Inhaltsübersicht

Kapitel 1: .NET-Architecture

Common Language Runtime (CLR)

- in der CLR ausgeführter Code wird als *managed code* bezeichnet
- die Kompilierung erfolgt in zwei Schritten: source code → Intermediate Language → native code
- Vorteile von managed code:
  - Plattformunabhängigkeit: ähnlich wie Byte Code in Java
  - Geschwindigkeitsverbesserung: der benötigte Teil der IL wird nur einmal beim Aufruf (just-in-time) kompiliert und für den jeweils verwendeten Prozessor optimiert
  - Sprachinteroperabilität: C#, VB 2005, Visual C++ 2005, Visual J# 2005, JScript.NET

Intermediate Language (IL)

- Unterstützung von Objektorientierung und Interfaces
  - strikte Verfolgung des objektorientierten Paradigmas mit einfacher Vererbung
  - Interfaces bilden Verträge, Klassen müssen die jeweiligen Methoden und Properties implementieren
  - in verschiedenen Sprachen geschriebene Klassen können direkt miteinander kommunizieren und beliebige Vererbungsstrukturen abbilden
- Strikte Trennung zwischen Werte- und Referenztypen
  - Wertetypen werden im *stack* gespeichert
  - Referenztypen werden im *managed heap* gespeichert
- Starke Datentypisierung:
  - jede Variable hat einen eindeutigen Datentyp
  - *Common Type System* (CTS): definiert die standardmäßig verfügbaren Datentypen der IL
  - *Common Language Specification* (CLS): spezifiziert ein Minimum an Standards, welche jede CLS-kompatible Sprache unterstützen muss, um Sprachneutralität herzustellen
  - CLS wirkt sich nur auf public- und protected-Member der Klassen aus
  - der *Garbage Collector* räumt nicht mehr benötigten Speicherplatz im managed heap auf
  - *code-based-security* anstelle von *role-based-security*
  - jede Anwendung läuft in seinem eigenen Prozess (4 GB unter 32 Bit-Systemen), welcher aus verschiedenen *Application Domains* bestehen kann
  - geprüfter Code, welcher auf keine Daten außerhalb der Application Domain zugreifen kann, wird *memory type safe* genannt
- Fehlerbehandlung über Ausnahmen
  - try{}, catch{}, finally{}
- Verwendung von Attributen
  - Unterstützung von benutzerspefizischen Attributen in den Metadaten
  - unterstützen *Reflection*

Assemblies

- ist eine selbstbeschreibende, logische Einheit, welche kompilierten Code des .NET-Frameworks enthält und aus mehreren Dateien bestehen kann
- die Assembly-Struktur für ausführbaren und library code ist identisch, nur dass der ausführbare Code einen Hauptprogrammeinstiegspunkt besitzt

- jede Assembly enthält Metadaten über sich selbst (*manifest*) und über die enthaltenen Datentypen und Methoden, die Reflection unterstützen
- *Private Assemblies*
  - werden meist mit Software ausgeliefert und können nur von dieser Anwendung verwendet werden
  - Anwendungen dürfen nur private Assemblies laden, die im gleichen oder einem Unterverzeichnis der Anwendung liegen
  - haben den gleichen Namen wie ihre Hauptdatei
- *Shared Assemblies*
  - können von jeder Anwendung genutzt werden und unterliegen deshalb erhöhten Risiken
  - Namenskollisionen zwischen verschiedenen Shared Assemblies müssen vermieden werden
  - Gefahr der Überschreibung von einer anderen Version der gleichen Assembly
  - müssen im *global assembly cache* (GAC) installiert werden
  - besitzen einen *strong name*, der auf der private key-Kryptografie beruht

.NET Framework Classes

- .NET Basisklassen sind eine große Ansammlung von managed code Klassen, die nahezu alle Aufgaben übernehmen, die vorher durch die Windows API bereitgestellt wurden
- kombinieren einfache Bedienbarkeit mit der Vielfalt der Windows API
- umfassen CTS-Datentypen, Windows GUI Elemente, Web Forms, Datenzugriff, Dateiverwaltungssystem, Netzwerk- und Webzugriff, Zugriff auf OS-Variablen, COM Interoperabiltität

Namespaces

- Gruppierung von Datentypen zur Vermeidung von Namenskonflikten zwischen Klassen
- Unterstützung von Namespace-Hierarchien
- alle Datentypen müssen einem Namespace zugeordnet sein
- Namespaces sind unabhängig von Assemblys: eine Assembly kann mehrere Namespaces verwenden, ein Namespace kann über mehrere Assemblys hinweg existieren
- es können Aliase für Namespaces definiert werden

```
using alias = NamespaceName;
```

Creating .NET Applications Using C#

- ASP.NET
- Web Forms / Web Server Controls
- XML Web Services
- Windows Forms / Windows Controls
- Windows Presentation Foundation (WPF)
- Windows Services
- Windows Communication Foundation (WCF)

