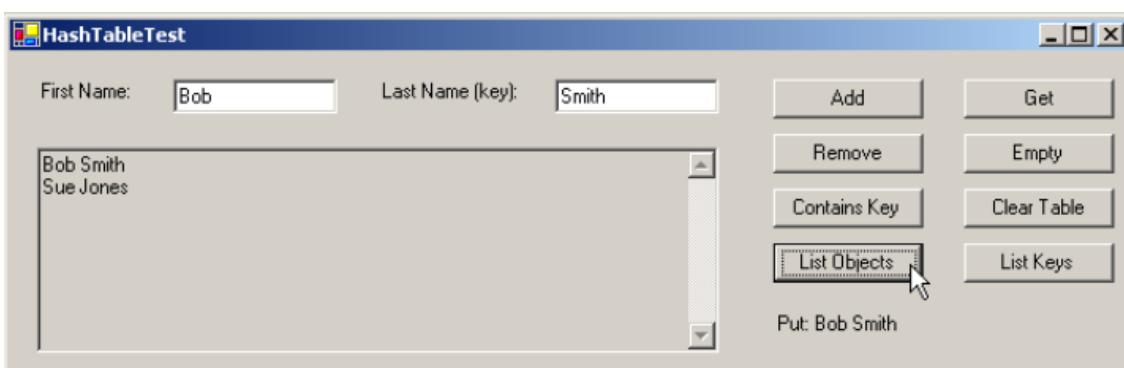
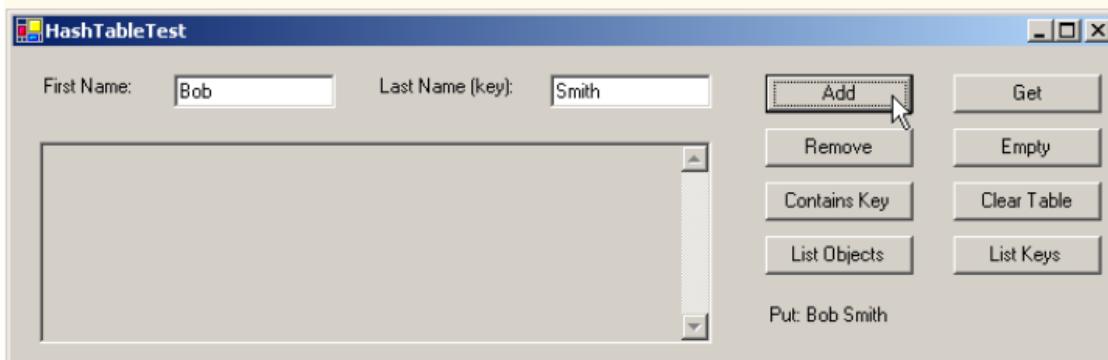
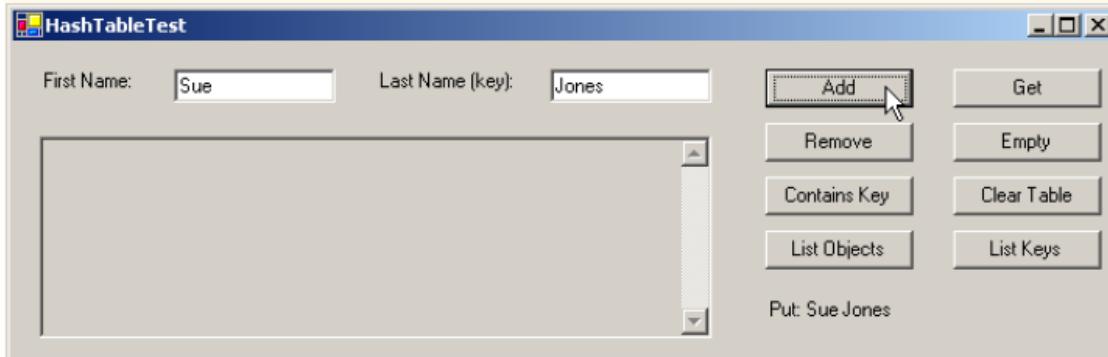
```

Was passiert mit dem Element obj2, das vorher die Position des Index 1 einnahm? Iterieren Sie durch die Liste mit Hilfe einer foreach-Schleife und bewerten Sie das Ergebnis.

2. Schreiben Sie ein Programm, das zwei sortierte ArrayListen aus Integer-Daten in eine einzige sortierte Liste überführt. Die merge()-Methode des Programms sollte eine Referenz auf beide ArrayLists erhalten und die sortierte neue ArrayList als Referenz zurückgeben.

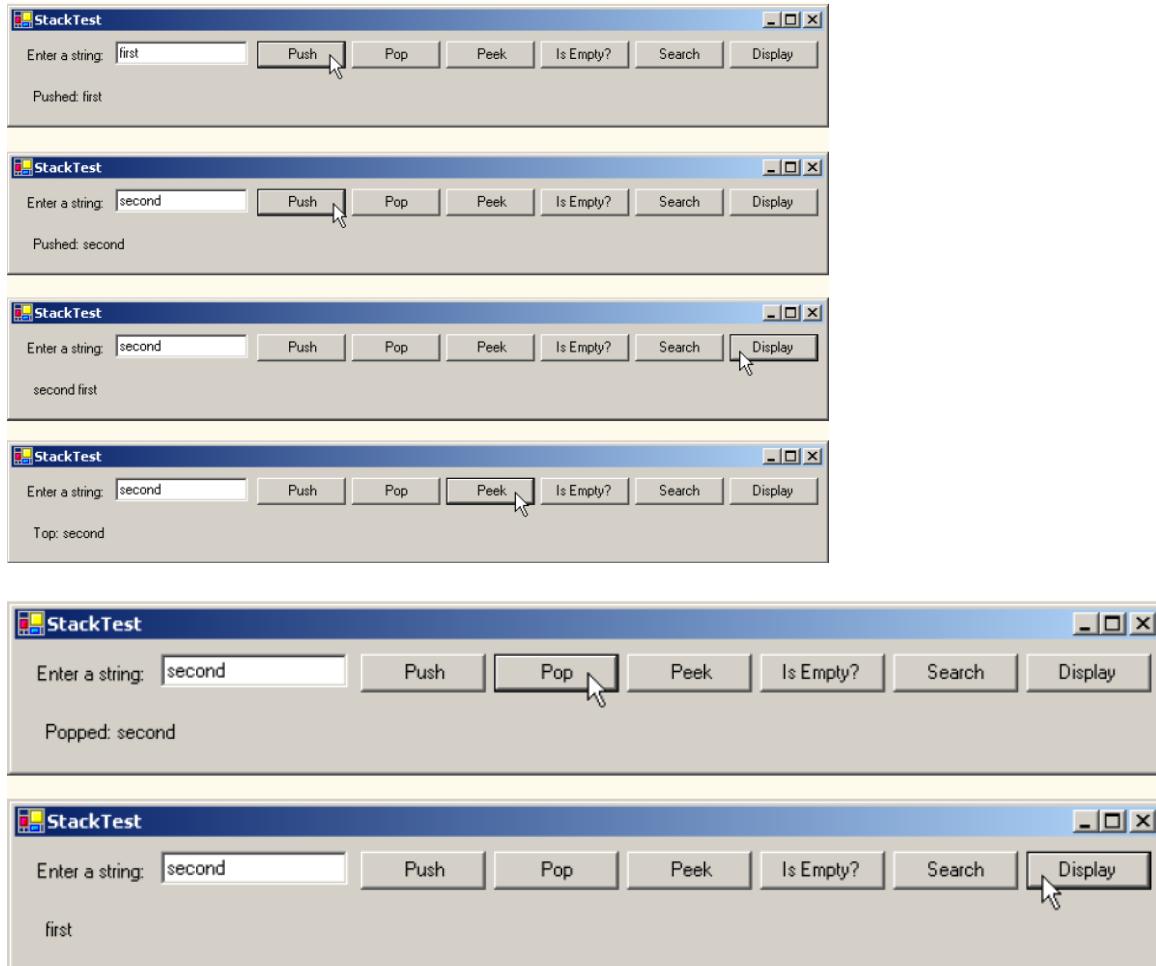
Aufgaben zu Hashtable

1. Erstellen Sie eine WinForm-Anwendung und implementieren Sie die Funktionalitäten gemäß folgender Abbildung:



Aufgaben zu Stacks

1. Erstellen Sie eine WinForm-Anwendung und implementieren Sie die Funktionalitäten gemäß folgender Abbildung:



2. Schreiben Sie ein Programm, dass eine Textzeile entgegennimmt und ein Stack-Objekt verwendet, um die Zeile in umgedrehter Reihenfolge wieder auszugeben.

Weiterführende Aufgaben

- In einer Firma werden telefonisch Aufträge entgegengenommen. Jeder Auftrag besteht aus den Daten
 - Bezeichnung des bestellten Produkts
 - Bestellte Menge
 - Kundennummer des Bestellers

Schreiben Sie ein Programm, das der Telefonist der Firma während der Auftragserteilung benutzt, mit welchem er folgenden Dialog führt:

```
Neuen Auftrag eingeben (j/n) ? j
Bestelltes Produkt: Himbeermarmelade
Bestellte Menge: 500
Kundennummer: 3678
Neuen Auftrag eingeben (j/n) ? j
... weitere Aufträge, solange bis ...
Neuen Auftrag eingeben (j/n) ? n
```

Das Programm speichert alle Auftragsdaten in einer Hashtable und schreibt diese Liste am Ende komplett und in der Eingabe-Reihenfolge auf der Console aus - eine Zeile pro Auftrag.

Entwickle eine Klasse EMailSpeicher, mit der man E-Mails von Benutzern verwalten kann.

Fülle die folgende Klasse sinnvoll aus:

```
class E-MailSpeicher
{
    Hashtable emails ...

    /**
     * Speichert Name und E-Mail-Adresse wie
     [K. Steinam, steinam@klara-oppenheimer-schule.de].
     */
    void hinzufügen( String name, String email ) {
        //your code goes here
    }

    /**
     * Sucht nach der E-Mail-Adresse mit dem exakten Namen.
     */
    String sucheExakt( String name ) {
        // your code goes here
    }

    /**
     * Liefert die erste E-Mail-Adresse mit dem Teil des Na-
     mens.
     *
     */
    String sucheUnscharf( String name ) {
        //your code goes here
    }
}
```

Wenn die E-Mail-Adresse mit dem Präfix "mailto:" beginnt, dann wollen wir dies abschneiden.

Erweitere das Programm so, dass alle Namen, die ein Muster ergeben, in einem Feld zurückgegeben werden.

Lösung siehe [hier](#)

- Man möchte gerne in einem Satz einige Wörter ersetzen, zum Beispiel alle Wörter "doof" durch "unschlau" und "pfusch" durch "ungünstig". Überlege, wie man eine Klasse WordFilter mit einer Methode addWordPair(String find, String replace) ausstatten kann, sodass die Methode String replace(String s) alle Wörter im Parameter untersucht und gegebenenfalls die Ersetzungen vornimmt.

Jede natürliche Zahl lässt sich als Summe schreiben. Für alle Zahlen größer 2 gibt es dabei mehr als eine Möglichkeit. Nehmen wir als Beispiel die Zahl 4, dann haben wir folgende so genannte Partitionen:

```
4
3+1
2+2
2+1+1
1+1+1+1
```

Die berechneten Summen kommen nur einmal vor. (Also nicht 1+3 und 3+1. Man erreicht dies durch eine Sortierung der Größe nach.) Ein weiteres Beispiel für 5:

```
5
4+1
3+2
3+1+1
2+2+1
2+1+1+1
1+1+1+1+1
```

- Schreibe ein Programm, welches für eine natürliche Zahl alle Partitionen ausgibt.
- Merke die Einsen in einer Variablen einsen. Merke zusätzlich, aus wie vielen unterschiedlichen Summanden eine Zahl aufgebaut ist.
- Summiere die Gesamtzahl der Summanden ebenfalls und vergleiche die Summe mit der Variablen einsen.

Ein Beispiel für die Zahl 6:

Partitionen	Anzahl unterschiedlicher Zahlen
6	1
5+1	2
4+2	2
4+1+1	2
3+3	1
3+2+1	3
3+1+1+1	2
2+2+2	1
2+2+1+1	2
2+1+1+1+1	2
1+1+1+1+1+1	1
	--
Summe :	19



Summiert man die Einsen der Partitionen auf, so kommt man ebenfalls auf 19!

Zeige das Phänomen noch an anderen Zahlen. (Anmerkung: Der Beweis dafür findet sich unter anderem in "Mathematical Gems III" von Ross Honsberger.)

Lösung siehe [noch keine](#)

- Listen sind eine Möglichkeit beliebig viele Daten sehr einfach abzuspeichern. Dabei werden die Daten miteinander verkettet. Bei einer einfach verketteten Liste enthält jedes Listenelement einen Zeiger auf das nächste. Außerdem gibt es noch einen Zeiger auf das erste Element, damit man die Liste wieder findet. Bei einer doppelt verketteten Liste gibt es zusätzlich noch einen Zeiger auf das vorige Element. Dadurch kann man leichter durch die Liste navigieren.

Als Beispiel erzeugen wir eine einfach verkettete Liste. Jedes Listenelement ist ein Objekt auf dem Heap, das ein Zeiger auf das nächste Listenelement enthält. Das letzte Listenelement zeigt auf null. Dies ist sehr einfach mit folgender Klasse zu realisieren.

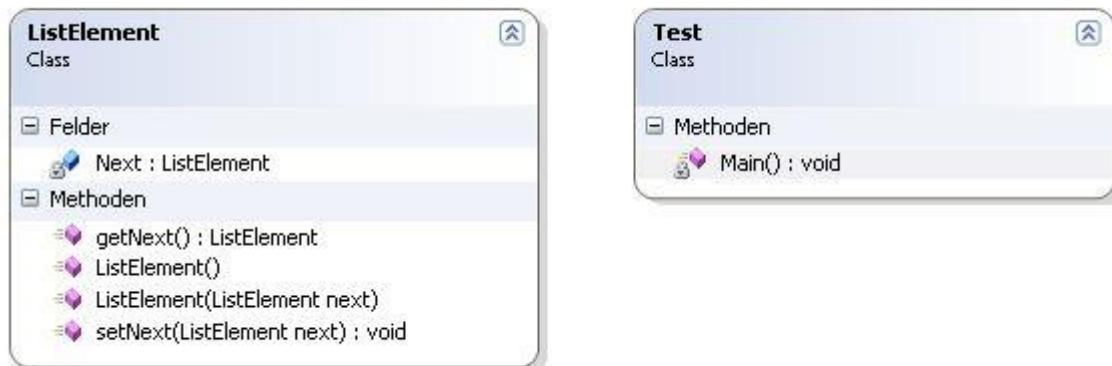
```
class ListElement
{   private ListElement Next;

    public ListElement () // Listelement ohne Nachfolger
    {   Next=null;
    }
    public ListElement (ListElement next) // mit Nachfolger
next
    {   Next=next;
    }
    public void setNext (ListElement next) // setze Nachfolger
    {   Next=next;
    }
    public ListElement getNext () // lies Nachfoger
    {   return Next;
    }
}
```

Mit Hilfe dieser Klasse erzeugen wir 10 Elemente. Die Listenelemente werden jeweils vorne an die Liste gehängt wird.

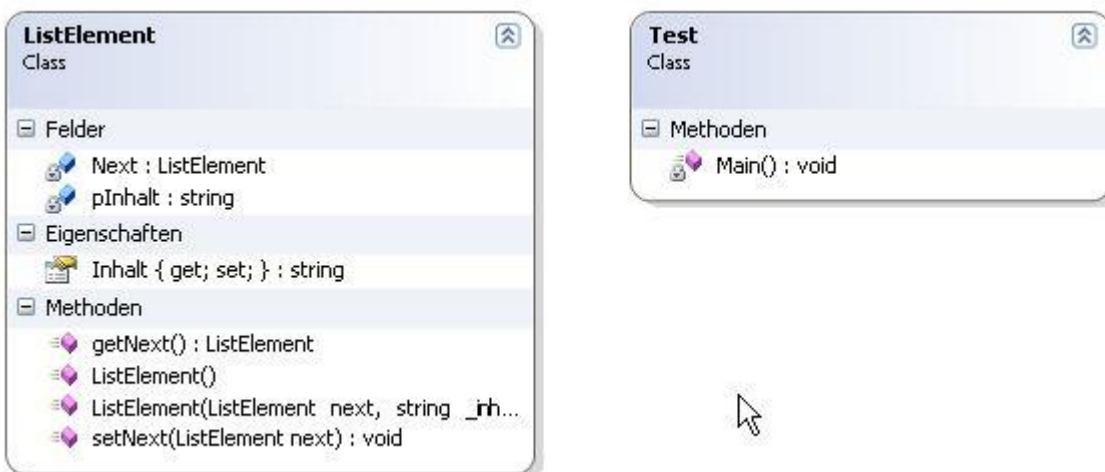
```
public class Test
{   public static void main (String args[])
    {   ListElement e=null;
        for (int i=0; i<10; i++)
            {   e=new ListElement (e);
```

}



- Wandeln Sie den Quellcode so um, dass Sie ein in c# lauffähiges Programm erhalten.

- Bisher speichert ListElement lediglich einen Verweis auf seine Nachfolger. Erweitern Sie die Klasse um ein Attribut **Inhalt** vom Typ string. Übergeben Sie den Wert für die Variable im Konstruktor und fügen Sie die Klasse eine öffentliche Property zum Lesen und Schreiben hinzu. Passen Sie ebenfalls die Main-Methode der Starterklasse an.



- Wandern Sie durch die Liste und geben Sie jeweils den Wert der Variable **Inhalt** jedes Listen-elementes aus.
 - Dieses Programm gibt die Zahlen in umgekehrter Reihenfolge aus, da das neueste Listen-Ele-
ment jeweils vorne angehngt wird, aber die Liste von vorne nach hinten gedruckt wird.

Lösung siehe [hier](#)

Ändern Sie das Hauptprogramm für die verkettete Liste (mit ListElement) so um, dass die neuen Listenelemente jeweils an das Ende der Liste angehängt werden.

Lösung siehe [hier](#)

Weitergehende Fragen



Objektorientierte Programmierung

Download:

- **Video.**
- **WebCast.**
- **WebCast.**
- **Arbeitsblatt.**
- **OOP_Teil2.**
- **Video.**
- http://en.csharp-online.net/Nested_Classes
- <http://msdn.microsoft.com/en-us/magazine/cc534993.aspx>
-

<http://www.microsoft.com/germany/msdn/library/net/ILDASMIstIhrBester-Freund.mspx> <http://stackoverflow.com/questions/48872/why-when-should-you-use-nested-classes-in-net-or-shouldnt-you>

- <http://verify.rwth-aachen.de/programmierungWS12/>

<http://de.wikipedia.org/wiki/Programmierparadigma> <http://stackoverflow.com/questions/538060/proper-use-of-the-idisposable-interface>

Grundsätze der Programmierung unterliegen einer ständigen Weiterentwicklung. Die bisher besprochenen Konzepte stammen aus dem Bereich der sog. **Strukturierten Programmierung**, einer Methode, die bis Mitte der 80er Jahre weitverbreitet war. Sie stieß jedoch mit immer komplexer werdenden Programmen und Abläufen an ihre Grenzen. Eine neue Sichtweise entwickelte sich, die Probleme mit einer neuen Art und Weise lösen wollte, der sog. **objektorientierten Programmierung**.

Grundlagen

Objektorientierte Programmierung entstand aus mehreren Schritten heraus.

- Notwendigkeit komplexerer Datentypen

Die einfachen Datentypen (string, int, float, etc.) reichten häufig nicht mehr aus, um sinnvoll Zustände in Variablen abspeichern zu können. Aus diesen Problemen entwickelten sich zunächst die komplexen Datentypen **enum**, **type** und **struct**

- Die Trennung von Methoden, die auf Daten operieren, und den zu verwaltenden Daten, war umständlich.

All diese Gründe führten zur Entwicklung neuer Konzepte und mündeten in die Entwicklung objektorientierter Programmiersprachen

- **Arbeitsblatt.**

Enumerationen und Structures bilden aus einfachen Datentypen komplexere Gebilde.

Sie sind jedoch weiterhin Value Types, d.h. sie werden **by val** übergeben.

Enumerations

Eine Enumeration ist ein eigener **Werttyp**, der aus einer Gruppe **benannter Konstanten**, der so genannten Enumeratorliste, besteht. Jeder Enumerationstyp verfügt über einen zugrunde liegenden Typ, der einem beliebigen ganzzahligen Typ außer char entsprechen kann. Per default ist der Rückgabetyp Int32.