Kapitel 2: C# Basics

Variables

- Beispielprogramm

```
using System;
namespace FPK
{
```

```
    class MyFirstClass
    {
            static void Main()
            {
                    Console.WriteLine("Hello World!");
                    return;
            }
    }
}
```

- Variablen, die Felder in Klassen oder Structs darstellen, werden bei Nicht-Initialisierung auf 0/Null gesetzt; Variablen in lokalen Methoden müssen explizit initialisiert werden
- Variablen können in einem (auch verschachtelten) Scope nur einmal deklariert werden, Ausnahme: Felder einer Klasse und lokale Methodenvariablen
- Konstanten sind implizit *static* und dürfen nicht noch zusätzlich als static gekennzeichnet werden
- sollen eigene Datentypen als Wertetyp behandelt werden, müssen sie als *struct* definiert werden
- vordefinierte Wertetypen: sbyte/byte, short/ushort, int/uint, long/ulong, float, double, decimal, bool, char
- vordefinierte Referenztypen: object, string
- wird ein String geändert, so wird jedes Mal ein neues Objekt erzeugt
- um in einem String Sonderzeichen ohne Umwege nutzen zu können, kann ein @ vorangestellt werden: string filepath = @"C:\Ordner\Datei.txt"
- gibt es mehr als eine Main-Methode, muss dem Compiler mitgeteilt werden, welche er zum Start aufrufen soll

Flow Control

- bei Switch-Anweisungen werden Code-Blöcke nicht mit geschweiften Klammern ausgedrückt, sondern das Ende muss mit *break* markiert werden, Ausnahme: leerer Block
- die Fallunterscheidung benötigt einen konstanten Ausdruck, Variablen sind nicht erlaubt
- innerhalb der Switch-Anweisungen sind *goto*-Anweisungen möglich

```
switch(country)
{
    case „America":
           CallAmericanOnlyMethod();
           goto case „Britain;
    case „France":
           language = „French";
           break;
    case „Britain":
           language = „English";
           break;
}
```

- die Reihenfolge der Anweisungen spielt keine Rolle, die default-Anweisung kann auch als erstes stehen, deshalb darf es keine Konstanten mit dem gleichen Wert geben
- mit Hilfe der *foreach*-Schleife ist es möglich, durch alle Elemente von *Collections* zu iterieren

```
foreach (int temp in arrayOfInts)
{
    Console.WriteLine(temp);
}
```

- der Wert eines Elements in einer foreach-Schleife kann nicht verändert werden

Jump Statements

- mit *goto* kann zu einer bestimmten Stelle eines Code-Blocks gesprungen werden

```
goto Label1;
Console.WriteLine(„This won't be executed");
```

```
Label 1:
    Console.WriteLine("Continuing execution from here");
```

- mit goto kann nicht in einen Codeblock wie eine for-Schleife gesprungen werden, nicht aus einer Klasse raus und auch nicht aus einem *finally*-Block
- wird eigentlich nur für die sehr rigide switch-Anweisung benötigt
- mit *break* können switch-Anweisungen und Schleifen beendet werden
- mit *continue* wird innerhalb von Schleifen zur nächsten Iteration gesprungen
- mit *return* wird eine Klassenmethode (ggf. mit Rückgabewert) beendet

Enumerations

- benutzerdefinierte Integertypen, die die Wartung des Codes unterstützen (typsichere Zuweisungen) und den Code klarer und einfacher zu programmieren machen

```
public enum TimeOfDay
{
    Morning = 0;
    Afternoon = 1;
    Evening = 2;
}
static void WriteGreeting(TimeOfDay timeOfDay)
{
    switch(timeOfDay)
    {
            case TimeOfDay.Morning:
                Console.WriteLine("GoodMorning!");
                break;
            case TimeOfDay.Afternoon:
                …
    }
}
```

- Enumerations werden intern wie Structs behandelt, so dass einige nützliche Methoden mit ihnen aufgerufen werden können (z.B. Umwandlung eines Strings in einen Enumwert)

Arrays

- alle Arrays sind Referenztypen

```
int[] integers = new int[32];
int[] copy = integers;    //refers to the same array
```

More on Compiling C# Files

- beim Kompilieren kann angegeben werden, ob eine Konsolen- oder Windowsapplikation, eine Klassenbibliothek mit Manifest (.dll) oder eine Komponente ohne Manifest (kann in ein anderes Manifest kompiliert werden) erzeugt werden soll

Using Comments

- XML-Kommentare beginnen mit: ///
- der Compiler kann daraus eine XML-Dokumentation erstellen, wobei automatisch Einträge für die Assembly und die Member einer Klasse erzeugt werden

The C# Preprocessor Directives

- werden nie in Code übersetzt, aber beeinflussen das Verhalten des Compilers
- sie beginnen alle mit # und enden ohne Semikolon
- *#define* und *#undef* definieren ein Symbol, welches z.B. mit *#if* interpretiert werden kann
- mit *#if*, *#elif*, *#else* und *#endif* können Codeteile abhängig vom Inhalt eines Symbols kompiliert werden

```
#if DEBUG
    Console.WriteLine(„Debug-Mode on.“);
#endif
```