```

using System;

public class EnumTest
{
    enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};

    public static void Main()
    {
        int x = (int) Days.Sun;
        int y = (int) Days.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}

=====
using System;

public class EnumTest
{
    enum Range :long {Max = 2147483648L, Min = 255L};
    public static void Main()
    {
        long x = (long) Range.Max;
        long y = (long) Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}

```

Finden Sie heraus, wie folgender Quellcode im .net-Framework implementiert wurde.

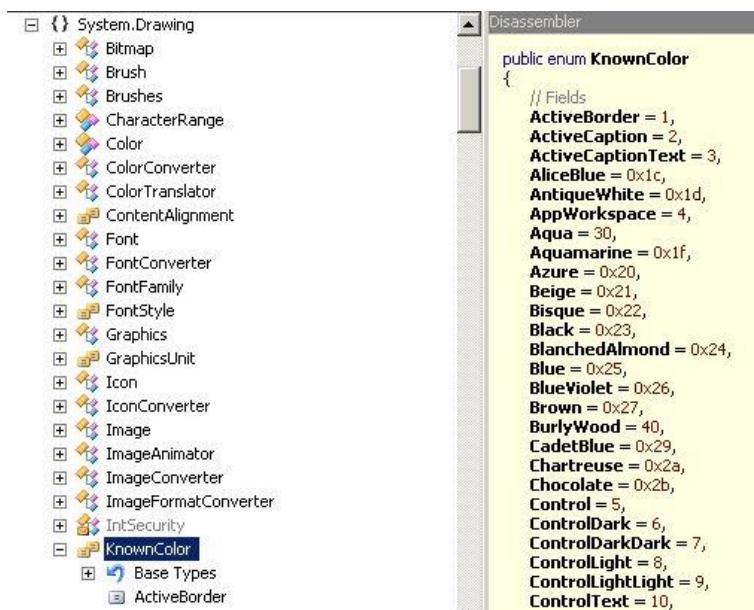
```

int farbe = (int) System.Drawing.KnownColor.AliceBlue;

Console.WriteLine(farbe);

```

Lösung mit Hilfe des Reflectors/ILSpy



The screenshot shows the ILSpy interface with the 'Disassembler' tab selected. On the left, a tree view displays the class hierarchy under the System.Drawing namespace. The 'KnownColor' class is expanded, and its definition is shown in the main pane. The code is as follows:

```
public enum KnownColor
{
    // Fields
    ActiveBorder = 1,
    ActiveCaption = 2,
    ActiveCaptionText = 3,
    AliceBlue = 0x1c,
    AntiqueWhite = 0x1d,
    AppWorkspace = 4,
    Aqua = 30,
    Aquamarine = 0x1f,
    Azure = 0x20,
    Beige = 0x21,
    Bisque = 0x22,
    Black = 0x23,
    BlanchedAlmond = 0x24,
    Blue = 0x25,
    BlueViolet = 0x26,
    Brown = 0x27,
    BurlyWood = 40,
    CadetBlue = 0x29,
    Chartreuse = 0x2a,
    Chocolate = 0x2b,
    Control = 5,
    ControlDark = 6,
    ControlDarkDark = 7,
    ControlLight = 8,
    ControlLightLight = 9,
    ControlText = 10,
```

Structures

Ein struct-Typ ist ein **Werttyp**, der Konstruktoren, Konstanten, Felder, Methoden, Eigenschaften, Indexer, Operatoren, Ereignisse und geschachtelte Typen enthalten kann.

Eine selbst geschriebene Structure **Adresse** hat z.B. folgenden Aufbau:

```
public struct Adresse
{
    private string pName;
    private string pVorname;
    private string pTelefon;

    public string Name
    {
        get{return this.pName; }
        set{this.pName = value; }
    }
    public string Vorname
    {
        get{return this.pVorname; }
        set{this.pVorname = value; }
    }
    public string Telefon
    {
        get{return this.pTelefon; }
        set{this.pTelefon = value; }
    }
}
```



Klassen und Instanzen

Zwischen Klassen und Strukturen gibt es viele Gemeinsamkeiten. Klassen haben jedoch einige zusätzliche Konzepte, z.B. Vererbung und sind sozusagen die Standardstruktur der objektorientierten Programmierung.

Bei der Arbeit mit dieser Struktur muss man zwischen 2 Elementen unterscheiden.

- Der Klasse

Sie definiert den Bauplan und enthält die Datendefinition sowie die Methoden, die mit diesen Daten arbeiten

Daten werden als **Attribute** bzw. **Felder** genannt und bilden strukturiert gesprochen, die globalen Variablen ab.

Methoden implementieren die Fähigkeiten einer Klasse, d.h. das, was Objekte dieser Klasse machen können.

- Objekte / Instanzen

Um mit dem Bauplan einer Klasse arbeiten zu können, muss man sich Objektvariablen schaffen, die ihre jeweiligen Daten verwalten und die Methoden auf sie anwenden.

Der Prozess des Schaffens von Objektvariablen heißt Instanziierung und ist grundsätzlich in 2 Schritte unterteilt.

- Deklaration einer Variablen vom Typ der Klasse
 - Erzeugen einer Objektvariablen mit Hilfe des **new**-Operators

- Nach dem Erzeugen spricht man auch von einer **Instanz** einer Klasse.

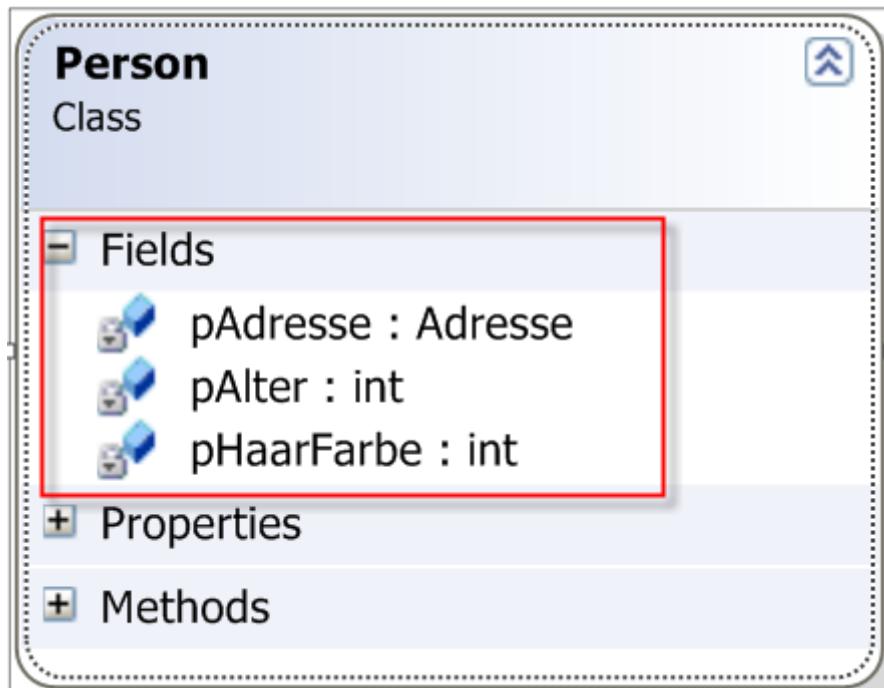
```
//Deklaration eines Objektes  
//Klasse    Objektvariable;  
  
Person steinam;
```

```
//Objekt wird instanziiert  
steinam = new Person();
```

```
//Alles auf einmal  
Person sierl = new Person();
```

Felder/Attribute

Über die dort definierten Variablen werden die Daten eines Objektes verwaltet/gehalten. Die Variablen können selbst wiederum komplexer Natur (sprich Referenzen auf Objekte) sein.

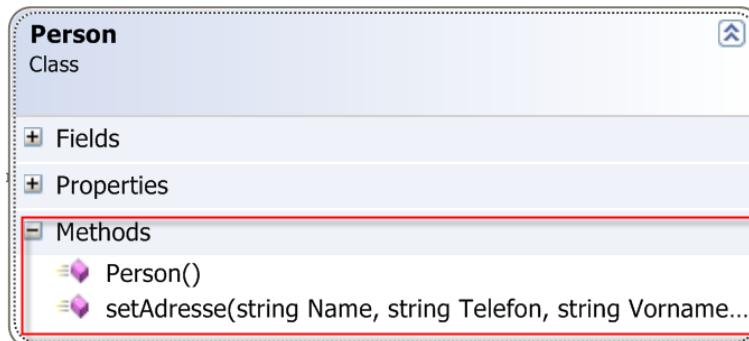


```
public class Person
{
    //private fields
    private Adresse pAdresse;
    private int pHaarFarbe;
    private int pAlter;
    ....
}
```

Methoden

Sie definieren das Verhalten eines Objektes, d.h. das, was ein Objekt kann. Innerhalb dieser Methoden wurde bisher von uns programmiert.

Eine sehr spezielle Methode ist der sog. **Konstruktor**. Dies ist die Methode, die beim Erzeugen eines Objektes quasi von selbst aufgerufen wird. Sie hat immer den Namen der jeweiligen Klasse, kann sich aber in den Parametern unterscheiden.



```

public Person()
{
    pAdresse = new Adresse();
}

public void setAdresse(string Name, string Telefon,
                      string Vorname, HaarFarbe myHaarfarbe)
{
    pAdresse.Name = Name;
    pAdresse.Telefon = Telefon;
    pAdresse.Vorname = Vorname;
    this.pHaarFarbe = (int)myHaarfarbe;
}

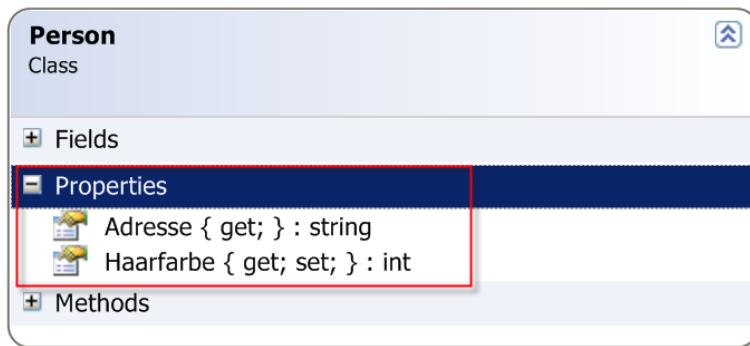
```

Properties

Die Variablen/Daten eines Objektes müssen mit Werten gefüllt werden. Hierfür schreibt man sich in vielen Programmiersprachen sog. **getter/setter** - Methoden, die zum Befüllen und Auslesen der Daten dienen.

Diese Vorgehensweise ist zwar auch unter .NET möglich, man hat aber darüberhinaus eine interessante Variante definiert, die sog. Properties. Sie sind zwar auch nichts anderes als Methoden, stellen sich für den Nutzer der Objekte aber als öffentliche Felder dar.

In der Weiterentwicklung des .net-Frameworks ist es mittlerweile dazu gekommen, dass es ausreicht, Properties zu definieren. Der Compiler definiert automatisch entsprechende Instanzattribute.



```
public class Person
{
    //private fields
    private Adresse pAdresse;
    private int pHaarFarbe;
    private int pAlter;

    //property
    public int Haarfarbe
    {
        get
        {
            return pHaarFarbe;
        }
        set
        {
            this.pHaarFarbe = value;
        }
    }
    .....
}
```

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren sind spezielle Methoden, die von der Laufzeitumgebung automatisch beim Erzeugen oder Zerstören einer Instanz aufgerufen werden.

- Konstruktor

Beim Erzeugen eines Objektes reserviert die Laufzeitumgebung im Hintergrund Speicher auf dem Heap, erzeugt das Objekt und ruft automatisch den parameterlosen Standardkonstruktor auf

Der Standardkonstruktor kann durch eigene Methoden überladen werden

```

CreateDestroy.cs (E:\unterricht\docbook_src\U1_U1_csharp\material)
1 public void CreateAndDestroy()
2 {
3 .
4 .    //Objekt Schueler erzeugen.
5     Person Schueler = new Person();
6 .
7 .
8 .    //Das Objekt lebt und arbeitet.
9     Schueler.setAdresse("NoName", "NoPhone",
10        "NoFirstName", HaarFarbe.White);
11    Console.WriteLine(Schueler.Adresse);
12 .
13 .
14 .    //Objekt Schueler zerstören.
15     schueler = null;
16 }

```

```

KonstruktorDestruktor.cs (E:\unterricht\docbook_src\U1_U1_csharp\material)
1 class Person
2 {
3 .
4 .    //Konstruktor wird beim Erzeugen des Objekts aufgerufen.
5     public Person()
6     {
7         pAdresse = new Adresse();
8         Console.WriteLine("Objekt ist erzeugt");
9     }
10 .
11 .
12 .    //Destruktor.
13     ~Person()
14     {
15         Console.WriteLine("Objekt zerstrt sich durch die GC");
16     }
17 }

```

- Destruktor

Der Destruktor ist das Gegenstck zum Konstruktor. In ihm werden Aufrumarbeiten durchgefrt, wenn die letzte Referenz auf eine Instanz freigegeben wird. Er wird normalerweise automatisch aufgerufen, kann aber wie in der Abbildung auch selbst definiert und mit eigenen Inhalten gefllt werden.

Statische Member

Man versteht darunter Felder, Methoden, Eigenschaften, etc., die benutzt werden knnen, ohne eine Instanz der entsprechenden Klasse bilden zu mssen.

Ein klassisches Beispiel fr eine statische Methode ist die **Main**- - Methode, die vor dem Instanziieren eines Objektes aufgerufen werden kann.

- Die statischen Eigenschaften werden durch das Schlsselwort **static** gekennzeichnet.
- Auf statische Member kann uber den Klassennamen zugegriffen werden.
- Innerhalb statischer Methoden oder Eigenschaften ist nur der Zugriff auf statische Member mglich

```

static.cs (E:\unterricht\docbook_src\U1_U1_csharp\material)
1 public void useStaticMember()
2 {
3     Person Wallner = new Person();
4     Person Steinam = new Person();
5     Console.WriteLine(Person.pAnzahl.ToString());
6 }

```

```

Personstatic.cs (E:\unterricht\)
public class Person
{
    //private fields.
    private Adresse pAdresse;
    private int pHaarFarbe;
    private int pAlter;

    public static int pAnzahl=0;

    //Konstruktor wird beim Erzeugen des Objekts aufgerufen.
    public Person()
    {
        pAdresse = new Adresse();
        System.Console.WriteLine("Objekt ist erzeugt");
        pAnzahl++;
    }
}

```

Ergebnis der Ausgabe: ???

Sichtbarkeit von Klassen und deren Membern

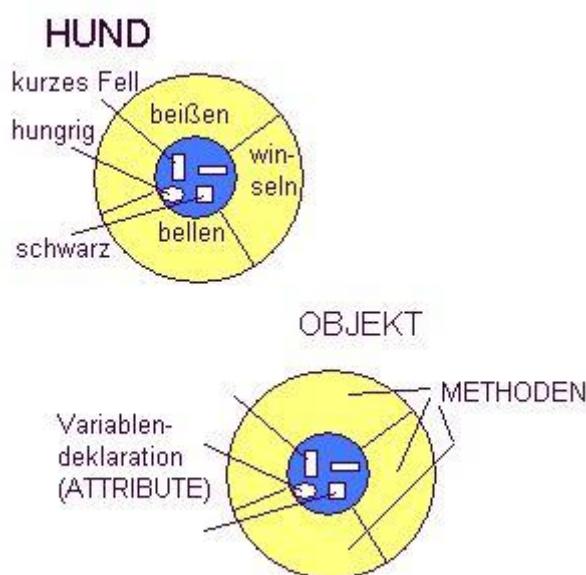
Klassen sowie darin enthaltene Methoden, Properties und Felder können durch sogenannte **Modifier** in ihrer Sichtbarkeit/ihrem Zugriff eingeschränkt werden. Es gelten folgende Schlüsselworte mit folgender Bedeutung.

Modifier	Bedeutung
private	- Der Zugriff ist nur innerhalb der Klasse möglich, in der die Deklaration erfolgt.
protected	Der Zugriff ist nur innerhalb der Klasse und aller davon abgeleiteten Klassen möglich (Vererbung)
internal	Der Zugriff ist innerhalb der Assembly (dll, exe) möglich, in der die Deklaration erfolgt.
protected	Der Zugriff ist innerhalb einer Assembly sowie innerhalb abgeleiteter Klassen möglich.
public	Der Zugriff ist von überall möglich.

Wegen des **Geheimnisprinzips** sollten die Attribute einer Klasse grundsätzlich private gesetzt sein. Ein Zugriff auf

Instanzattribute
Methoden bzw.
eine bessere

die Inhalte der
sollte nur über
Properties erfolgen, um
Kontrolle zu erhalten.



Die finalize-Methode

In C# gibt es ein spezielles Sprachkonstrukt, das an die Destruktoren von C++ erinnert. Dieses Konstrukt wird durch eine Tilde (~), gefolgt vom Klassennamen deklariert. Das in Listing 4 gezeigte Codebeispiel soll dies verdeutlichen.

```
public class A
{
    ~A()
    {
        //some finalization code
    }
}
```

Es sei vermerkt, dass es sich hierbei um keinen Destruktor handelt, denn dieser Code wird vom C#-Compiler durch den in Listing 5 gezeigten Code ersetzt.