- mit *#warning* und *#error* können Warnungen und Fehler für den Entwickler erzeugt werden, sobald der Compiler die jeweilige Stelle erreicht hat
- mit *#region* und *#endregion* können Codeteile in einem Editor zusammengefasst und ausgeblendet werden

C# Programming Guidelines

- soll ein reserviertes Schlüsselwort als Bezeichner verwendet werden, muss ein @ vorangestellt werden: @abstract
- alle Namen von Namespaces, Klassen und Membern folgen dem *Pascal casing* (der erste Buchstabe jedes Wortes wird groß geschrieben)
- für private Member-Felder, Methoden-Parameter und zum Unterscheiden von Elementen, die sonst den gleichen Namen hätten (z.B. Properties) wird *camel casing* verwendet (das erste Wort klein, dann jedes nachfolgende mit einem Großbuchstaben beginnen)
- Wann werden Properties statt Methoden verwendet? Wenn sich der Inhalt nicht (ständig) ändert, wenn das Setzen von Properties in beliebiger Reihenfolge möglich ist, wenn der Inhalt des Properties clientseitig ausgelesen werden kann, wenn das Lesen der Werte schnell von statten geht und das Setzen von Werten keine Seiteneffekte hervorruft
- Felder sollten immer als *private* deklariert werden

Kapitel 3: Objects & Types

Class Members

- Klassen sind Referenztypen, die im heap gespeichert werden,
- es wird zwischen *Data Member* und *Function Member* unterschieden
- Data Member umfasst Felder (alle der Klasse zugehörigen Variablen), Konstanten und Events
- Function Member umfassen Methoden, Properties, Konstruktoren, Operatoren und Indexer
- Parameter können als Wert (standardmäßig) oder als Referenz übergeben werden
- bei Referenztypen wird die Speicheradresse kopiert (bei call by value)
- ref-Parameter: legt fest, dass der Parameter als Referenz übergeben wird; muss sowohl beim Aufruf als auch bei der Parameterdefinition angegeben werden
- out-Parameter: analog dem ref-Parameter, aber auch für nicht-initialisierte Variablen
- Properties besitzen eine get- und eine set-Methode zum Auslesen von Werten

```
private string foreName;
public string Forename
{
    get
    {
            return foreName;
    }
    set
    {
            foreName = value;
    }
}
```

- für ein read-only-Property wird die set-Methode weggelassen
- get- und set-Methoden können unterschiedliche Zugriffsmodifizierer besitzen, jedoch muss mindestens eine dem gleichen Level wie das Property angehören

- wird kein Konstruktor für eine Klasse erstellt, so wird automatisch ein Basis-Konstruktor angelegt, der alle Memberfelder ausnullt
- soll die Klasse nur als Container für statische Member dienen, so kann der Konstruktur als private deklariert werden
- es kann ein statischer, parameterloser Konstruktur ohne Zugriffsmodifizierer erstellt werden, der vor dem ersten Verwenden der Klasse ausgeführt wird (z.B. um Daten von einer externen Quelle zu besorgen)
- ein Konstruktur kann einen anderen Konstruktor der selbern Klasse (this) oder der direkt übergeordneten Basisklasse (base) aufrufen

```
public Car(string description) : this(model,4)
```

- ein read-only-Feld ist flexibler als eine konstante Variable, da z.B. der Wert zur Laufzeit durch Berechnungen ermittelt werden kann
- einem read-only-Feld kann nur in einem Konstruktur ein Wert zugeordnet werden
- durch das *partial*-Schlüsselwort kann eine Klasse, ein Struct oder ein Interfaces sich auf mehrere Dateien verteilen
- *static*-Klassen haben den gleichen Effekt wie eine Klasse mit nur einem privaten statischen Konstruktor

Structs

- Structs werden als Wertetypen auf dem Stack gespeichert
- Structs unterstützen keine Vererbung (mit Ausnahme von Interfaces)
- es wird automatisch ein Standardkonstruktor ohne Parameter bereitgestellt, der nicht überschrieben werden darf
- beim Erzeugen eines Structs muss kein *new*-Operator aufgerufen werden
- wird er aufgerufen, so wird der Standardkonstruktur aufgerufen und alle Felder werden initialisiert (deshalb ist es auch nicht möglich, Felder mit Initialwerten zu belegen)

Kapitel 4: Inheritance

Implementation Inheritance

- durch das Deklarieren von Funktionen oder Properties als *virtual* wird das Überschreiben in abgeleiteten Klassen durch *override* ermöglicht
- wird eine Methode mit der gleichen Signatur in Ober- und Unterklasse ohne die Schlüsselwörter virtual und override deklariert, so versteckt die Methode der Unterklasse die andere
- der Compiler wird eine Warnung ausgeben, wenn beim Verdecken nicht das *new*-Schlüsselwort verwendet wird
- bei new handelt es sich um *Early Binding*: wenn die Oberklasse zur Unterklasse gecastet wird, wird trotzdem die Methode der Oberklasse ausgeführt
- bei override handelt es sich um *Late Binding*: wenn die Oberklasse zur Unterklasse gecastet wird, wird zur Laufzeit der Typ der Instanz ermittelt und somit die Methode der Unterklasse aufgerufen
- abstrakte Funktionen werden nur durch *abstract* deklariert, sie sind implizit *virtual*
- Klassen und Methoden können als *sealed* deklariert werden, so dass von ihnen nicht vererbt werden kann bzw. sie nicht überschrieben werden können
- werden bei vererbten Klassen keine Konstruktoren hinzugefügt, so wird beim Aufruf zunächst der Standard-Konstruktor der jeweiligen Oberklassen ausgeführt, angefangen mit *System.Object*

- wird im Konstruktor der abgeleiteten Klasse keine Referenz auf einen Konstruktor der Oberklasse angegeben, wird automatisch der Basiskonstruktor der Oberklasse aufgerufen
- wenn in der Oberklasse ein Konstruktor mit Parametern deklariert wird, so erzeugt das System keinen Basiskonstruktor mehr → beim Erzeugen eines Objektes der Unterklasse (wessen Konstruktor keine Referenz aufweist) gibt es deswegen einen Fehler

Modifiers

- Sichtbarkeitmodifzierer: public, protected, internal, private, protected internal
- Andere Modifizierer: new, static, virtual, abstract, override, sealed, extern

Interfaces

- Interfaces besitzen keinen Konstruktor, keine Felder und keine Operatorüberladungen

Kapitel 5: Arrays

Arrays

- um ein zweidimensionales, rechteckiges Array anzulegen, wird folgender Code benötigt

```
int[,] twodim = new int[3,3];
```

- ein Jagged Array ist flexibler, da jede Dimension eine variable Größe besitzen kann

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] {1, 2};
jagged[1] = new int[4] {3, 4, 2, 7};
jagged[2] = new int[3] {4, 1, 3};
```

- da Arrays Referenztypen sind, wird bei einer Zuweisung einer Array-Variable zu einer anderen lediglich die Speicheradresse übergeben
- um ein Array zu kopieren, muss die Methode *Clone()* aufgerufen werden (die Referenzen innerhalb des Arrays verweisen aber immer noch auf die gleichen Objekte)
- zum Sortieren von Elementen müssen diese das Interface *IComparable* (welches als einzige Methode CompareTo besitzt) implementieren
- hat man keinen Zugriff auf die zu sortierenden Klassen oder möchte das Sortieren anderweitig durchführen, kann das Interface *IComparer* implementiert werden, welches unabhängig von den zu vergleichendenen Klassen ist
- die Klasse Array implementiert die Interfaces IEnumerable, ICollection und IList

Enumerations

- zum Durchlaufen einer foreach-Schleife wird eine Methode *GetEnumerator()* benötigt, die einen Enumerator zurückliefert (wird z.B. durch das Interface IEnumerable implementiert)
- der Enumerator implementiert das Interface IEnumerator, welches die Methoden MoveNext() und Reset() sowie das Property Current vorsieht
- mit Hilfe der yield-Anweisung wird die Erstellung eines Enumerators erleichtert

```
public IEnumerator GetEnumerator()
{
    for (int i=0; i<list.Length; i++)
    {
            yield return list[i];
    }
}
```

Kapitel 6: Operators & Casts

Operators

- mit dem *checked*-Schlüsselwort wird die Überlaufprüfung für arithmetische Operationen und Konvertierungen mit ganzzahligen Typen explizit aktiviert, mit *unchecked* deaktiviert (standardmäßig aktiv)
- *is* überprüft, ob ein Objekt mit einem bestimmten Typ kompatibel ist, d.h. ob eine Umwandlung ohne Ausnahme erfolgreich verlaufen würde
- *as* erweitert den is-Operator um eine Konvertierung, falls diese kompatibel sind; ansonsten wird *null* zurückgeliefert
- Nullable Types (z.B. *int?*)sind Datentypen, die alle Werte des zugrunde liegenden Typen annehmen können - und zusätzlich den Wert null.
- der Null-Coalescing-Operator *??* prüft, ob der erste Operand ungleich null ist, ist dies der Fall, hat der ganze Ausdruck den Wert des ersten Operanden, ansonsten besitzt er den Wert des zweiten Operanden

```
string fileName = tempFileName ?? „Untitled“;
```

Type Safety

- Implizite Umwandlungen können nur von einem kleinen Integerdatentyp in einen größeren oder gleich großen erfolgen
- Nullable Types können nicht implizit in Nicht-Nullable-Types umgewandelt werden
- Umwandlungen, die nicht implizit durch den Compiler erfolgen können, kann der Benutzer explizit machen
- zur sicheren expliziten Umwandlung kann der *checked*-Operator hinzugefügt werden