```
public class A
{
    protected override void Finalize()
    {
        //some finalization code
        base.Finalize();
    }
}
```

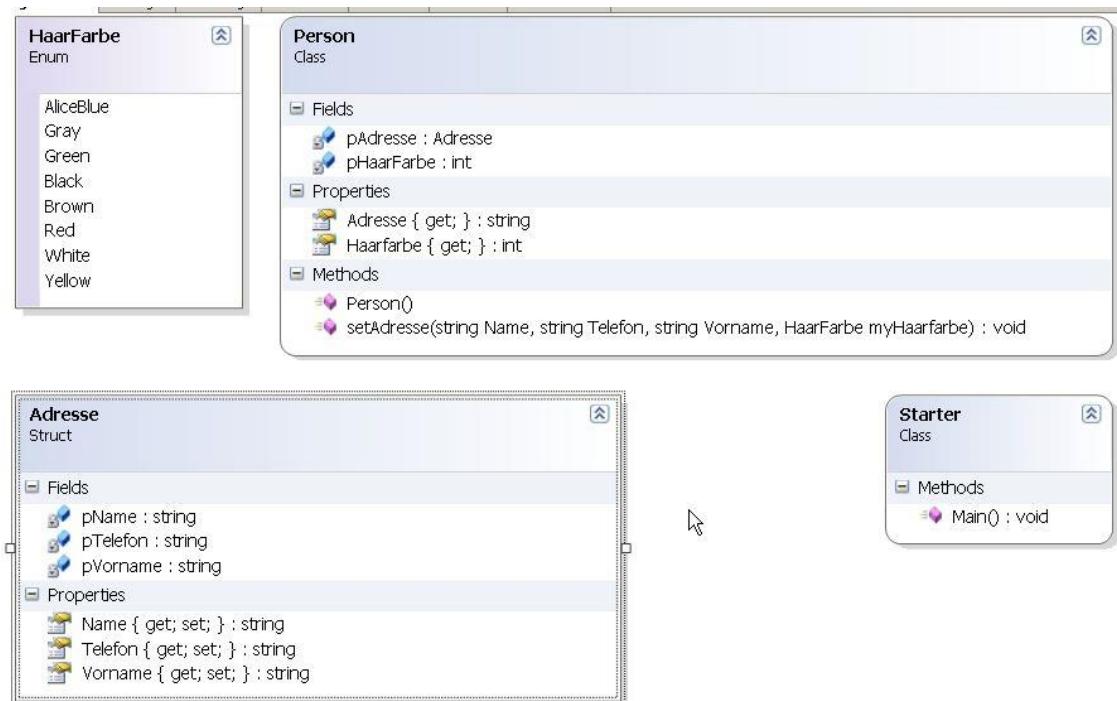
Doch wozu braucht man eigentlich die Finalize-Methode ? In der Finalize-Methode sollen all jene Ressourcen freigegeben werden, die der Garbage Collector verwalten kann ("managed Code").

Aufgaben

- Structures und Klassen sind sich sehr ähnlich. Wo liegen die Unterschiede?
- Finden Sie heraus, was folgender Quellcode ausgibt und wie er im .net-Framework implementiert wurde.

```
int farbe = (int) System.Drawing.KnownColor.AliceBlue;
Console.WriteLine(farbe);
```

- Implementieren Sie analog zum vorher gezeigten Quellcode folgendes Klassendiagramm



Die Methode Main() der Klasse Starter soll ein Person-Objekt mit allen wesentlichen Informationen erzeugen. Anschließend soll sie per Console.WriteLine() dessen Adresse und die Haarfarbe ausgeben.

- Fügen Sie der Klasse Person in der obigen Abbildung einen neuen Konstruktor hinzu, der als Parameter eine Struktur **Adresse** erwartet und diese seiner internen Struktur zuweist.
- Fügen Sie der Klasse Person ein statisches Feld Anzahl hinzu (Datentyp int). Bei jedem Erzeugen eines neuen Objektes soll dieses Feld die Anzahl der bisher insgesamt gebildeten Objekte aufnehmen.

6. Welche Ausgabe erzeugt folgender Quellcode. Begründen Sie !!

```
// struct2.cs
using System;

class TheClass
{
    public int x;
}

=====
struct TheStruct
{
    public int x;
}

=====
class TestClass
{
    public static void structtaker(TheStruct s)
    {
        s.x = 5;
    }

    public static void classtaker(TheClass c)
    {
        c.x = 5;
    }

    public static void Main()
    {
        TheStruct a = new TheStruct();
        TheClass b = new TheClass();
        a.x = 1;
        b.x = 1;
        structtaker(a);
        classtaker(b);
        Console.WriteLine("a.x = {0}", a.x);
        Console.WriteLine("b.x = {0}", b.x);
    }
}
```

7. Funktioniert folgender Quellcode (es sei angenommen, es gibt eine Klasse Person)

```

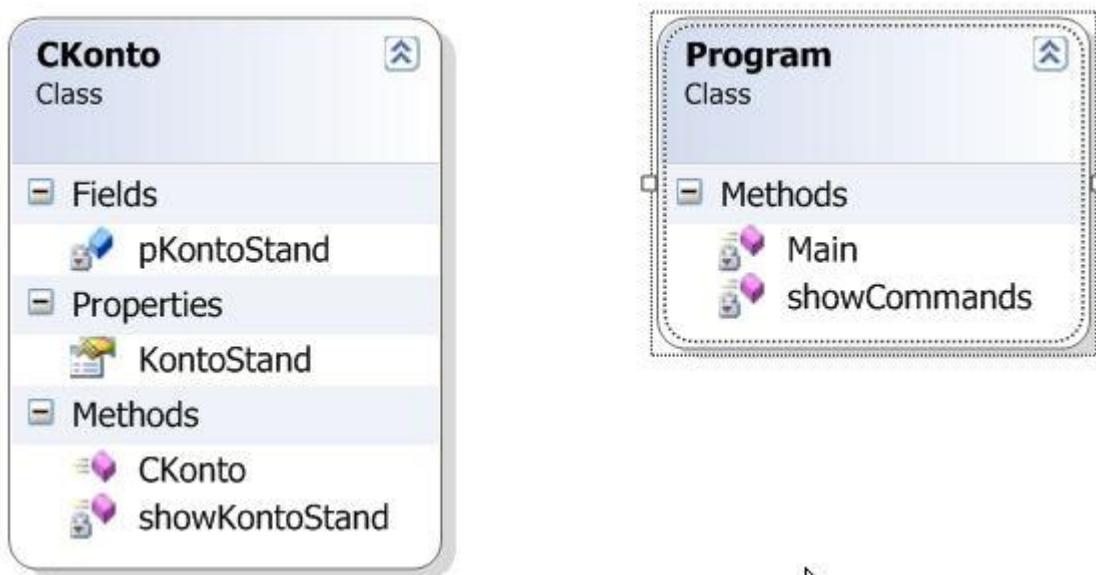
class UserClass
{
    public void erzeuge()
    {
        Factory Fabrik = new Factory();
        Person Sierl = Fabrik.createPerson();
    }
}
=====
class Factory
{
    public Factory()
    {

    }

    public Person createPerson()
    {
        return new Person();
    }
}

```

8. Erstellen Sie eine Konsolenanwendung, um ein Programm für eine einfache Kontoführung zu erstellen.



Entwickeln Sie eine in einer separaten Datei gespeicherte Klasse **CKonto**. Als einziges Feld soll diese Klasse eine als **private** deklarierte Variable für den Kontostand besitzen.

Der Zugriff auf den Kontostand erfolgt ausschließlich über eine als `public` deklarierte Eigenschaft für den Schreib-/Lesezugriff.

Implementieren Sie einen Konstruktor, über den Sie den Kontostand mit einem Wert von 5000 initialisieren.

Die Benutzerschnittstelle sollte so aufgebaut sein, dass sowohl zu Beginn als auch nach jeder Ein- bzw. Auszahlung der aktuelle Kontostand angezeigt wird. Mit einer `switch`-Anweisung können Sie dem Benutzer ermöglichen, Ein- bzw. Auszahlungen vorzunehmen sowie das Programm zu beenden.

9. Wir betrachten eine Bank und ihre Kunden. Ein Kunde kann genau ein Konto eröffnen.

Für jeden Kunden werden dessen Name, Adresse und das Datum der Kontoeröffnung erfasst. Bei der Kontoeröffnung muss der Kunde gleich eine Einzahlung vornehmen. Für jedes Konto wird ein individueller Habenzins und auch ein individueller Sollzins festgelegt; außerdem besitzt jedes Konto eine eindeutige Kontonummer.

Ein Kunde kann Beträge einzahlen und abheben, er kann das Konto allerdings nicht überziehen. Des Weiteren werden Zinsen gutgeschrieben und evtl. Überziehungszinsen abgebucht. Ein Kunde kann sein Konto wieder auflösen. Bei der Auflösung des Kontos hört er auf, Kunde zu sein.

Erstellen Sie die notwendigen Klassen zur Abbildung des Sachverhaltes.

Welche Probleme können Sie zur Zeit noch nicht lösen.

10. Stellen Sie fest, in welchem Speicherbereich die Objekte des folgenden Quellcodes alloziert werden.

```
public class A
{
    public byte[] buffer = new byte[10000]
```

Lösung gibt es [hier](#)

Weitergehende Aufgaben

- [UebungZuEnums.doc](#) ÜbungzuEnums.doc
- **Übungsaufgabe Teilnehmer_Tagung.**
- Ostern rückt näher und es ist Zeit, sich um die Wunschzettel für den Osterhasen zu kümmern.

Realisieren Sie einen Wunschzettel für den Osterhasen mit unterschiedlichen Artikeln wie folgt:

- Buch: mit Autor und Titel
- Elektronik-Artikel: mit Firma und Modell
- Gewand mit Größe und folgenden Untertypen
- Hose
- Hemd

Jeder Artikel hat einen Preis.

Realisieren Sie Methoden für den Zugriff auf den Preis und auf die unterschiedlichen Eigenschaften der Artikel, Konstruktoren, die eine sinnvolle Konstruktion der Artikel erlauben, und Methoden `toString()` für die String-Darstellung der unterschiedlichen Artikel.

Realisieren Sie dann eine Wunschzettel (Klasse `Wishlist`) zur Verwaltung der Geschenkartikel. Es soll möglich sein:

- Gewünschte Artikel anzufügen
- Den Gesamtbetrag zu berechnen
- Den gesamten Wunschzettel auf der Konsole auszugeben (`printWishlist()`)
- Die Artikel einer bestimmten Art auf der Konsole auszugeben (z.B. `printBooks()`, `printClothes()`,). Verwenden Sie für diese Methoden den typeOf-Operator.

Gehen Sie bei der Lösung wie folgt vor:

Skizzieren Sie Klassen und die Methoden in Prosa (kurze Beschreibung)

Implementieren Sie die Klassen und die Methoden in C# oder einer Ihnen bekannten objekt-orientierten Programmiersprache.

Stellen Sie einen Testplan auf und testen Sie das Programm

Lösung gibt es [hier](#)

- Erstellen Sie eine Klasse, von der sich genau ein einziges Objekt instanziieren lässt. Ein wiederholtes Aufrufen des new-Operators sollte ein bereits vorhandenes Objekt zurückgeben, ansonsten sollte ein neues Objekt erzeugt werden.
- Klären Sie mit Hilfe der MSDN die Begriffe **abstrakte Klasse** bzw. **Interface** und **Vererbung**. Versuchen Sie dann anschließend, die ppt-Datei zur SimUDuck-Anwendung zu verstehen und das Klassendiagramm nachzuvollziehen.



Vererbung

Einfache Vererbung

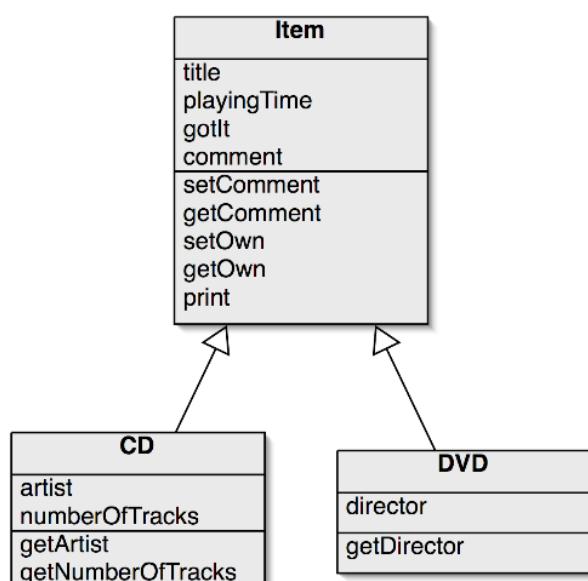
Beim Erstellen von Klassen wird man häufig Abhängigkeiten/Gemeinsamkeiten zwischen Klassen erkennen. Irgendwie gehören sie zusammen aber doch auch wieder nicht. Die OOA löst diese Problematik durch das Konzept der Vererbung.

Es bedeutet, dass Klassen Attribute und Fähigkeiten anderer Klassen übernehmen und gleichzeitig erweitern können. Häufig wird man auch erst im Laufe der Analyse eine bestehende Klasse in mehrere Klassen unterteilen wollen.

CD
title
artist
numberOfTracks
playingTime
gotIt
comment
setComment
getComment
setOwn
getOwn
print

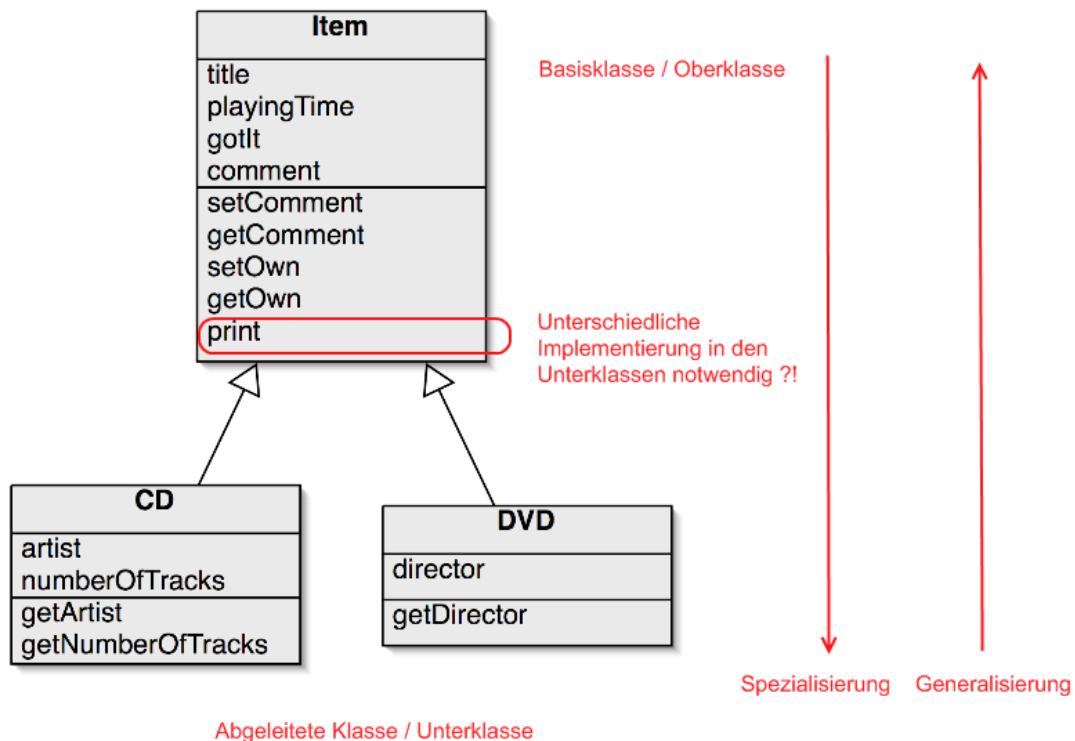
DVD
title
director
playingTime
gotIt
comment
setComment
getComment
setOwn
getOwn
print

In diesem Beispiel verfügen die Klasse CD und DVD über gemeinsame Attribute und Methoden. Sie werden in einer Oberklasse Item ausgelagert. Zusätzliche Informationen und Fähigkeiten bleiben in den spezialisierten Unterklassen.



Im Zusammenhang mit Vererbung existieren einige Begriffe:

- Basisklasse: Die Klasse, die alle Gemeinsamkeiten aufnimmt (in unserem Fall Item)
- Abgeleitete Klasse / Unterklassen: Die Klassen, die spezielle Attribute und Methoden aufnehmen (in unserem Fall CD und DVD)
- Generalisierung: Untersuchung des Vererbungsaspektes von den Speziellen Klassen zu den Allgemeinen Klassen
- Spezialisierung: Untersuchung des Vererbungsaspektes von den Allgemeinen Klassen zu den Speziellen Klassen



Es sind folgende Dinge zu beachten:

- Informationen werden in Ober- und Unterklassen gehalten um Redundanz zu vermeiden.
- Die Unterklassen brauchen aber Zugriffsmöglichkeiten auf Attribute und Methoden der Oberklasse
- Die Unterklassen halten über den Vererbungsmechanismus den Vertrag der Oberklasse. Konkrete Instanzen der Unterklassen können überall dort benutzt werden, wo Instanzen der Oberklasse erwartet werden. In einem solchen Falle ist dann aber lediglich der Zugriff auf die Schnittstelle der Oberklasse möglich. Die Unterklassen sind somit vom gleichen Typ wie die Oberklasse. Eine DVD ist damit auch ein Item.
- Die Oberklasse hat im Allgemeinen keine Kenntnis darüber, dass es abgeleitete Unterklassen gibt.



Implementierung

Bei der Implementierung des obigen Beispiels ist zu beachten, dass

- insgesamt 3 Klassen erzeugt werden müssen
- die Unterklassen Zugriff auf Attribute und Methoden der Oberklasse haben müssen und dementsprechend Zugriffsmodifizier (internal) gewählt werden müssen
- gewisse Methoden in den Unterklassen überschrieben werden müssen. Hierzu muss in der Unterklasse mit dem Schlüsselwort virtual, in den Unterlassen mit dem Schlüsselwort override gearbeitet werden.

Oberklasse

Die Klasse Item hält 4 Attribute und 2 getter/setter-Methoden. Der Konstruktor wird verwendet, um Titel und Spielzeit zu setzen. Die print()-Methode gibt den Titel und die Spielzeit aus.