```
int i = checked((int) val);
```

- *Parse()* dient der Umwandlung von String in einen numerischen oder boolschen Wert
- *Boxing* kann implizit erfolgen, *Unboxing* muss immer explizit angegeben werden

Comparing Objects for Equality

- für Referenztypen gibt es vier Möglichkeiten, Objekte auf Gleichheit zu prüfen
  - statische *ReferenceEquals()*-Methode: prüft, ob zwei Referenzen auf den gleichen Speicherbereich verweisen
  - virtuelle *Equals()*-Methode: arbeitet wie ReferenceEquals, kann aber überschrieben werden
  - statische *Equals()*-Methode: nimmt zwei Parameter auf und ruft intern die virtuelle Equals-Methode auf
  - ==-Operator: bester Kompromiss zwischen striktem Referenzvergleich und strikten Wertevergleich, wurde z.B. für die String-Klasse überschrieben
- für Wertetypen gelten die gleichen Prinzipien: ReferenceEquals für Referenzvergleich (macht aber keinen Sinn, da Wertetypen geboxt werden müssen, und dadurch immer verschiedene Referenzen aufweisen), Equals für Wertevergleich (wurde für ValueTypes überschrieben) und == als Zwischenlösung (muss eigenständig implementiert werden)

Operator Overloading

- beim Ausführen von Operatoren sucht der Compiler nach dem besten Matching für die Parameter
- alle Operatorüberladungen müssen als public und static deklariert werden

```
public static Vector operator + (Vector lhs, Vector rhs) {}
```

- die Reihenfolge der Parameter wird strikt eingehalten
- Vergleichsoperatoren müssen immer paarweise überladen werden und ein *bool* zurückgeben

User-Defined Casts

- auch für benutzerdefinierte Casts muss angegeben werden, ob sie implizit (erlaubt auch explizites Umwandeln) oder explizit (erlaubt kein implizites Umwandeln) sein sollen

```
public static implicit operator float (Currency value) {}
```

- der Parameter in der Klammer gibt den Ausgangsdatentyp an, der Rückgabewert ist der Datentyp, in den der Wert umgewandelt werden soll
- mit Hilfe von *System.Convert* können sichere Umwandlungen ohne Rundungsfehler durchgeführt werden

```
Convert.ToUInt16()
```

- einen benutzerdefinierten Cast in eine ableitete Klasse ist nicht möglich, da es diese schon gibt
- bei Casts zwischen zwei Klassen muss dieser in der Definition von einen der beiden Klassen definiert werden
- bei den systeminternen Umwandlungen von Ober- und Unterklassen erfolgt keine Datenumwandlung, sondern es wird lediglich eine neue Referenz auf das alte Objekt gesetzt, ohne es zu verändern
- ist eine andere Umwandlung gewünscht, so kann in der abgeleiteten Klasse ein Konstruktur erstellt werden, der als Parameter die Oberklasse aufnimmt
- wird eine Wertetyp in ein Referenztyp (wie Object) umgewandelt, erfolgt das Boxing durch das Anlegen eines Objektes mit den Werten des Wertetyps
- auch ein mehrstufiges implizites Casten ist möglich, wenn zwischen den Datentypen jeweils implizite Casts existieren

Kapitel 7: Delegates & Events

Delegates

- Beispiele, wo Methoden andere Methoden aufrufen müssen
  - Starten von parallelen Threads, der Konstruktor der Thread-Klasse benötigt als Parameter die auszuführende Methode

    ```
    System.Threads.Thread.Start()
    ```

  - Generische Bibliotheksklassen, zum Beispiel ein sortierbares Array von Objekten: nur der Client Code weiß, wie die enthaltenen Objekte sortiert bzw. verglichen werden sollen
  - bei einem Event muss die nachfolgende Methode als Parameter übergeben werden
- Delegates sind spezielle Typen von Objekten, da sie statt Daten die Adresse von (statischen oder sich auf Instanzen beziehenden) Methoden beinhalten
- Bei der Definition von Delegates müssen die benötigten Parameter und der Rückgabetyp angegeben werden

```
delegate void intMethodInvoker(int x);
```

- Konstrukturen von Delegates haben genau einen Parameter, der die Methode aufnimmt, auf den der Delegate zeigen soll; die Signatur muss dabei identisch sein

```
private delegate string GetAString();
static void Main()
{
    int x = 40;
    GetAString firstStringMethod = x.ToString;
    // gleichbedeutend mit: GetAString firstStringMethod = new GetAString(x.ToString);
```

```
    // x.ToString ist die Adresse der Methode, x.ToString() würde sie aufrufen
    Console.WriteLine("String is " + firstStringMethod());
}
```

Anonymous Methods

- eine anoyme Methode ist ein Codeblock der als Parameter an den Delegaten übergeben wird

```
delegate string DelegateTest(string val);
DelegateTest anonDel = delegate(string param)
{
    param += "This was added to the string.";
    return param;
}
Console.WriteLine(anonDel("Start of string.");
```

- wird benutzt, wenn der Code nur einmal für einen Delegationsaufruf verwendet wird
- es darf keine Sprunganweisungen aus dem anonymen Code raus geben, außerdem kann auf unsafe code und ref- sowie out-Parameter nicht zugegriffen werden

Multicast Delegates

- ein Delegat kann mehrere Methoden ausführen, wobei die Methoden als Rückgabewerte sinnvollerweise nur void verwenden sollten