```
public class Item
{
    private string title;
    private double playingTime;
    private bool gotIt;
    private string comment;

    public Item(string _title, double _playingTime)
    {
        this.title = _title;
        this.playingTime = _playingTime;
    }

    public string getComment()
    {
        return this.comment;
    }

    public void setComment(string _comment)
    {
        this.comment = _comment;
    }

    public void setOwn(bool _own)
    {
        this.gotIt = _own;
    }

    public bool getOwn()
    {
        return this.gotIt;
    }
}
```

```

        public void print()
    {
        Console.WriteLine(this.title + ", Spielzeit: " +
this.playingTime.ToString());
    }
}

```

Unterklasse CD

Eine erste Implementierung sieht wie folgt aus:

```

public class CD : Item
{
    private string artist;
    private int numberOfTracks;

    public CD(string _artist, int _numberOfTracks)
    {
        this.artist = _artist;
        this.numberOfTracks = _numberOfTracks;
    }

    public string getArtist()
    {
        return this.artist;
    }

    public int getNumberOfTracks()
    {
        return this.numberOfTracks;
    }
}

```

Dies führt aber zu folgenden Problemen:

```

1 public class CD : Item
2 {
3     private string artist;
4     private int numberOfTracks;
5
6     public CD(string _artist, int _numberOfTracks)
7     {
8         this.artist = _artist;
9         this.numberOfTracks = _numberOfTracks;
10    }
11
12    public string getArtist()
13    {
14        return this.artist; return this.
15    }
16
17    public int getNumberOfTracks()
18    {
19        return this.numberOfTracks;
20    }
21 }

```

Fehler

1 Fehler | 0 Warnungen | 0 Meldungen

Zeile Beschreibung Datei

21 "CD_DVD_ITEM.Item" enthält keinen Konstruktor, der 0-Arumente akzeptiert. (CS1729) CD.cs

Konstruktorproblematik

Beim Erzeugen einer Unterklasse wird es aufgrund der Vererbung auch notwendig, eine Instanz der Oberklasse zu erzeugen. Da diese jedoch in unserer Implementierung keinen parameterlosen Konstruktor zulässt, erzeugt der Compiler einen Fehler. Hätten wir einen parameterlosen Konstruktor, hätte der Compiler keinen Fehler gemeldet.

Daran ist prinzipiell auch nichts Verwerfliches, da wir in der Oberklasse über den Konstruktor das Setzen von Zuständen implementiert haben, die für unser Verhalten wichtig sind (Titel und Spielzeit). Auch eine CD sollte über diese Informationen verfügen. Der Konstruktor von CD muss deshalb diese Informationen an den Konstruktor der Oberklasse Item weiterreichen.

Dies erfolgt durch das Weiterleiten im Konstruktoraufzug:

```

public class CD : Item
{
    private string artist;
    private int numberOfTracks;

    //Weiterleiten an den Konstruktor der Oberklasse
    //Konstruktorsignatur der Oberklasse
    public CD(string _artist, int _numberOfTracks, string _title, double _playingTime) : base( _title,
    _playingTime)
    {
        this.artist = _artist;
        this.numberOfTracks = _numberOfTracks;
    }
}

```

```
public class CD : Item
{
    private string artist;
    private int numberOfTracks;

    public CD(string _artist, int _numberOfTracks, string _title, double _playingTime) : base(_title, _playingTime)
    {
        this.artist = _artist;
        this.numberOfTracks = _numberOfTracks;
    }

    public string getArtist()
    {
        return this.artist;
    }

    public int getN
    {
        get { return this.numberOfTracks; }
        set { this.numberOfTracks = value; }
    }

    public string getComment()
    {
        return this.comment;
    }

    public void setComment(string value)
    {
        this.comment = value;
    }

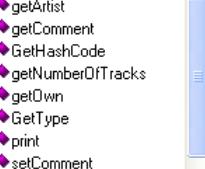
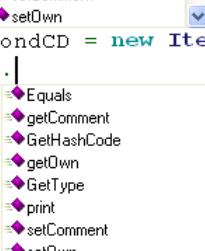
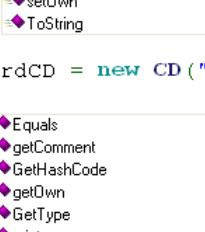
    public override string ToString()
    {
        return string.Format("CD: {0} - {1} - {2} - {3}", artist, title, playingTime, comment);
    }
}
```

Korrekte Implementierung des Konstruktors für CD; Aufruf eines Konstruktors der Oberklasse

öffentliche Methoden der Oberklasse sind sichtbar,
was ist aber mit den privaten Attributen ?

Vererbung als Vertrag

Durch den Vererbungsmechanismus sind die Unterklassen in der Lage, überall dort einsetzbar zu sein, wo eigentlich Objekte der Oberklasse verwendet werden sollen (LSP - Liskov Substitution Problem)

<pre>CD firstCD = new CD ("Calogero", 12, "Remember Me", 60.00); firstCD.</pre>		<p>firstCD ist Instanz von CD</p> <p>firstCD hat alle öffentlichen Methoden der Ober- und der Unterklasse</p> <p>Vererbung</p>
<pre>Item secondCD = new Item ("Last Dance", 12.00); secondCD.</pre>		<p>secondCD ist Instanz von Item</p> <p>secondCD kennt nur die Methoden der Oberklasse</p>
<pre>Item thirdCD = new CD ("Hannah Montana", 10, "Bst Of both worlds", 30.00); thirdCD.</pre>		<p>Was ist thirdCD ?</p> <p>thirdCD ist zwar eine Instanz von CD (new CD(...)), hält aber nur den Vertrag von Item (Item thirdCD)</p> <p>Nur die öffentlichen Methoden von Item sind deshalb verfügbar.</p> <p>Ist das gut oder schlecht ?</p>

Zugriff auf Attribute und Methoden der Oberklasse

Durch die Vererbung wird zunächst der Zugriff auf öffentliche Methoden und Attribute gewährt. Private Attribute und Methoden sind durch das Prinzip der Kapselung weiter geschützt.

Will man den Unterklassen Zugriff auf diese Elemente gewähren, so müssen die Zugriffsmodifizierer der Oberklasse geändert werden.

In C# stehen folgende Modifizierer zur Verfügung:

URL: ms-help://MS.LHSMDK.1033/MS.LHSNETFX30SDK.1033/dv_csharp/html/dc083921-0073-413e-8936-a613e8bb7df4.htm

Collapse All Code: Multiple
C# Language Reference
Accessibility Levels (C# Reference)
[See Also](#) [Send Feedback](#)

Use the access modifiers, [public](#), [protected](#), [internal](#), or [private](#), to specify one of the following declared accessibilities for members.

Declared accessibility	Meaning
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protectedinternal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.

Only one access modifier is allowed for a member or type, except when you use the **protectedinternal** combination.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

In unserem Beispiel könnten die bisher als private deklarierten Attribute der Oberklasse mit dem Schlüsselwort protected ersetzt werden.

```

public class Item
{
    #region private Attribute
    protected string title;
    protected double playingTime;
    protected bool gotIt;
    protected string comment;
    #endregion

    public class CD : Item
    {
        private string artist;
        private int numberOfTracks;

        private new void make()
        {
            this._ir = ir;
            this._t = t;
            this._r = r;
        }

        public CD()
        {
            this._ir = ir;
            this._t = t;
            this._r = r;
        }
    }
}

```

Deklaration von protected in Oberklasse macht Attribute in Unterklasse direkt benutzbar

Überschreiben von Methoden der Oberklasse

Die Methode print() der Klasse Item gibt zur Zeit den Titel sowie die Spielzeit aus.

```

public class Item
{
    private Attribute
    Konstruktor

    getter and setter

    public void print()
    {
        Console.WriteLine(this.title + ", Spielzeit: " + this.playingTime.ToString());
    }
}

```

Dieses Verhalten kann eventuell nicht das sein, was Instanzen der Klasse CD möchten.

Doch wie kann die Ausgabe für Objekte der Klasse CD geändert werden, ohne eine andere Methode benutzen zu müssen. Um es allgemeiner auszudrücken:

Wie kann eine Unterklasse ein anderes Verhalten als die Oberklasse implementieren ?

Die Lösung besteht im Neudefinieren der Methode in den Unterklassen.

The screenshot shows a code editor with the following C# code:

```

public class CD : Item
{
    private string artist;
    private int numberOfTracks;

    public void print()
    {
        Console.WriteLine("Hello World");
    }
}

```

A red arrow points from a callout box to the warning in the error list below:

Eine Neudefinition der print()-Methode führt zu einer Warnung des Compilers. Die print()-Methode der Oberklasse wird ausgeblendet

Fehler

! Zeile Beschreibung

11 "CD_DVD_ITEM.CD.print()" blendet den vererbten Member "CD_DVD_ITEM.Item.print()" aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausble.

Um die Warnung des Compilers zu umgehen, gibt es zwei Möglichkeiten:

Überschreiben mit Hilfe von new

```

public class CD : Item
{
    private string artist;
    private int numberOfTracks;
    public new void print()
    {
        Console.WriteLine("Hello World");
    }
}

```

Als Nebeneffekt ist jedoch zu bemerken, dass beim folgendem Quellcode nicht die Implementierung der Unterklasse aufgerufen wird.

```
Item newCD = new CD(....);
newCD.print()
```

Benutzen des Schlüsselwortes override

Dies setzt allerdings voraus, dass in der Oberklasse die jeweilige Methode als virtual, abstract oder override deklariert wurde

```
public class CD : Item
{
    private string artist;
    private int numberOfTracks;

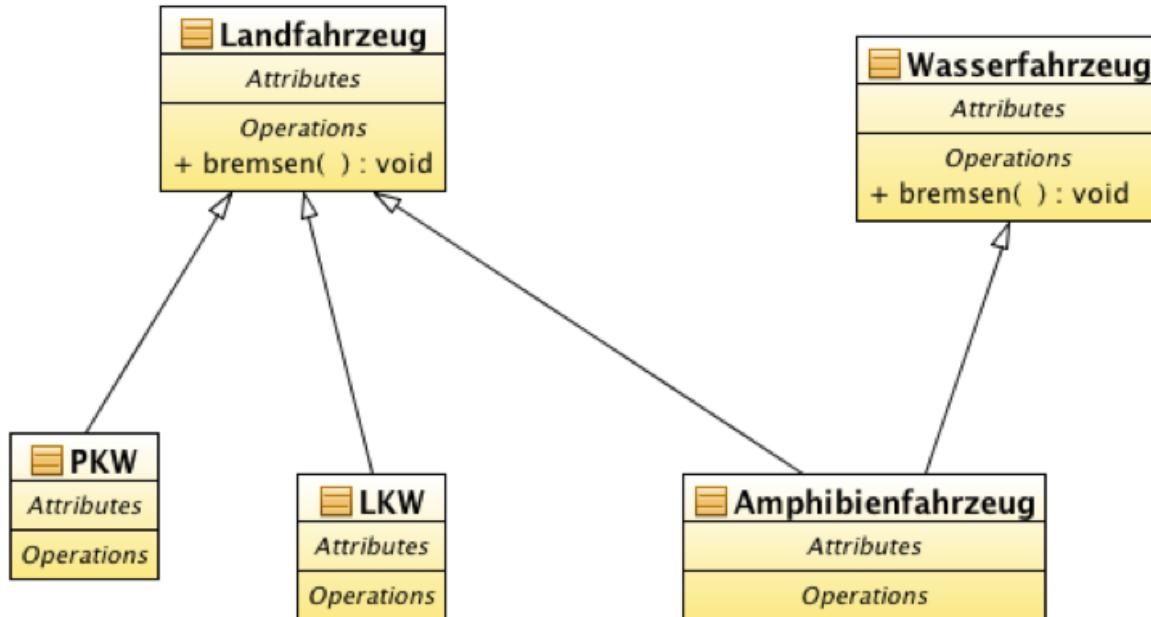
    //Zunächst rufen wir die print()-Methode der Oberklasse auf
    //und fügen anschließend unsere eigene Implementierung hinzu
    public override void print()
    {
        base.print();
        Console.WriteLine("Artist: " + this.artist);
        Console.WriteLine("Nr Tracks:" + this.numberOfTracks);

    }
    ....
}

public class Item
{
    ....
    public virtual void print()
    {
        Console.WriteLine(this.title + ", Spielzeit: " +
this.playingTime.ToString());
    }
}
```

Mehrfachvererbung

Mehrfachvererbung erweitert die grundsätzliche Vererbung um die Fähigkeit, von mehreren Oberklassen gleichzeitig ableiten zu können. Damit ist aber auch bereits die Problematik vor-gegeben.

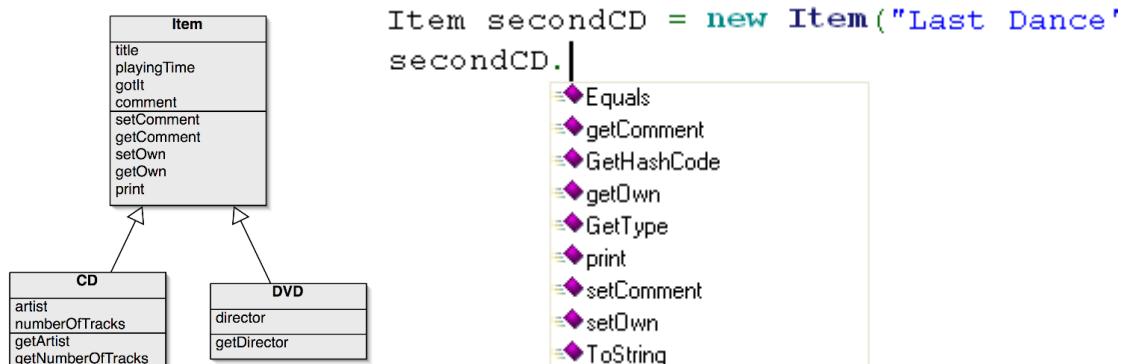


Welche Methode `bremsen()` sollen Amphibienfahrzeuge ausführen ??

Aufgrund dieser Nebeneffekte haben sich die meisten Programmiersprachen vom Konzept der Mehrfachvererbung abgewendet, ohne jedoch die prinzipiellen Vorteile der Mehrfachverer-bung mit Hilfe von Interfaces nicht zu implementieren.

Abstrakte Klasse

Die Implementierung des vorherigen Beispiels CD/DVD hatte einen großen Nachteil:



Es gibt eigentlich keine Instanzen von Item, sie waren aber jederzeit erzeugbar.

Das Konzept der Abstrakten Klasse kann dieser Problematik begegnen.

- Definition der Klasse oder einer Methode als `abstract`
- Damit ist eine Instanziierung dieses Typs ausgeschlossen
- Klasse kann dennoch konkrete Implementierungen von Methoden sowie Attribute besitzen
- Methoden, von denen bekannt ist, dass Sie die Unterklassen überschreiben müssen, werden als abstrakt definiert. Sie bestehen lediglich aus der Methodensignatur.
- Damit müssen ableitende Unterklassen diese Methoden implementieren, ansonsten können keine Instanzen der Unterklassen gebildet werden.

Eine Implementierung der Klasse Item als abstrakte Klasse kann wie folgt aussehen.

```

using System;

namespace CD_DVD_ITEM
{
    public abstract class Item
    {
        #region private Attribute
        protected string title;
        protected double playingTime;
        protected bool gotIt;
        protected string comment;
        #endregion

        #region Konstruktor
        public Item(string _title, double _playingTime) {
            this.title = _title;
            this.playingTime = _playingTime;
        }
        #endregion

        #region getter and setter

        public string getComment() {
            return this.comment;
        }

        public void setComment(string _comment) {
            this.comment = _comment;
        }

        public void setOwn(bool _own) {
            this.gotIt = _own;
        }

        public bool getOwn() {
            return this.gotIt;
        }
        #endregion

        public abstract void print();
    }
}

```

Da die Methode print() jetzt als abstract definiert wurde, müssen die konkreten Klassen CD und DVD diese Methode implementieren.

```
public class CD : Item
{
    private string artist;
    private int numberOfTracks;

    public override void print()
    {
        //das geht jetzt natürlich nicht mehr
        //base.print();
        Console.WriteLine("Artist: " + this.artist);
        Console.WriteLine("Nr Tracks: " + this.numberOfTracks);
    }
}
```

Interface

Das Konzept des Interfaces führt die Idee der abstrakten Klasse weiter. Während die abstrakte Klasse durchaus in der Lage ist, eigene Attribute und Implementierungen von Methoden zu besitzen, reduziert sich das Interface im Prinzip lediglich auf die Deklaration von leeren Methodensignaturen. Diese werden dann von konkreten Klassen implementiert. Eine Implementierung mehrerer Interfaces ist möglich (siehe Mehrfachvererbung).

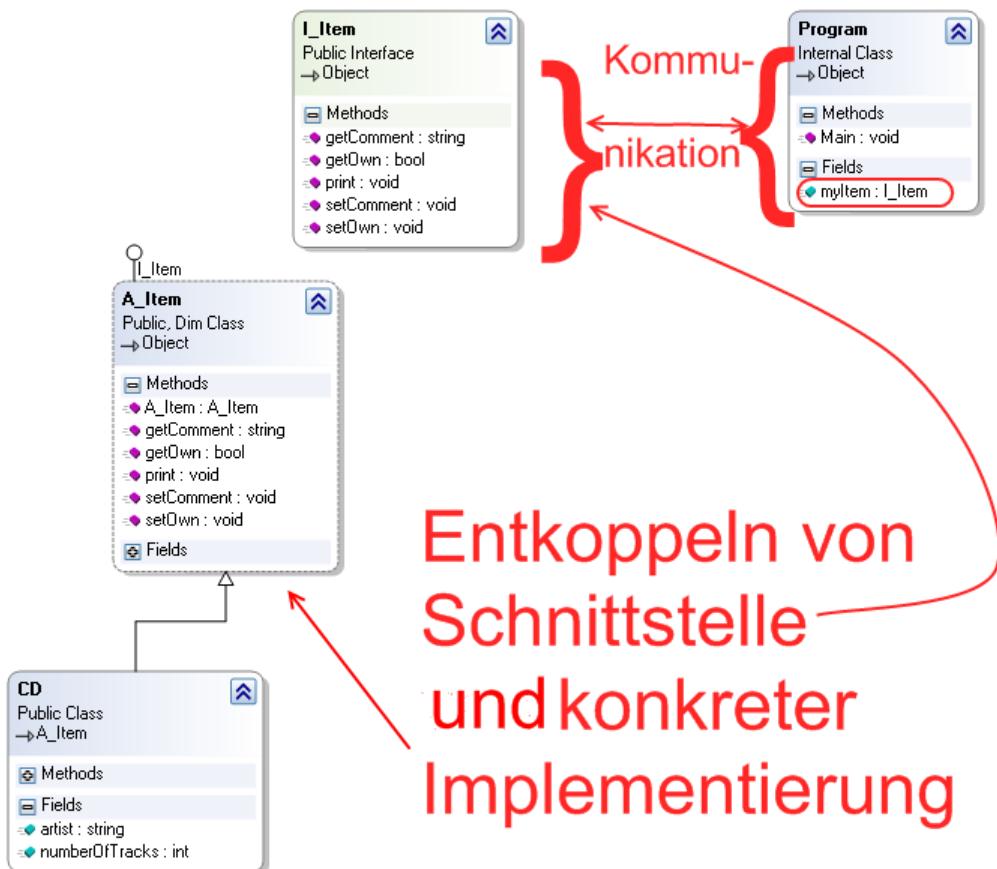
Interfaces sind dann das geeignete Konzept, wenn es darum geht, die Schnittstellen von Klassen herauszuarbeiten, d.h. die Frage zu beantworten:

Wer kommuniziert mit mir über welche Methoden?

Die Implementierung der bisherigen (abstrakten) Klasse Item zu einem Interface sieht wie folgt aus:

```
using System;

namespace CD_DVD_ITEM
{
    public interface I_Item
    {
        string getComment();
        void setComment(string _comment);
        void setOwn(bool _own);
        bool getOwn();
        void print();
    }
}
```



Polymorphie / Late vs. Early Binding

Polymorphie bedeutet „Vielgestaltigkeit“. In der OOP wird der Begriff zunächst dazu genutzt, um darzustellen, dass verschiedene Klassen gleiche Methoden besitzen können. Diese Methoden können aber verschieden implementiert sein und damit eben auch verschiedene Dinge tun.

Unter Late/Early Binding versteht man den Zeitpunkt, wann der Compiler bzw. die Runtime-Umgebung festlegen kann, welche Methode denn nun tatsächlich aufgerufen werden soll.

Die wahre Bedeutung der Polymorphie und des Late/early Bindings erschließt sich durch die Vererbung und dem Nutzen von Klassen der Vererbungshierarchie von den jeweiligen Klienten.

Wir greifen das Beispiel der CD/DVD - Applikation auf und erweitern es um die Klasse DVD.

```

namespace CD_DVD_ITEM
{
    class Program
    {
        static I_Item[] myItems = new I_Item[10];
        static CD myCD1;
        static DVD myDVD1;

        public static void Main(string[] args)
        {
            //Early Binding
            //Bereits der Compiler kann erkennen, welche print()
            //Methode aufgerufen werden soll
            //Konkreter Typ ist bekannt
            myCD1 = new CD("Calogero", 12, "Remember Me", 60.00);
            myCD1.print();
            myDVD1 = new DVD("Steinam", "Hannibal Lector", 2.45);
            myDVD1.print();

            myItems[0] = myCD1;
            myItems[1] = myDVD1;

            //Vorbereiten des Late Bindings
            //Zufälliges Belegen der Slots des Arrays
            Random number = new Random(1234);
            for(int y = 2; y<=9; y++) {
                int zahl = number.Next(0, 10);
                if(zahl <=5) {
                    myItems[y] = myCD1;
                }
                else{
                    myItems[y] = myDVD1;
                }
            }
            //Late-Binding
            //Der Compiler kann nicht erkennen, welche
            //print() - Methode aufgerufen werden kann.
            //Erst zur Laufzeit kann dies entschieden werden.
            for(int i = 0; i<=9; i++) {
                myItems[i].print();
            }
            Console.ReadKey();
        }
    }
}

```

Beziehungen

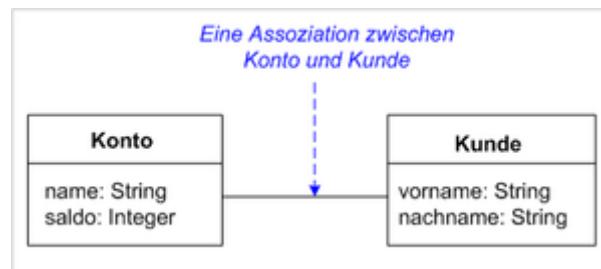
Inhalte:

- Assoziation
- Aggregation/Komposition
- Assoziative Klasse
- Navigierbarkeit/Multiplizität/Rolle

Während die Vererbung eine Typ-basierte Beziehung darstellt, sind Beziehungen zwischen Klassen nicht nur auf diese Art herstellbar. Objekte kommunizieren per Nachrichten mit anderen Objekten; häufig halten sie sich auch andere Objekte zur Erfüllung bestimmter Aufgaben. In all diesen Szenarien spricht man von **Beziehungen** zwischen Objekten

Assoziation

Unter Assoziation versteht man eine Beziehung zwischen zwei oder mehreren Typen. Unterstehende Abbildung zeigt dies in seiner einfachsten Form, einer Linie zwischen zwei Klassen. Sie drückt eine „Sich-Kennen“-Eigenschaft zwischen Objekten der jeweiligen Klasse aus.



Die Linie kann durch verschiedene Elemente in ihrem Informationsinhalt erweitert werden.

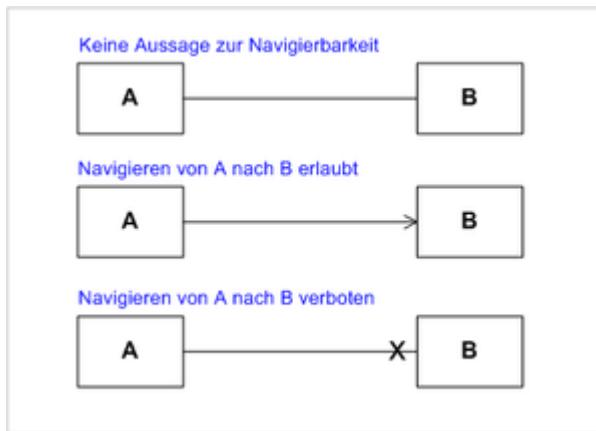
Multiplizität

Wie im Entity-Relationship-Modell kann die Assoziation um die Angabe erweitert werden, wieviele Objekte einer Seite jeweils mit einem Objekt der anderen Seite in Beziehung stehen. Dies wird durch eine Zahl bzw. ein * oder einer Min-Max-Notation zum Ausdruck gebracht.



Navigierbarkeit

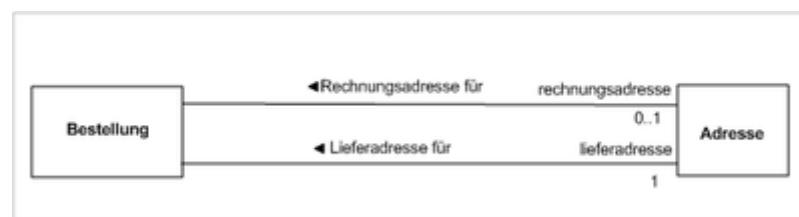
Eine Assoziation bildet eine Art Brücke zwischen zwei Typen: startet man bei der Instanz des einen beteiligten Typs kann man über eine Objektbeziehung zur Instanz des zweiten Typs navigieren. Die Navigierbarkeit von Assoziationsenden wird mit Hilfe eines → dargestellt und kann durch ein –x eingeschränkt werden. Dabei unterscheidet man drei Arten, wie die Navigierbarkeit festgelegt werden kann:



- Keine Aussage (keine Pfeile)
- Erlaubte Navigation
- Nicht erlaubte Navigation

Rollen

Gerade wenn es mehrere Assoziationen zwischen zwei Klassen gibt, kann der Informationsgehalt durch die Darstellung der Rolle erweitert werden. Die Rolle bezeichnet die Funktion, die ein Teilnehmer der Beziehung gegenüber der anderen Seite spielt. Sie wird an das jeweilige Assoziationsende geschrieben.



Im vorliegenden Beispiel gibt es zwei Beziehungen zwischen Bestellung und Adresse. Durch die unterschiedlichen Rollennamen kann man sie gut unterscheiden. Die beiden kleinen Dreiecke neben dem Assoziationsnamen unterstützen die Leserichtung.

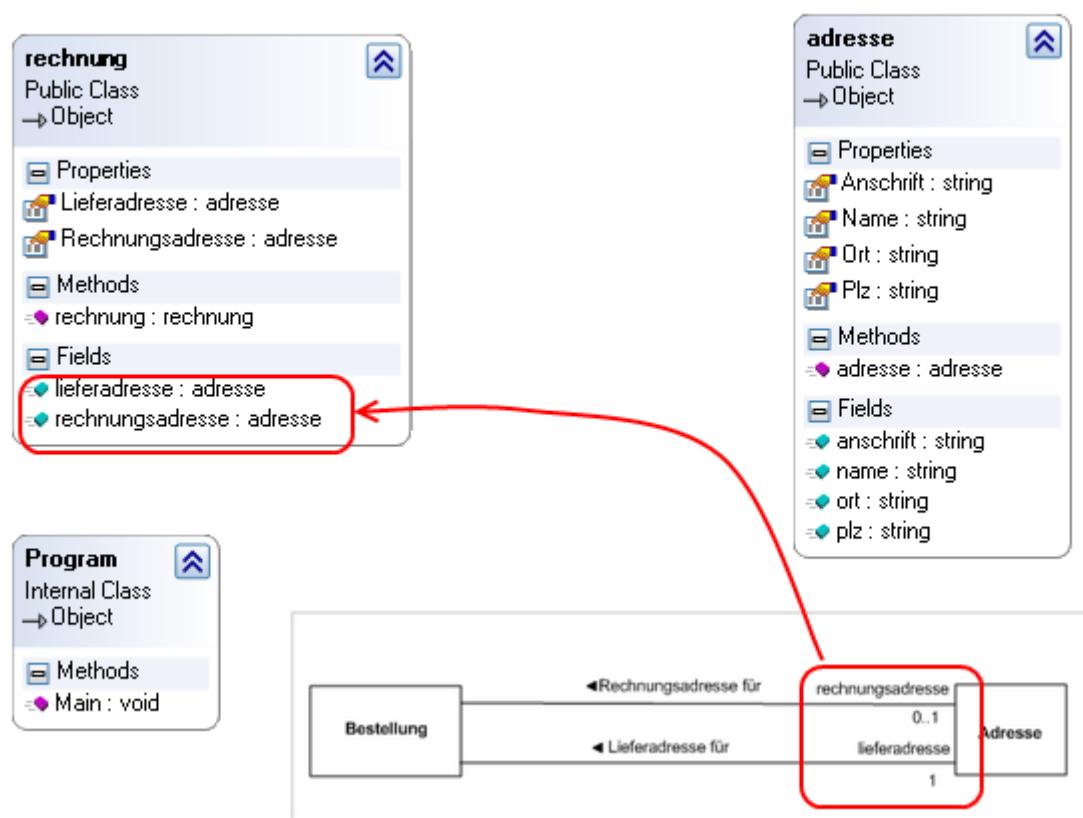
Adresse [ist] Rechnungsadresse für Bestellung

Implementierung

Obenstehendes Beispiel von Adresse und Rechnung soll als Ausgangsgrundlage einer Implementierung dienen.

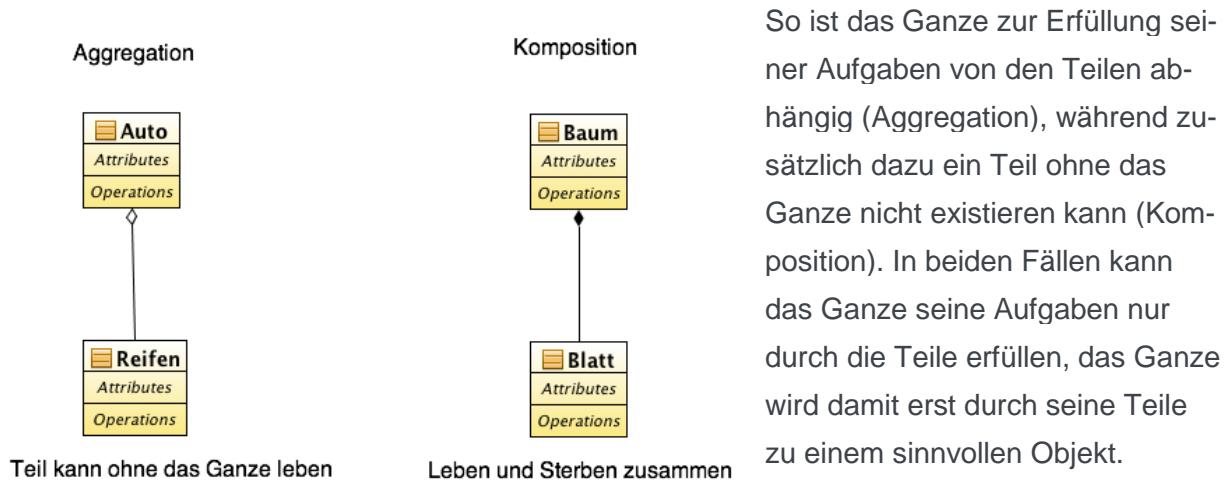
Folgende Grundüberlegungen können vorgenommen werden:

- Da keine Navigation angegeben ist, nehmen wir an, dass die Rechnung die Adresse kennt, nicht jedoch umgekehrt
- Ein Rechnungsobjekt hat 0 oder eine Rechnungs- bzw. Lieferadresse, d.h. sie muss jeweils eine Referenz auf maximal eines dieser beiden Adressen haben; dies kann durch eine entsprechende Referenzvariable erreicht werden.
- Die Rollennamen sind geeignete Namen für die Namen der Referenzvariablen.



Aggregation/Komposition

Diese beiden Begriffe spiegeln eine besondere Art von Beziehungen zwischen Objekten wieder, nämlich sog. Teil-Ganzes-Beziehungen zwischen den Klassen. Es wird damit auch eine gewisse Abhängigkeit zwischen den Klassen ausgedrückt.



Die Frage, ob eine Beziehung als Aggregation oder Komposition gestaltet werden soll, hängt häufig vom jeweiligen Problem ab und kann nicht immer gleich beantwortet werden.

Die Entscheidung für Aggregation und Komposition hat aber gewisse Konsequenzen für die Implementierung, insbesondere bei der Komposition:

- Das Ganze darf bei der Komposition die alleinige Referenz auf das Teil besitzen
- Deshalb muss das Ganze das Teil selbst erzeugen
- Das Ganze darf keine Referenzen auf das Teil an Klienten weitergeben, da sonst die Kontrolle verloren gehen kann

Aufgabe

- **SimuDuck.**
- Gefangenendilemma

Lösung

“Gefangenendilemma” ist ein rundenbasiertes Spiel für zwei Spieler.

Zwei Verbrecher werden gemeinschaftlich einer Tat beschuldigt. Beide werden getrennt verhört. Beide haben die Möglichkeit, zu schweigen oder auszusagen, d.h. das gemeinsam begangene Verbrechen zuzugeben. Der Staatsanwalt bietet beiden einen Deal an:

- Schweigen beide Verbrecher, erhalten beide 2 Haftjahre (Strafpunkte), da die Beweise nicht für mehr ausreichen.
- Sagen beide Verbrecher aus, erhalten beide 4 Haftjahre.
- Schweigt ein Verbrecher und der andere sagt aus, so erhält der aussagende Verbrecher 1 Haftjahr, der schweigende erhält 5 Haftjahre.

Das Dilemma besteht darin, dass jeder Verbrecher für sich günstiger dasteht, wenn er aussagt (4 gegen 5 bzw. 1 gegen 2 Haftjahre), die optimale Lösung für beide jedoch wäre, dass beide schweigen (4 Haftjahre gesamt). Dieses Dilemma lässt sich auf viele andere Situationen übertragen.

Noch interessanter wird das Spiel, wenn man es über mehrere Runden wiederholt. In jeder Runde haben beide Spieler die Möglichkeit, zu betrügen (auszusagen) oder zu kooperieren (schweigen). Nach jeder Runde werden Strafpunkte nach dem folgenden Schema verteilt:

- A und B kooperieren: Jeder erhält 2 Strafpunkte
- A und B betrügen: Jeder erhält 4 Strafpunkte
- A kooperiert und B betrügt: A erhält 5 Strafpunkte, B erhält 1 Strafpunkt.
- B kooperiert und A betrügt: B erhält 5 Strafpunkte, A erhält 1 Strafpunkt.

Jeder Spieler kennt die Ergebnisse der vorigen Runden und hat die Gelegenheit, sich für vergangenes Unrecht zu rächen oder auch zu vertrauensselige Gegenspieler gnadenlos auszunutzen. Gewonnen hat am Ende der, der nach einer bestimmten Anzahl von Runden die wenigsten Strafpunkte hat.

Auch zwei Computerspieler können dieses Spiel gegeneinander spielen. Hier verfolgt jeder Spieler eine bestimmte Spielweise, die gegen die Spielweise des Gegenübers bestehen muss.



Hinweis

Lesen Sie zunächst den untenstehenden Text und

- Erstellen Sie dann ein Klassendiagramm ihrer Lösung
- Erstellen Sie ein Ablaufdiagramm der Methodenaufrufe
- Schreiben Sie dann eine Simulation des Spiels

Dazu benötigen Sie eine Hauptklasse Dilemma, die die Spieler abwechselnd ziehen lässt und am Ende den Sieger ermittelt. Jeder Spieler besitzt eine bestimmte Spielweise, z.B. HalliGalli, Sprite oder Levenshtein (siehe unten).

Alle Spielstrategien implementieren das Interface **GefSpielweise**, das folgende Methoden umfasst:

- boolean getNextDecision(); Diese Methode gibt als Ergebnis die nächste eigene Spielentscheidung zurück.
- void setOpponentsLastDecision (boolean decision); In dieser Methode wird dem Spieler der letzte Zug des Gegners mitgeteilt.

Schreiben Sie eine Klasse Dilemma mit folgenden Eigenschaften:

- Im Konstruktor werden zwei Objekte der Klasse GefSpielweise übergeben, die die Spielweise der beiden Spieler bestimmen.
- Eine Methode public void play(int n) führt n Spielrunden aus. n ist den Spielern nicht bekannt. Eine Spielrunde besteht aus:
 - Abfrage der Spieldoktrin beider Spieler,
 - Verteilen der Punkte.
 - Beide Spieler bekommen den Zug des gegnerischen Spielers mitgeteilt und überdenken so ihre Taktik für die nächste Runde.

Nach den n Spielrunden wird das Ergebnis auf dem Bildschirm ausgegeben. Es gewinnt der Spieler mit den wenigsten Strafpunkten.

Schreiben Sie außerdem die Strategie-Klassen HalliGalli, Sprite, Levenshtein, Zufall und Muster (und was Sie sonst noch wollen), die alle das Interface GefSpielweise implementieren. Achtung, nicht alle Klassen benötigen die Züge des Gegenspielers, um die eigene Strategie zu bestimmen.



- HalliGalli

In der ersten Runde kooperiert der Spieler, in allen darauf folgenden wird der vorherige Spielzug des anderen Spielers kopiert.

- Sprite

Der Spieler kooperiert solange, bis sein Gegenüber das erste Mal betrügt. Danach ist er so erzürnt, dass er immer betrügt, egal, was der andere Spieler nun macht.

- Levensthein

In der ersten Runde kooperiert der Spieler. Ansonsten kooperiert er nur, wenn in der vorherigen Runde beide Spieler dasselbe gemacht haben.

- Zufall

Zu 50% kooperiert der Spieler, zu 50% betrügt er.

- Muster

Der Spieler verfolgt konsequent das Muster kooperieren/kooperieren/betrügen.

Testen Sie „Ihr Dilemma“. Sie können natürlich auch andere Spielweisen übergeben.