```
delegate void DoubleOp(double value);
DoubleOp operations = MathOperations.MultiplyByTwo;
operations += Math.Operations.Square;
```

- mit -- und -= kann sich eine Methode vom Delegaten abmelden
- die Reihenfolge der durchgeführten Methoden kann nicht von der Runtime gesichert werden
- wird in einer Methode eine Exception geworfen, werden die anderen angemeldeten Methoden nicht mehr ausgeführt → Lösung: eigene Iteration über *GetInvocationList*

```
DemoDelegate d1 = One;
d1 += Two;
Delegate[] delegates = d1.GetInvocationsList();
foreach (DemoDelegate d in delegates)
{
    try
    {
            d();
    }
    catch (Exception)
    {
            Console.WriteLine("Execution caught");
    }
}
```

Events

- auch Events sind Multicast Delegates, an denen sich Methoden mit += anmelden können
- Event-Handler haben immer void als Rückgabetyp und folgen der Namenskonvention Objekt_Event

```
private void Button_Click (object sender, EventArgs e) {}
```

- beim Deklarieren eines Events muss der Eventhandler angegeben werden

```
public event SampleEventHandler SampleEvent;
```

Kapitel 8: Strings & Regular Expressions

System.String

- String-Objekte können nicht geändert werden, bei Veränderungen an dem Objekt wird stets ein neues Objekt erzeugt

- die *StringBuilder*-Klasse bietet nicht so viele Operationen an wie String, arbeitet aber beim Ersetzen, Anhängen oder Entfernen von Textabschnitten sehr effizient

```
StringBuilder s = new StringBuilder("hallo");
```

- StringBuilder reserviert mehr Platz auf dem Heap, als er benötigt, weswegen er zwei Hauptproperties besitzt: *Length* (Länge des Strings) und *Capacity* (reservierter Speicherplatz)
- mit *MaxCapacity* kann der maximal erreichbare Speicherplatz angegeben werden
- System.WriteLine benutzt intern die statische Methode String.Format, welche wiederum StringBuilder.AppendFormat aufruft, um formatierte Strings anzuzeigen
- AppendFormat prüft, ob der übergebene Datentyp das Interface IFormattable implementiert hat
- um eigenen Klassen eine formatierte Ausgabe zu ermöglichen, muss das Interface IFormattable implementiert werden und der übergebene String ausgewertet werden

```
struct Vector : IFormattable
{
    public string ToString (string format, IformatProvider formatProvider)
    {
            if (format == null)
                    return ToString();
            string formatUpper = format.ToUpper();
            switch (formatUpper)
            {
                    case "N":
                            return "||" + Norm().ToString() + "||";
                    …
            }
    }
}
```

Regular Expressions

- im Namespace *System.Text.RegularExpressions* werden hilfreiche Klassen zum Auswerten von regulären Ausdrücken angeboten
- die *Matches()*-Methode sucht in einem angegeben String nach Mustern und liefert die gefundenen Ergebnisse als *MatchCollections* zurück

```
MatchCollection Matches = Regex.Matches(„Zu untersuchender
Text",“te",RegexOptions.IgnoreCase | RegexOptions.ExlicitCapture);
```

- mit *Match.Index* kann die Stelle des gefundenen Wertes ausgegeben werden
- „\b“ stellt den Anfang oder den Ende eines Wortes dar, „\S“ stellt jedes beliebige Zeichen außer einem Leerzeichen dar, „\S*“ heißt beliebig viele Zeichen
- möchte man nach einen meta-character suchen, muss man ein „\“ voranstellen
- alternative Zeichen können durch ein „[a|b]“ dargestellt werden, einen Bereich von Zahlen kann z.B. durch „[0-9]+“ dargestellt werden
- Zeichen können zu Gruppen zusammengefasst werden, so werden mit „(an)+“ an, anan, ananan etc. gefunden
- ein Match-Objekt besteht aus einer GroupCollection, jede Group besitzt eine CaptureCollection mit beliebig vielen Captures
- mit „?:“ wird dem Compiler mitgeteilt, dass eine Gruppe nicht gespeichert werden soll

Kapitel 9: Generics

Overview

- Vorteile von Generics

- Geschwindigkeit: im Gegensatz zu nicht generischen Collection-Klassen wird kein Boxing und Unboxing mehr benötigt
- Typensicherheit: da bei der Erstellung von generischen Collection-Klassen der Typ festgelegt wird, werden falsche Zuweisungen schon vom Compiler entdeckt
- Wiederverwendung: eine generische Klasse kann für verschiedene Datentypen und in verschiedenen .NET-Sprachen verwendet werden

Generic Classes' Features

- anstelle von *null* kann einem generischen Typ *default* zugeordnet werden, da Wertetypen kein null unterstützen (sie werden mit default auf 0 gesetzt)
- setzt eine generische Klasse voraus, dass der eingesetzte Datentyp z.B. ein bestimmtes Interface implementieren muss, so kann dies über Constraints festgelegt werden