```
public class DasSpiel {
    public static void main(String[] args) {
        //Neues Objekt vom Typ Dilemma mit 2 Spielern
        Dilemma D = new Dilemma(new HalliGalli(), new Muster());
        //100 mal spielen
        D.spiele(100);
    }
}
```

Ergebnis (die genaue Formatierung ist unwichtig): 266:266

Erweitern Sie das Spiel: Nach jedem Zug kann der Spieler entscheiden, welche Spielweise er für die nächste Runde anwenden will. Implementieren Sie alle notwendigen Änderungen des Quellcodes.

Statements und Exceptions

Lernziele:

- Grundlegende Konstrukte des ExceptionHandlings
- try-catch-finally, throws, throw
- Exception Hierarchie
- Eigene Exception-Klassen
- Logging von Exceptions
- Tracing and Profiling, Code Analyzer (FxCop) Tracer und DebugView
- FxCop, <http://www.razorsoft.net/>, <http://dotnet.jku.at/projects/Prof-It/Screenshots.aspx>

Download:

- Siehe **Übung**.
- Siehe **PAP**.
- <http://blogs.msdn.com;brada/archive/2004/03/25/96251.aspx>
- http://www.ftponline.com/vsm/2003_05/online/wagner/
- <http://www.codeproject.com/csharp/errortrapper.asp#xx807498xx>
- <http://www.codeproject.com/csharp/csmverrorhandling.asp>
- <http://www.codeproject.com/dotnet/exceptionbestpractices.asp#xx1038229xx>
- <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconBaseExceptionHierarchy.asp>
- <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp>
- <http://www.codeproject.com/dotnet/unhandledexceptions.asp#xx962361xx>
- <http://www.codeproject.com/dotnet/ExceptionHandling.asp>
- <http://musingmarc.blogspot.com/2005/09/exception-handling-in-net-some-general.html>
- <http://codebetter.com/blogs/karlsequin/archive/2006/04/05/142355.aspx>
- <http://codebetter.com/blogs/karlsequin/default.aspx>
- <http://www.codeproject.com/Articles/9538/Exception-Handling-Best-Practices-in-NET>
- <http://stackoverflow.com/questions/183589/best-practice-for-exception-handling-in-a-windows-forms-application>

Profiler und Tracer

Folgender Webcast beschäftigen sich ab ca. der 50. Minute mit Exceptionhandling [MSDN Teil6](#)

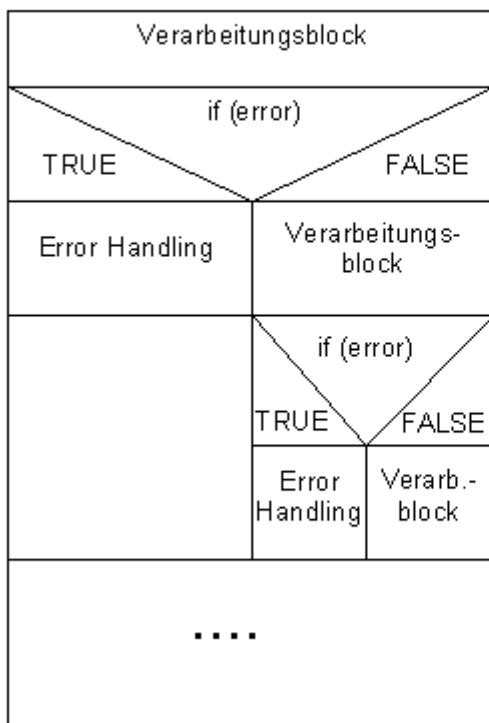
- <http://www.codeproject.com/Articles/5498/TraceTool-12-4-The-Swiss-Army-Knife-of-Trace>

Konzept

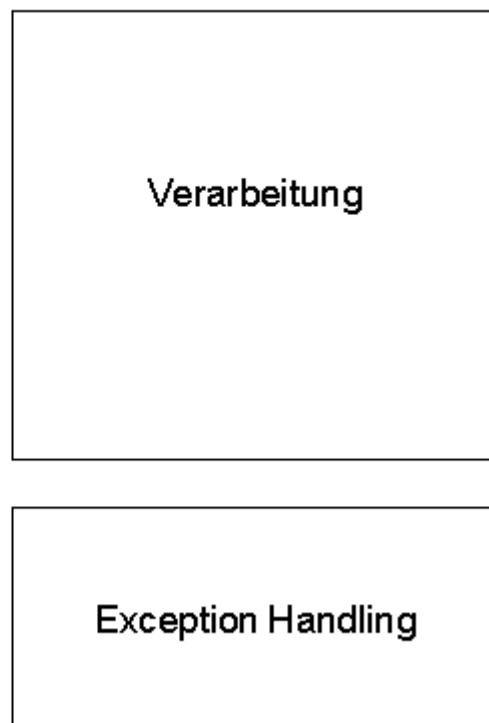
Während der normalen Abarbeitung einer Methode kann zur Laufzeit ein anormales Ereignis auftreten, das die normale Ausführung der Methode unterbricht. Ein solches anormales Ereignis wird als Exception bezeichnet. Ein defensiver Programmierstil gebietet es, auf Ausnahmen vorbereitet zu sein. Das bedeutet, dass man Programmcode zur Erkennung und Behandlung von Exceptions vorsehen muss.

Fehlerbehandlung kann unterschiedlich organisiert werden. Wichtig ist dabei, dass normaler Code und Fehlerbehandlungscode nicht vermischt werden sollten. Moderne Programmiersprachen stellen dazu spezielle Mechanismen zur Verfügung.

Klassisches Programm

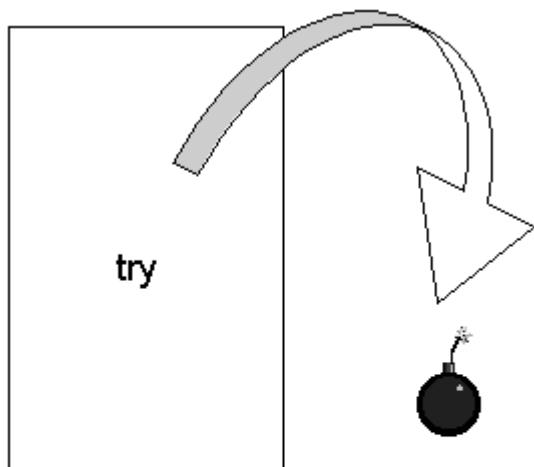


C# / Java / ...



Eine der traditionellen Methoden zur Behandlung von Fehlern ist die Rückgabe eines Fehlercodes durch kritische Funktionen. Der zurückgelieferte Fehlercode muss nach jedem Aufruf geprüft werden. Ziel des Exception Handlings ist es, normalen und fehlerbehandelten Code übersichtlich zu trennen und Ausnahmesituationen sicher zu behandeln.

Implementierung



Mit Hilfe von `try` wird ein Block von beliebigen Anweisungen gekennzeichnet, deren Ausführung versucht werden soll. Eventuell auftretende Exceptions können dann mit Hilfe von `catch` behandelt werden. Ein Exception Handler hat das Ziel, eine Exception zu entschärfen, d.h. eine Methode vom Ausnahmezustand in den Normalzustand zu überführen.

**ohne
Exception Handler**



Eine `try`-Anweisung hat die folgende Struktur:

```

try
{
    //try-Block. Das ist der normale Code
    //in dem Fehler auftreten können
    int c = 0;
    int a = 5 /c;
}

catch (Exceptiontyp1 name1)
{
    .....
    //catch-Block fängt Fehler
    //der Klasse ExceptionTyp1 ab
}

catch (Exceptiontyp2 name2)
{
    .....
    //catch-Block fängt Fehler
    //der Klasse ExceptionTyp2 ab
}
.....
//weitere Catch-Konstrukte
finally
{
    .....
    //optional, wird durchlaufend egal ob ein
    //Fehler aufgetreten ist oder nicht
}

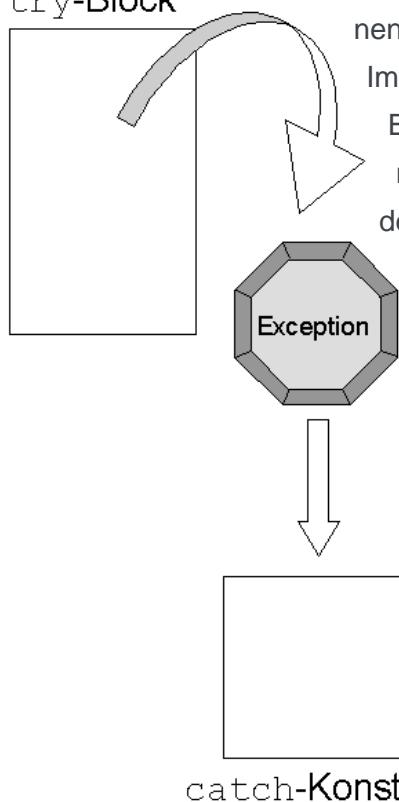
```

Das Auslösen einer Ausnahme bricht die Anweisung ab. Die Kontrolle wird an das Laufzeit-
system der virtuellen Maschine übergeben. Das Laufzeitsystem sucht ei-

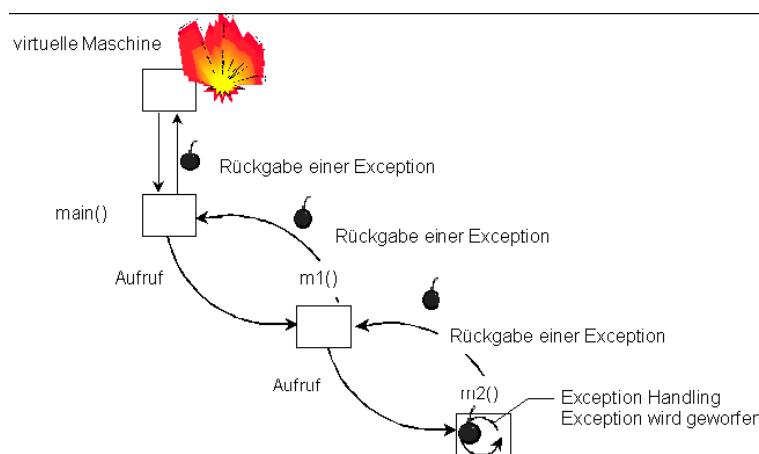
try-Block

nen Handler für die Ausnahme in der Umgebung des try-Blockes.

Im einfachsten Fall steht dieser Handler direkt hinter dem try-
Block. Nach Abarbeitung des Handlers wird das Programm
nach dem Handler fortgesetzt, d.h. es wird nicht an die Stelle
des Auslösens zurückgekehrt. Falls kein Handler da ist, wird
das Programm von der virtuellen Maschine abgefangen.



catch-Konstrukt



```

class Class1
{
    static void Main(string[] args)
    {
        macheWas();
    }

    private static void macheWas()
    {
        tuewas();
    }

    private static void tuewas()
    {
        int c = 0;
        int b = 5 / c;
    }
}

class Class1
{
    static void Main(string[] args)
    {
        try
        {
            macheWas();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }

    private static void macheWas()
    {
        tuewas();
    }

    private static void tuewas()
    {
        int c = 0;
        int b = 5 / c;
    }
}

```

Weiterreichen/Erzeugen von Exceptions per throw

Mit Hilfe der **throw**-Anweisung können Methoden Exceptions an ihre Aufrufer weiterleiten. Dies ist insbesondere dann hilfreich, wenn eine Methode nicht wissen kann, was der Aufrufer im Fehlerfall eigentlich vorhat.

Damit gibt es insgesamt 3 Möglichkeiten des Umgangs mit Exceptions:

- Automatische Weitergabe

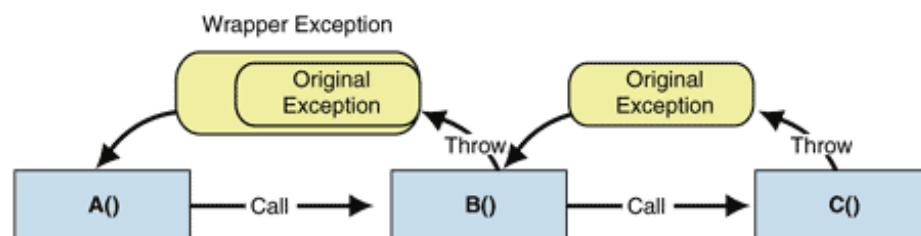
Unter diesem Ansatz ist zu verstehen, dass man die Exception eigentlich ignoriert. Dies bewirkt, dass der Aufrufstack nach oben zurück durchgearbeitet wird, bis ein entsprechender catch-Block gefunden wird.

- Catch und Rethrow

Bei diesem Ansatz wird die Exception gefangen und darauf entsprechend reagiert. Sollte die Exception nicht behandelbar sein, so wird die gleiche Exception per throw weiter nach oben gereicht.

- Catch, wrap and throw the wrapped exception

Je weiter man die Aufrufhierarchie nach oben zurückgeht, desto weniger interessant wird der eigentliche Ausnahmetyp. Durch das Wrappen einer Exception kann eine angemessenerer Exception weiter nach oben gereicht werden, ohne die eigentliche Ursache zu vergessen. Sie wird zur sog. Inner Exception der nach oben weitergereichten Ausnahme



```

class Class1
{
    static void Main(string[] args)
    {
        try
        {
            macheWas();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }

    private static void macheWas()
    {
        tuewas();
    }

    private static void tuewas()
    {
        //throw geht auch hier
        //throw new Exception("exception werfen macht Spass");

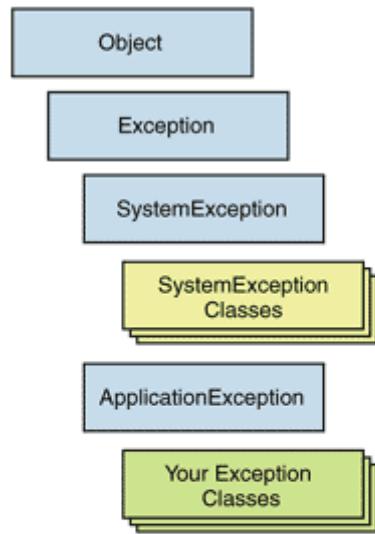
        try
        {
            int c = 0;
            int b = 5 / c;      //DivideByZero-Exception
        }

        catch(Exception ex)
        {
            throw new Exception("Hilfe ein Fehler", ex);
        }
    }
}

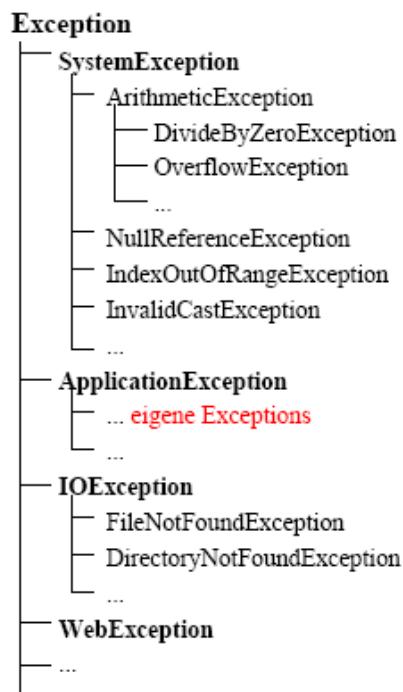
```

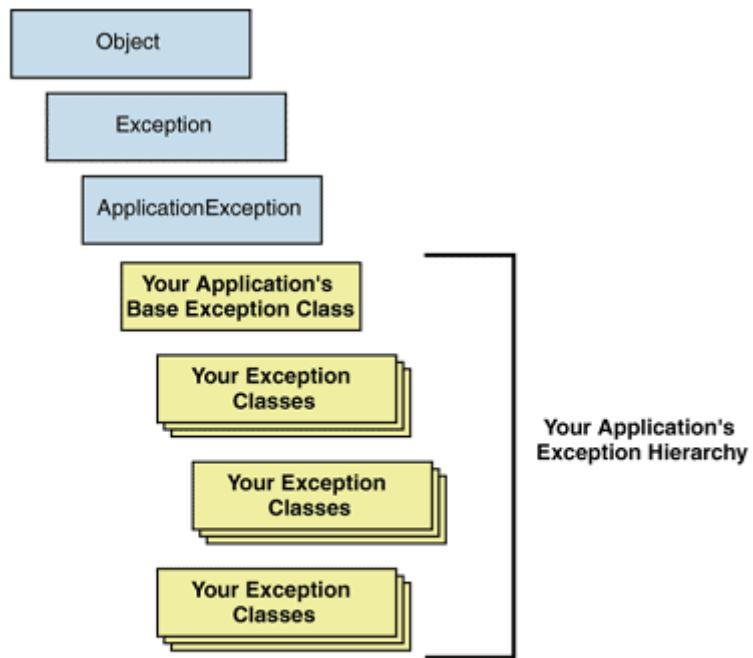
Exception-Hierarchie

Exception sind Klassen und somit kann man verschiedene Typen von Ausnahmen bilden.
.NET kennt folgende Struktur von Ausnahmen:



Exception-Hierarchie (Auszug)





```

using System;
public class EmployeeListNotFoundException: ApplicationException
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}

```

Welche Informationen sollten erfasst werden

Es sollte immer sichergestellt werden, dass für die jeweilige Zielgruppe des Programms alle relevanten Informationen vorhanden sind. Mögliche Zielgruppen sind:

Zielgruppe

Benötigte Information

Anwender

- Hinweis auf Erfolg oder Misserfolg ihrer Handlung
- Verständliche und gut dargestellte Fehlermeldungen
- Hinweise was sie tun sollten, um das Problem zu beheben

Programmierer

- Datum und Zeit des Auftretens der Ausnahme
- Wo trat die Ausnahme auf
- Welche Ausnahme genau trat auf
- Welche Stellen müssen benachrichtigt werden und welche Informationen benötigen sie.
- Welche weiteren Informationen sind mit der Ausnahme verbunden; welchen Zustand hatte das System, als die Ausnahme auftrat.

Tabelle 11.2. Zielgruppen von Fehlermeldungen

Folgende Informationen könnten für die obigen Zielgruppen interessant sein:

Information

Herkunft

Datum und Zeit der Ausnahme

`DateTime.Now`

Name des Computers

`Environment.MachineName`

Quelle der Ausnahme

`Exception.Source`

Typ der Ausnahme

`Type.FullName` obtained from `Object.GetType`

Fehlermeldung der Ausnahme

`Exception.Message`

Stacktrace der Ausnahme

`Exception.StackTrace`: this trace starts at the point the exception is thrown and is populated as it propagates up the call stack.

Call stack

`Environment.StackTrace` – the complete call stack.

Application domain name

`AppDomain.FriendlyName`

Assembly name

`AssemblyName.FullName`, in the `System.Reflection` namespace

Assembly version

Included in the AssemblyName.FullName

Thread ID

AppDomain.GetCurrentThreadId

Thread user

Thread.CurrentPrincipal in the System.Threading namespace

Tabelle 11.3. Informationen für Fehlermeldungen

Aufgaben

- Übersetzen Sie folgende Aussagen sinngemäß in die deutsche Sprache
- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables programmers to create applications that can resolve exceptions, often allowing a program to continue execution as if no problems were encountered.
- Exception handling enables the programmer to remove error-handling code from the main line of the program's execution. This improves program clarity and enhances modifiability.
- When a method detects an error and is unable to handle it, the method throws an exception. There is no guarantee that there will be an exception handler to process that kind of exception. If there is, the exception will be caught and handled.
- A try block consists of keyword try followed by braces ({}) that delimit a block of code in which exceptions could occur.

Immediately following the try block are zero or more catch handlers. Each catch specifies in parentheses an exception parameter representing the exception type the catch can handle.

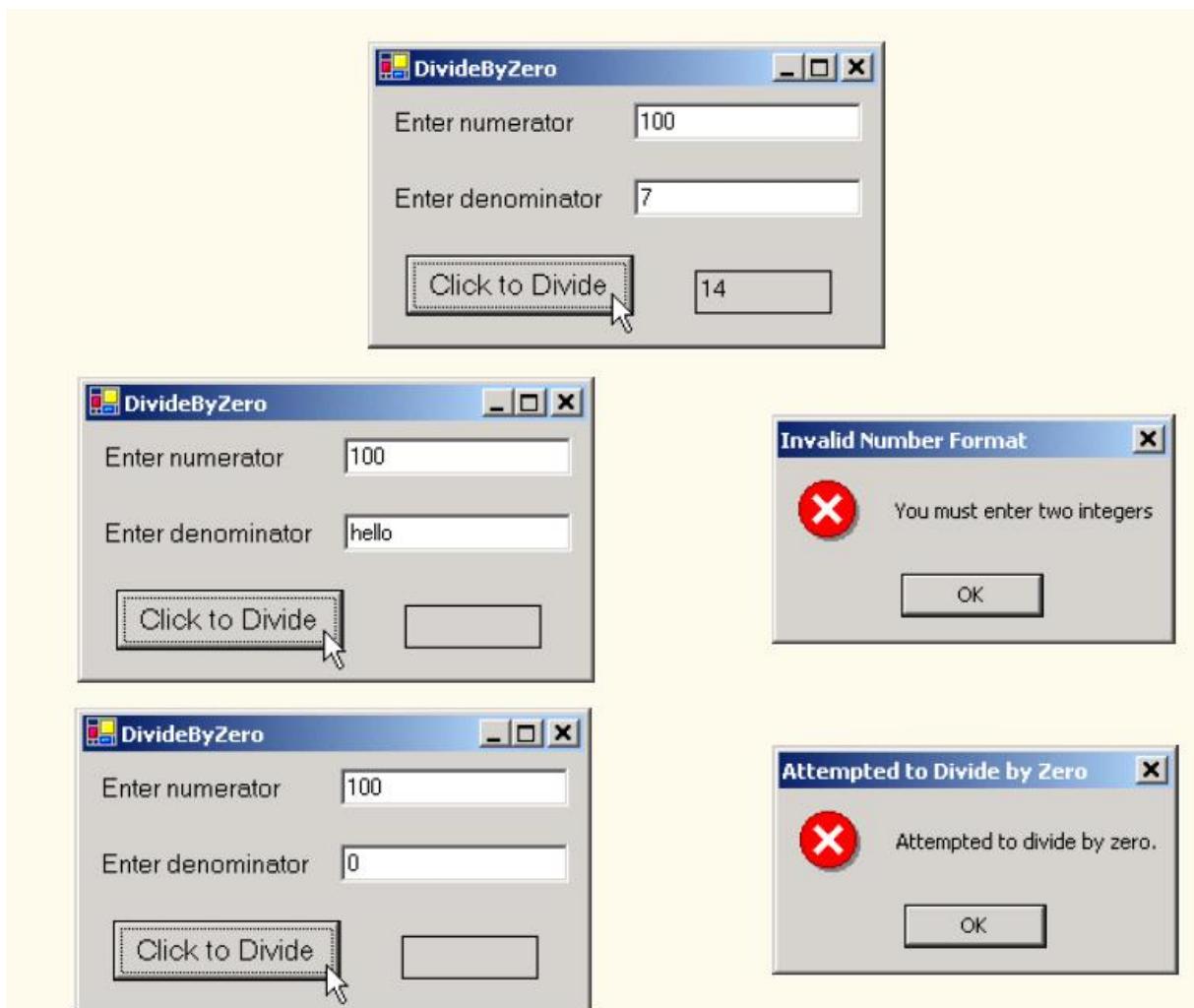
If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with a caught exception object.

There can be one parameterless catch handler that catches all exception types.

After the last catch handler, an optional finally block contains code that always executes, regardless of whether an exception occurs.

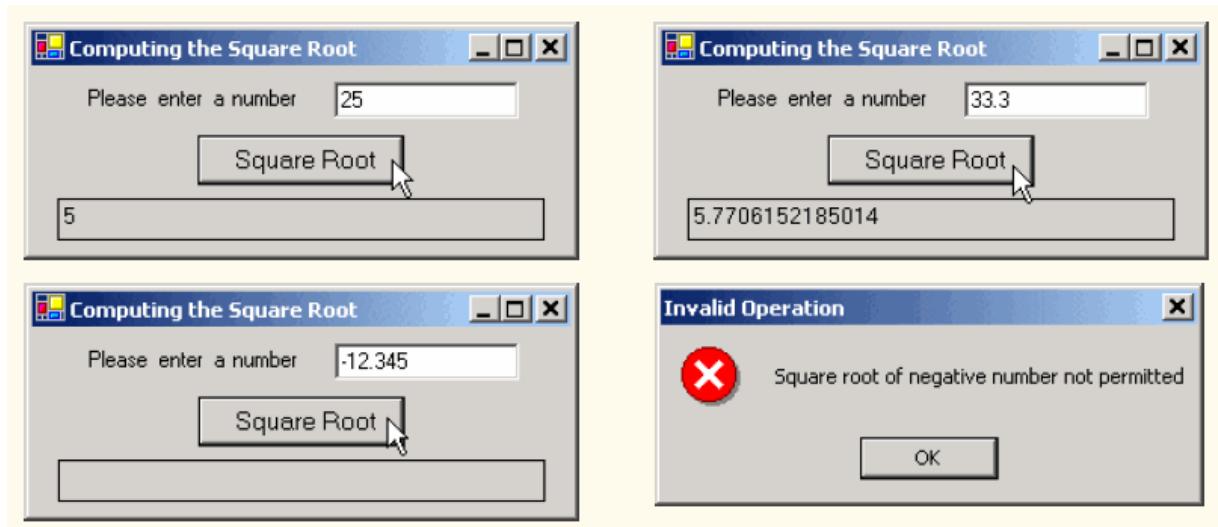


- C# uses the termination model of exception handling. If an exception occurs in a try block, the block expires and program control transfers to the first catch handler following the try block.
- The CLR searches for the first catch handler that can process the type of exception that occurred. The appropriate handler is the first one in which the thrown exception's type matches, or is derived from, the exception type specified by the catch handler's exception parameter.
- Implementieren Sie eine WinForm-Anwendung incl. Fehlerbehandlung gemäß untenstehender Abbildung.



Lösung siehe [hier](#)

- Erstellen Sie eine WinForm-Anwendung, die aus einer übergebenen Zahl die Quadratwurzel zieht. Zum Berechnen der Wurzel benutzen Sie die Funktion Math.Sqrt(double Wert)



Fangen Sie die folgenden Fehler ab:

- FormatException

Ein SystemException. Sie wird von Math.Sqrt geworfen, wenn das Format eines Arguments nicht den Parameterspezifikationen der aufgerufenen Methode entspricht. Wir müssen diesen Umstand berücksichtigen, da wir nicht wissen, welche Daten in die Textbox eingetragen werden.

- NegativeNumberException

Eine eigene Exception, die geworfen wird, wenn der Operand kleiner als die Zahl 0 ist. Erstellen Sie die diese Exception-Klasse selbst, indem Sie sie von der Klasse ApplicationException ableiten. Zur korrekten Vorgehensweise informieren Sie sich bitte in der MSDN.

- try - catch - throw

Es ist eine Konsolenanwendung folgenden Inhalts vorhanden:

```
using System;
using System.IO;
class Class1 {
    static void Main(string[] args) {
        StreamReader myFile = new StreamReader("C:\\\\Text.txt");
        Console.WriteLine(myFile.ReadToEnd());
        Console.ReadLine();
        myFile.Close();
    }
}
```

Das Programm öffnet eine vorhandene Textdatei. Überlegen Sie sich, welche Fehler bei solch einem Vorgang auftreten können und ergänzen Sie den Quellcode um die notwendigen Fehlerbehandlungen.

Lagern Sie den Zugriffscode auf die Textdatei in eine eigene Klasse aus. Das Erkennen möglicher Fehler soll in der neuen Klasse stattfinden; die Reaktion auf einen Fehler in der nutzenden Funktion

- Was ist ein sog. Stacktrace ?
- Erläutern Sie das Konzept der sog. **Inner Exception-Property** der Exception-Klassen.
- Beurteilen Sie folgende beiden Quellcodeausschnitte. Welche Konsequenzen haben die unterschiedlichen Zeilen im catch-Block.

```
class Class1
{
    static void Main(string[] args)
    {
        try
        {
            macheWas();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }

    private static void macheWas()
    {
        tueWas();
    }

    private static void tueWas()
    {
        try
        {
            int c = 0;
            int b = 5 / c;      //DivideByZero-Exception
        }

        catch (Exception ex)
        {
            throw new Exception("Hilfe ein Fehler");
        }
    }
}
```

```

class Class1
{
    static void Main(string[] args)
    {
        try
        {
            macheWas();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }

    private static void macheWas()
    {
        tueWas();
    }

    private static void tueWas()
    {
        try
        {
            int c = 0;
            int b = 5 / c;      //DivideByZero-Exception
        }