```
public class DocumentManager<TDocument>
    where TDocument : IDocument
```

- ist ein Constraint gesetzt, so kann z.B. in einer foreach-Schleife so auf Eigenschaften des Interfaces zugegriffen werden, als würde die Klasse sie selbst beinhalten

```
foreach (TDocument doc in documentQueue)
    Console.WriteLine(doc.Title);
```

- es gibt Constraints für Oberklassen, Interfaces und Standardkonstruktoren, welche z.T. kombiniert werden können
- generische Klassen können als Oberklasse dienen, jedoch muss in der abgeleiteten Klasse der generische Datentyp wiederholt (für eine generische Unterklasse) oder spezifiziert (für eine nicht-generische Unterklasse) werden
- statische Member existieren für jede Klasseninstanz einer generischen Klasse
- es können generische Interfaces definiert werden, die generische Parameter aufnehmen
- beim Aufruf von generischen Methoden muss der Datentyp nicht explizit angegeben werden, da der Compiler den Typ anhand der übergebenen Parameter bestimmen kann

Other Generic Framework Types

- Nullable-Datentypen sind ebenfalls generisch, so ist *int?* nur eine Kurzform für *Nullable<int>*
- der EventHandler <TEventArgs> verringert die Anzahl der benötigten EventHandler

```
public sealed delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
    where TEventArgs : EventkonArgs
```

- mit *ArraySegment<T>* können Teile eines Arrays verwaltet und übergeben werden

Kapitel 10: Collections

Collection Interfaces and Types

- Collection-Klassen können unterteilt werden in Collections, die Daten vom Typ Object verwalten, generische Collections und spezialisierte Collections (wie z.B. StringCollection)
- Collection-Klassen implementieren eine Reihe von Interfaces, die es zum Teil auch in generischen Varianten gibt: IEnumerable (für foreach), ICollection, IList (für Indexer), IDictionary, IComparer, IEqualityComparer

Lists

- Object-Version *ArrayList* , generische Version *List<T>*
- implementiert die Interfaces IList, ICollection und IEnumerable

- bei Aufruf des Standardkonstruktors wird eine leere Liste erzeugt, beim ersten Hinzufügen erhöht sich die Kapazität auf vier Objekte, beim Hinzufügen des fünften Elements auf acht Objekte usw., wobei bei jeder Erhöhung der Kapazität ein komplett neuer Speicherblock durch die Liste belegt wird
- es kann dem Konstruktor auch zu Beginn die Kapazität mitgeteilt werden, die sich dann bei Bedarf auch wieder verdoppeln kann
- statt einer gewöhnlichen foreach-Schleife bietet die Klasse List<T> eine *ForEach()*-Methode, die eine Methode als Parameter (*Action<T>*) aufnimmt
- für die Suche gibt es ebenfalls einen Delegaten: *FindIndex(Predicate<T> match)*
- es werden mehrere Möglichkeiten zum Sortieren der Liste angeboten, der Gebrauch der *Sort()*-Methode ohne Parameter ist nur möglich, wenn die Listenelemente das Interface IComparable implementiert haben
- mit *ConvertAll()* können alle Elemente der Collection in einen anderen Datentyp umgewandelt werden
- die Methode *AsReadOnly()* liefert eine *ReadOnlyCollection<T>* zurück

Queue

- Object-Version *Queue*, generische Version *Queue<T>*
- Queue implementiert die Interfaces ICollection, IEnumerable und IClonable, Queue<T> die Interfaces IEnumerable<T> und ICollection
- mit *Enqueue()* wird ein neues Element angehangen, mit *Dequeue()* erhält man das erste Element der Schlange
- die Kapazitäten der Klasse werden analog zu den Listenklassen gehandhabt

Stack

- ähnlicher Aufbau wie Queue, nur LIFO statt FIFO
- ein Element wird mit *Push()* angehangen und mit *Pop()* ausgegeben

Linked Lists

- es gibt nur die generische Variante LinkedList<T>
- implementiert die Interfaces ICollection<T>, IEnumerable<T>, ICollection, IEnumerable, ISerializable und IDeserializationCallback
- das Einfügen in der Mitte der Liste funktioniert sehr schnell, das Finden eines Elements dauert aber verhältnismäßig lange, da die Liste nur sequentiell durchlaufen werden kann

Sorted Lists

- die Klasse SortedList<TKey, TValue> sortiert die Elemente anhand des Schlüssels
- das Einfügen eines Elements kann über die Add()-Methode erfolgen, die als Parameter den Schlüssel und den Wert aufnimmt, oder direkt über den Indexer mit Hilfe des Schlüssels
- neben der generischen Klasse gibt es auch noch die nicht-generische Klasse SortedList

Dictionaries

- auch als HashTable oder Map bekannt
- erlaubt das schnelle Nachschlagen mit Hilfe von Schlüsseln
- die Klasse TDictionary<TKey, TValue> ist ähnlich aufgebaut wie SortedList<TKey, TValue>

- wann immer eine Klasse die Speicherstelle eines Elements in Erfahrung bringen will, ruft es die Methode *GetHashCode()* auf
- falls zwei Elemente den gleichen Hash Code besitzen, überprüft das Dictionary die Objekte mit Hilfe der Equals-Methode
- die Klasse Object prüft mit Hilfe von Equals, ob die Speicheradresse von zwei Elementen gleich ist, GetHashCode() basiert deswegen auf den Speicheradressen (Nachteil: zwei Elemente mit gleichen Werten erhalten trotzdem verschiedene Hash Codes)
- die Klasse SortedDictionary<TKey, TValue> ist als binärer Suchbaum implementiert (im Vergleich zu SortedList ist diese Klasse schneller beim Einfügen und Entfernen von unsortierten Daten, jedoch verbraucht SortedList weniger Speicherplatz)

Bit Arrays

- die Klassen *BitArray* und *BitVector32* stellen Klassen zum Verarbeiten von einer Anzahl von Bits
- während BitArray größenveränderbar ist, stellt BitVector32 immer 32 Bits dar, ist jedoch durch die Speicherung eines Integers auf dem Stack deutlich performanter

Kapitel 11: Memory Management & Pointers

Memory Management under the Hood

- das Deallokieren von Wertetypen auf dem Stack erfolgt in umgekehrter Reihenfolge wie das Allokieren der Daten
- der Stack Pointer zeigt zunächst auf das Ende des für den Stack reservierten Speicherbereichs, da sich der Stack abwärts von hohen Speicheradressen zu niedrigen füllt
- beim Deklarieren eines Referenztyps wird auf dem Stack lediglich eine 4 Byte große Referenz für ein Objekt der Klasse angelegt, mit new() wird auf dem Managed Heap der Speicherplatz für das wirkliche Objekt belegt und die Referenz im Stack zeigt nun auf diesen Speicherbereich im Heap
- im Gegensatz zum Stack füllt sich der Heap aufwärts bzgl. der Speicheradressen
- der Garbage Collector entfernt alle Speicherobjekte im Heap, für die es keine Referenz auf dem Stack mehr gibt und defragmentiert anschließend den Heap

Freeing Unmanaged Resources

- der Garbage Collector kann unmanaged resources (wie z.B. Dateihandle oder Netzwerk- und Datenbankverbindungen) nicht aus dem Speicher entfernen
- zum Freigeben der unmanaged resources kann zum einen ein Destruktor (Finalizer) deklariert werden, und zum anderen kann das Interface IDisposable implementiert werden
- der Destruktor hat den gleichen Namen wie die Klasse mit einem vorangestellten ~, aber ohne Parameter und ohne Modifizierer