        catch(Exception ex)
        {
            throw;
        }
    }
}

```

- Inwieweit unterscheiden sich die Properties/Methoden ToString, Message, StackTrace von Exception-Klassen ?
- Inwieweit unterscheiden sich SystemExceptions von ApplicationExceptions
- Welchen Sinn macht es, sich eigene Exception-Klassen zu schaffen



Weitergehende Aufgaben

Referate

Halten Sie ein Kurzreferat (20 min) zu folgenden Themen

- Die Logging Engine Log4Net (Theorie und praktischer Einsatz)
- Der Logging Block von Microsoft
- Erläutern Sie die Aufgabe eines Profilers bzw Tracers anhand der folgenden Tools:
 - Profile# der Firma Softprodigy (in der Developer Edition kostenlos)

<http://www.softprodigy.net/products>

- <http://www.eqatec.com/tools/tracer> (Freies Trace-Tool)
- <http://www.eqatec.com/tools/profiler> (Freies Profiler-Tool)
- Tracetool von Codeproject

<http://www.codeproject.com/csharp/tracetool.asp>

- Tracetool von Coyote .NET

<http://www.tracingfor.net>

- <http://www.jetbrains.com/profiler/download/index.html>
- Laden Sie sich von <http://www.codeproject.com/dotnet/ExceptionHandling.asp> den entsprechenden Sourcecode herunter und analysieren Sie ihn.
- Lesen Sie den folgenden Artikel: <http://www.codeproject.com/dotnet/exceptionbestpractices.asp#xx1038229xx>
- Karl Seguin ist ein bekannter Entwickler bei Microsoft. Was sagt er zum Thema Exception ?<http://codebetter.com/blogs/karlseguin/archive/2006/04/05/142355.aspx>



Softwareentwicklung

Die folgenden Einheiten beschäftigen sich mit den grundlegenden Informationen, die am Beginn des schulischen Ausbildung notwendig sind.

Lernziele

Fertigstellungsgrad: 50 %

<http://www.sts.tu-harburg.de/~r.f.moeller/lectures/se-ss-04/06-Qualitaet-Metriken-Tests.pdf>

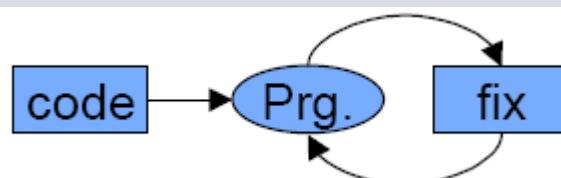
FxCop, NDepend

Unterricht

Vorgehensmodelle

- Wasserfallmodell
- Prototypmodell
- V-Modell
- Extreme Programming, Agile Development
- CMM - Capability and Maturity Model

Code And Fix



Eigenschaften Code And Fix

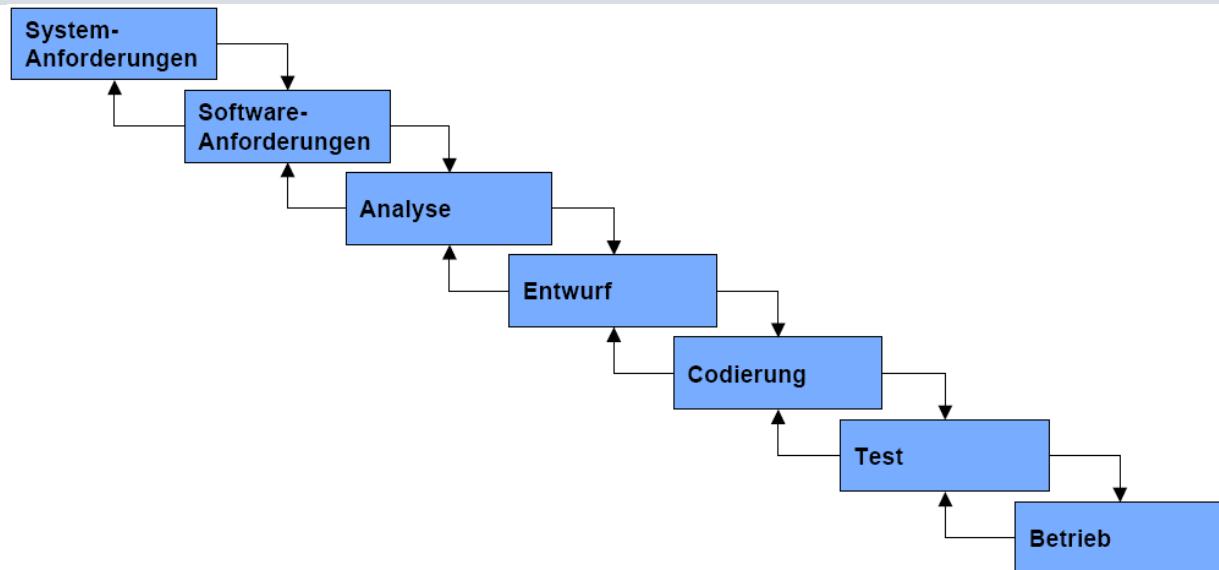
- Schreibe ein Programm
- Finde und behebe die Fehler im Programm

Nachteile

- Fehlerbehebung strukturiert Programm so um, dass weitere Fehlerbehebungen und die Weiterentwicklung immer teurer werden. Eine **Entwurfsphase** wird notwendig.

- Selbst gut entworfene Software wird von den Benutzern oft nicht akzeptiert. Eine **Definitionsphase** vor dem Entwurf wird nötig.
- Fehler sind schwer zu finden, da Tests schlecht vorbereitet und Änderungen unzureichend durchgeführt wurden. Eine separate **Testphase** wird nötig.

Wasserfallmodell



Eigenschaften Wasserfallmodell

- Weiterentwicklung des stufenorientierten Modells
- Sukzessive Stufen der Entwicklung mit Rückkopplung

Charakteristika

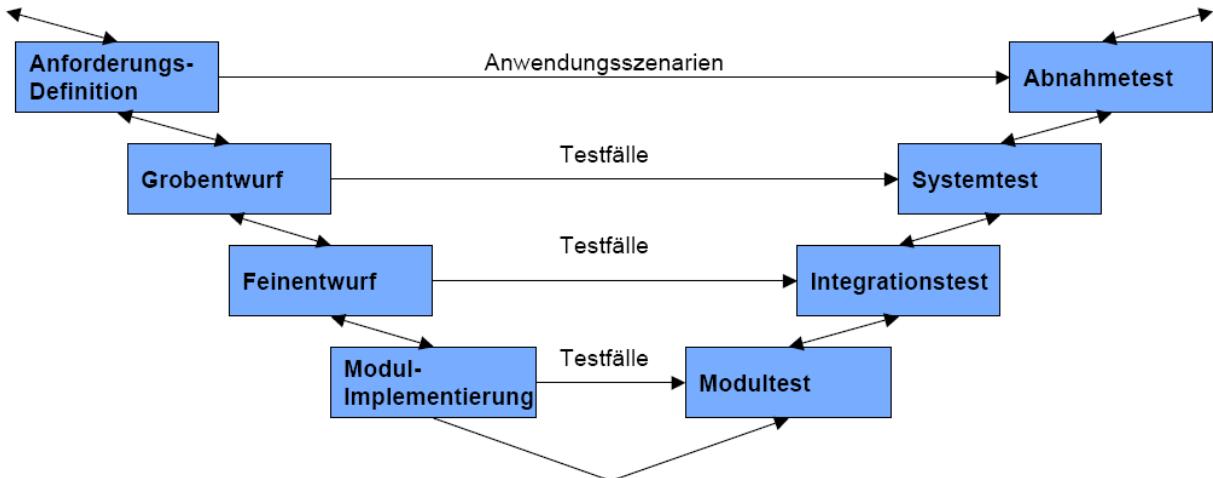
- Aktivitäten sind in der richtigen Reihenfolge und vollen Breite durchzuführen.
- Am Ende jeder Aktivität steht ein Dokument (dokumentgetriebenes Modell)
- Entwicklungsablauf ist sequentiell, vorhergehende Aktivität muß beendet werden, bevor die nächste beginnt.
- Orientiert am Top-down-Vorgehen
- Einfach, verständlich, wenig Managementaufwand
- Benutzerbeteiligung nur in der Definitionsphase

Nachteile

- Notwendige „Kurskorrekturen“ nicht frühzeitig erkennbar
- Sequentialität nicht immer nötig
- Gefahr, daß Dokumente wichtiger als das System werden

- Risikofaktoren werden u.U. zu wenig berücksichtigt

V-Modell



Eigenschaften V-Modell

Erweiterung des Wasserfallmodells, das **Qualitätssicherung** integriert.

Verifikation und **Validation** werden Bestandteile des Modells.

[Anmerkung] Verifikation

“Are we building the product right?”

Überprüfung der Übereinstimmung zwischen Software-Produkt und seiner Spezifikation.



[Anmerkung] Validation

“Are we building the right product?”

Eignung bzw. Wert eines Produkts bezogen auf seinen Einsatzzweck.

Vorteile

- Integrierte, detaillierte Beschreibung von Systemerstellung, Qualitätssicherung, Konfigurationsmanagement und Projektmanagement

- Generisches Vorgehensmodell
- Gut geeignet für große Projekte

Nachteile

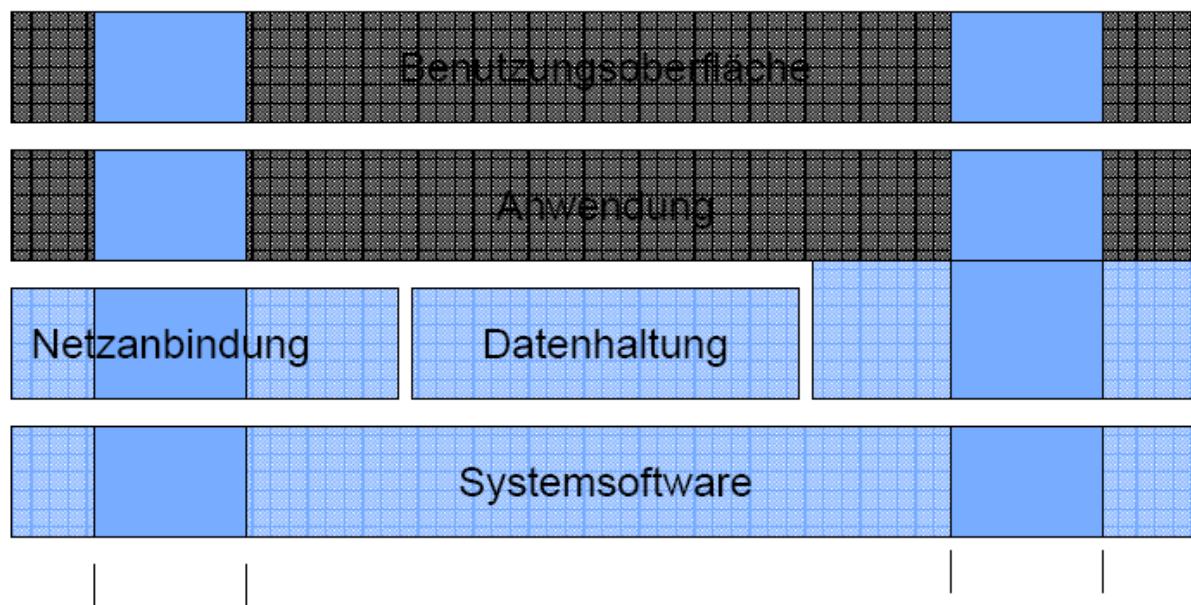
- Unkritische Übernahme der Konzepte, die für eingebettete Systeme

entwickelt wurden, für andere Anwendungstypen - Software-Bürokratie bei kleinen und mittleren Projekten - Ohne CASE-Unterstützung nicht handhabbar

Prototypmodell

- Auftraggeber / Endbenutzer können oft Anforderungen nicht vollständig / explizit formulieren. Dies ist aber in klassischen Definitionsphasen nötig!
- Kooperation zwischen Anwendern und Entwicklern endet mit der Definitionsphase: Entwicklungsabteilungen ziehen sich nach Definitionsphase zurück und präsentieren erst nach Fertigstellung das Ergebnis; wünschenswerte Koordination zum Lernen von den jeweils anderen unterbleibt
- Oft existieren unterschiedliche Lösungswege, die besser experimentell erprobt werden und mit dem Auftraggeber diskutiert werden können.
- Manche Anforderungen lassen sich theoretisch nicht garantieren (z.B. Echtzeitanforderungen). Vor dem Abschluß der Definitionsphase muß also ggf. einiges ausprobiert werden.
- Das Überzeugen des Auftraggebers von der prinzipiellen Durchführbarkeit oder Handhabung einer Idee während der Akquisitionsphase wird nicht unterstützt (Folge für Verantwortungsteilung, Mittelfluss, etc).
-





Ein Software-Prototyp

- ... ist nicht das erste Muster einer großen Serie (beliebig kopierbar, Massenfertigung)
- ... ist keine Simulation, sondern zeigt ausgewählte Eigenschaften des Zielprodukts im praktischen Einsatz (vgl. z.B. Windkanal oder Architekturmodell)
- ... dient zum Klären von relevanten Anforderungen oder Entwicklungsproblemen.
- ... dient als Diskussionsbasis für Entscheidungen.
- ... dient zu experimentellen Zwecken und Sammeln von praktischen Erfahrungen.

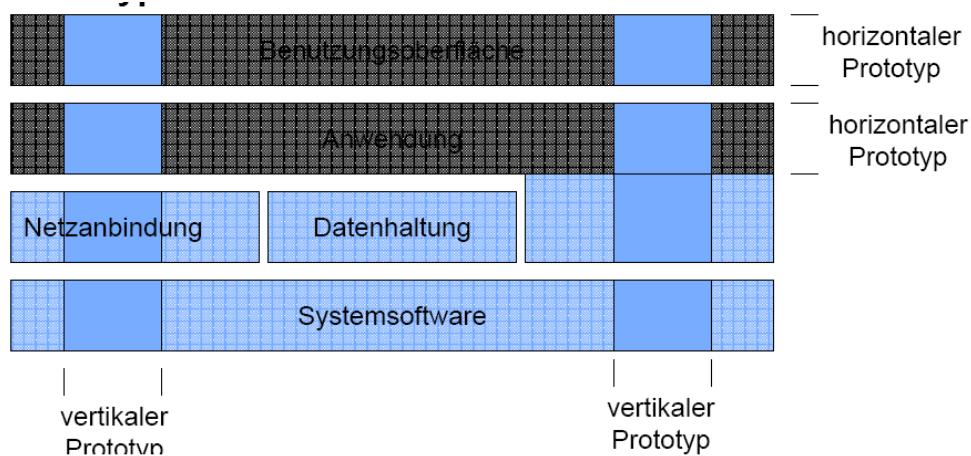
Arten von Prototypen

- Demonstrationsprototyp
Dient zur Auftragsakquisition; verschafft Eindruck, wie das Produkt aussehen kann. Wichtig: Wird später weggeworfen!
- Prototyp im engeren Sinne:
Wird parallel zur Modellierung des Anwendungsbereiches erstellt, um Aspekte der Benutzungsschnittstelle oder Teile der Funktionalität zu veranschaulichen. Dient zur Analyse. (Exploratives Prototyping)
- Labormuster:
Dient zur Beantwortung konstruktionsbezogener Fragen und Alternativen.

- Pilotsystem:

Dient nicht nur zur experimentelle Erprobung oder Veranschaulichung, sondern ist schon Kern des Produkts. Unterscheidung zwischen Prototyp und Produkt verschwindet später. Die Weiterentwicklung erfolgt in Zyklen unter Beteiligung der Benutzer. Es ist ein wesentlich sorgfältigerer Entwurf nötig, da der Prototyp später weiterbenutzt wird! Benutzerdokumentation wird ebenfalls nötig. (Evolutionäres Prototyping)

- Horizontale und vertikale Prototypen



Bewertung

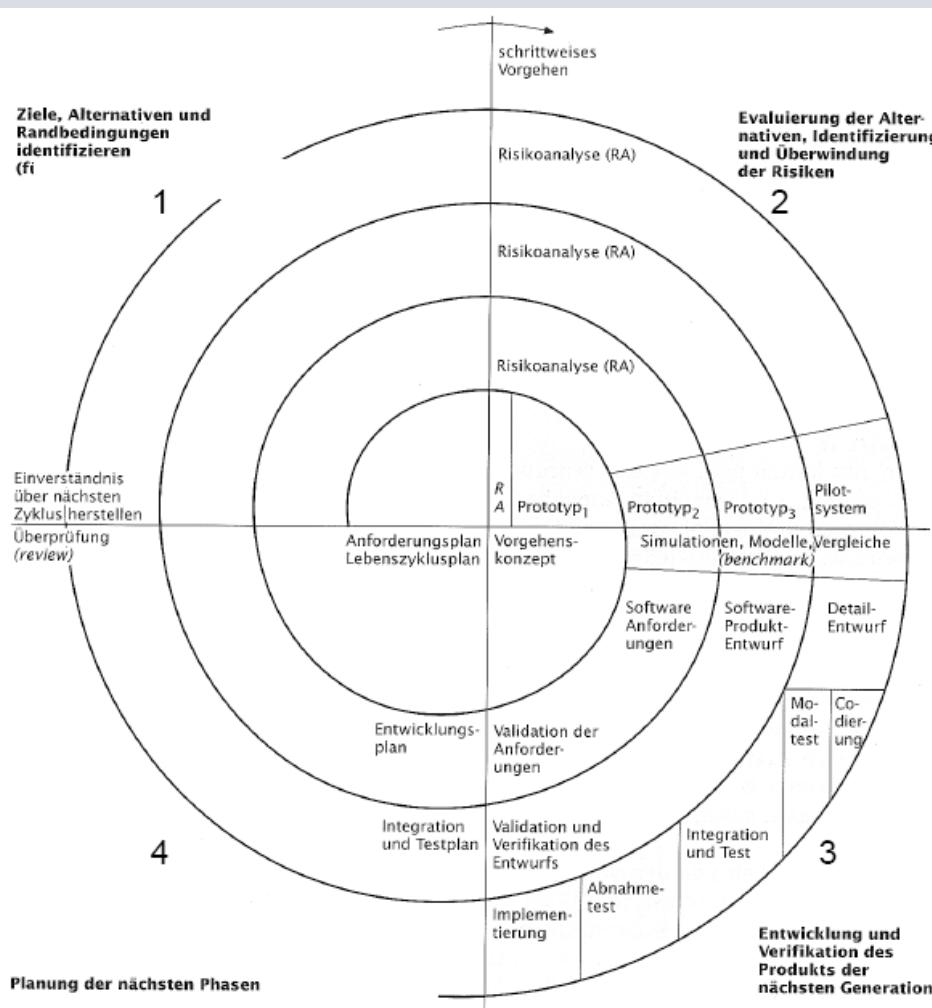
Vorteile

- Reduktion des Entwicklungsrisikos durch frühzeitige/stärkere Rückkopplung.
- Sinnvoll in andere Prozeßmodelle integrierbar.
- Prototypen sind durch geeignete Werkzeuge schnell erstellbar. ("Rapid Prototyping")

Nachteile

- Höherer Entwicklungsaufwand.
- Gefahr, daß ein „Wegwerf“-Prototyp nicht weggeworfen wird.
- Prototypen werden oft als Ersatz für Dokumentation angesehen.

Spiralmodell



- Das Spiralmodell ist eigentlich ein Modell höherer Ordnung
- Für jedes (Teil-)Produkt sind zyklisch vier Schritte zu durchlaufen:
 - Schritt 1:
 - Identifizierung der Ziele des Teilprodukts (Leistung, Funktionalität, Anpaßbarkeit, ...)
 - Alternative Möglichkeiten zur Realisierung des Teilprodukts finden.
 - Randbedingungen bei verschiedenen Alternativen finden
 - Schritt 2:
 - Evaluierung der Alternativen unter Berücksichtigung aller Alternativen
 - Identifizieren und ggf. Überwinden von Risiken (durch Prototypen, Simulation, ...)
 - Schritt 3:
 - Abhängig vom Risiko wird ein Prozeßmodell festgelegt (oder eine Kombination).
 - Anwendung des Modells
 - Schritt 4:
 - Planung des nächsten Zyklus, Überprüfung der nächsten 3 Schritte im nächsten Zyklus, Einverständnis mit Beteiligten sichern.

Bewertung

Eigenschaften

- Risikogetriebenes Modell, da Hauptziel die Minimierung des Risikos ist.
- Ziel: Beginne im Kleinen, halte die Spirale so eng wie möglich und erreiche das Ziel mit minimalen Kosten.

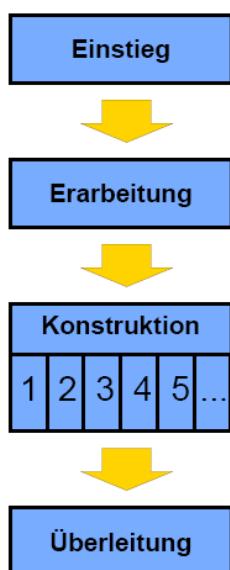
Vorteile

- Periodische Überprüfung und ggf. Neufestlegung des Prozeßmodells
- Prozeßmodell ist nicht für die gesamte Dauer des Projekts festgelegt.
- Flexibel, leichtere Umsteuerung
- Erleichtert Wiederverwendung von Software durch Betrachtung von Alternativen.

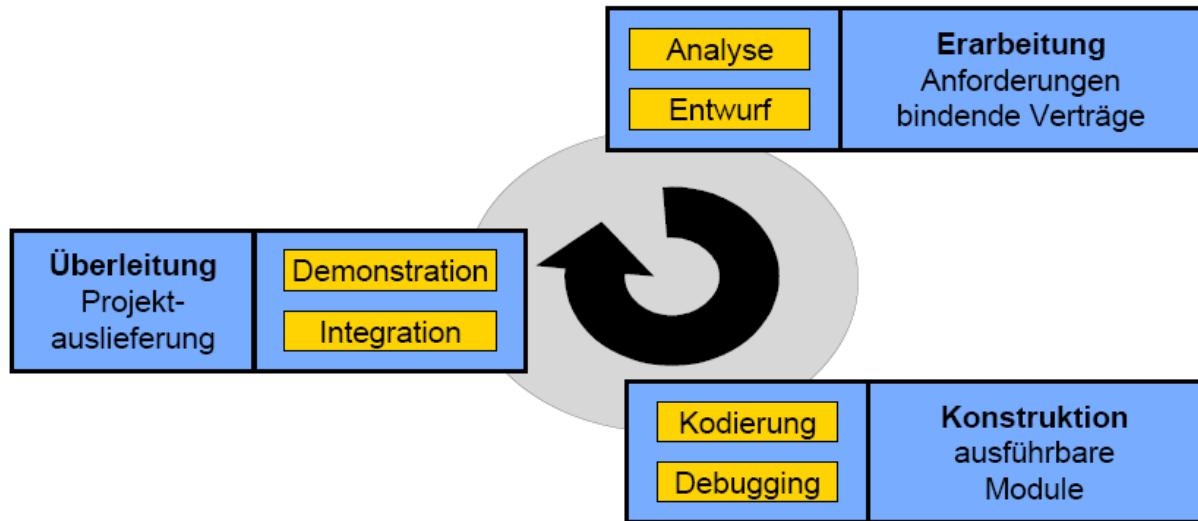
Nachteile

- Hoher Managementaufwand
- Für kleine und mittlere Projekte weniger gut geeignet.
- Wissen über Identifizierung und Management von Risiken ist noch nicht sehr verbreitet.

Unified Process



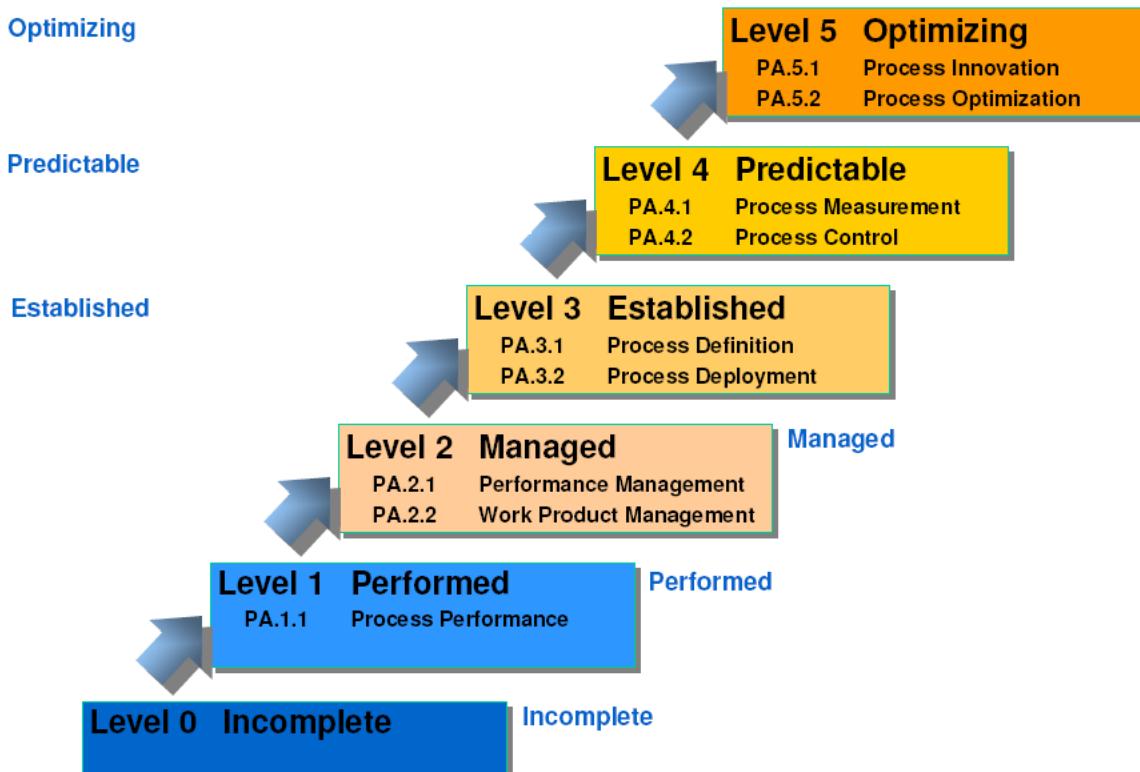
- Der Einstieg etabliert das Geschäftsziel und legt den Umfang des Projektes fest.
- In der Erarbeitungsphase werden detaillierte Anforderungen gesammelt, Analyse betrieben und Entwurf grundsätzliche Architekturentscheidungen getroffen sowie der Plan für die Konstruktion gemacht. (use case diagrams)
- Die Konstruktion ist ein iterativer und inkrementeller Prozeß. Jede Iteration dieser Phase baut Software- Prototypen mit Produktqualität, die getestet werden und einen Teil der Anforderungen des Projekts umsetzen. (use-case driven)
- Die Überleitungsphase enthält den Beta-Test, Leistungssteigerung und Benutzer-Training.



XP (Extreme Programming)

Hier Gedanken zu Extreme Programming

CMM - Capability Maturity Model



In welcher Firma möchten Sie lieber arbeiten ?

Eigenschaft

Unreife Organisation

Reife Organisation

Verantwortlichkeiten und Rollen

Weitgehend ungeklärt. Jeder Mitarbeiter sucht sich seine Rolle. Dies führt zu ungeklärten Verantwortlichkeiten.

Definierte Verantwortlichkeiten. Es werden Ziele gesetzt und Ergebnisse beurteilt. Keine große Überlappung von Aufgaben

Umgang mit Veränderungen

Jeder hat seinen eigenen Arbeitsstil und führt Veränderungen ein, wie ihm gerade passt.

Die Mitarbeiter folgen einem geplanten und definierten Prozess. Sie teilen das Know-How über den Prozess und lernen aus Erfahrungen.

Reaktion auf Probleme

Chaos ist die Regel. Die Mitarbeiter sind damit beschäftigt, das gerade aktuelle Problem zu lösen. Jeder hält sich für einen Helden. Häufige Überstunden und Nacharbeit.

Die Probleme werden auf der Grundlage eines fundierten Wissens analysiert; professionelle Vorgehensweise.

Zuverlässigkeit

Häufig verspätete Auslieferung der Software; Schätzungen zu Kosten und Terminen sind unzuverlässig.

Schätzungen sind weitgehend zuverlässig, der Umfang des Projektes wird kontrolliert. Ziele werden in der Regel erreicht.

Tabelle 13.1. In welcher Firma arbeiten Sie ?

In den USA war man sich in den 80/90-Jahren zunehmend darüber im Klaren, dass bei modernen Waffensystemen Software eine kritische Komponente darstellt. Versagt in solchen Systemen die Software, dann ist die Mission gefährdet. Man zog daraus den Schluss, dass die Qualität der Software, wie sie von großen Systemherstellern produziert wird, um Klassen besser werden muss. Dies ist der Zweck von CMM.

Grundlage des CMM ist ein fünfstufiges Modell, bei dem jeder Ebene eine gewisse Reife (Maturity) zugeordnet wird.

Die Ebenen des CMM

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimizing

Softwareergonomie

Aufgaben