```
class MyClass
{
    ~MyClass()
    {
            //destructor implementation
    }
}
```

- der Compiler übersetzt den Destruktor in die entsprechende Finalize()-Methode

```
protected override void Finalize()
{
    try
    {
```

```
            //destructor implementation
        }
        finally
        {
            base.Finalize();
        }
    }
```

- der Aufruf des Destruktors durch die CLR ist indeterministisch und kostet Performance, da der Garbage Collector beim ersten Durchgang den Destruktor aufruft, und erst beim zweiten Durchgang das Objekt wirklich löscht
- die Alternative zum Destruktor ist das Interface IDisposable, welches deterministisch abläuft und als einzige Methode Dispose bereitstellt, die keine Parameter aufnimmt und keinen Rückgabewert besitzt

```
class MyClass : IDisposable
{
    public void Dispose()
    {
        //implementation
    }
}
```

- mit Hilfe der using-Anweisung wird nach dem letzten Gebrauch einer Instanz automatisch Dispose aufgerufen, auch wenn eine Exception aufgetreten ist

```
using (MyClass c1 = new MyClass())
{
    //do your processing
}
```

- die Vorteile beider Verfahren lassen sich kombinieren, in dem man beide implementiert

Unsafe Code

- C# erlaubt nicht den direkten Zugriff auf Inhalte, die von Pointern referenziert werden
- zwei Hauptgründe zum Einsatz von Pointern:
  - Abwärtskompatibilität: zum Ausführen von nativen Windows API-Funktionen werden z.T. Pointer benötigt
  - Performance: Pointer sind eine sehr effiziente Methode zum Aufrufen und Manipulieren von Daten
- der Einsatz von Pointern ist nur in als *unsafe* gekennzeichneten Bereichen möglich

```
unsafe int GetSomeNumber()
{
    //code that can use pointers
}
```

- wird eine Klasse als unsafe deklariert, so sind alle Member implizit unsafe
- eine lokale Variable kann nicht als unsafe deklariert werden
- um unsafe-Code auszuführen, muss dem Compiler dies explizit über eine Compiler-Option mitgeteilt werden
- zur Deklaration eines Pointers wird dem Datentyp ein * angehängt

```
int* pWidth;
```

- der &-Adress-Operator konvertiert einen Wertetyp in einen Pointer und speichert die Adresse ab
- der *-Indirection-Operation konvertiert einen Pointer in einen Wertetyp und speichert den Inhalt ab

```
int x = 10;
int* pX;
pX = &x;
*pX = 20; //x has the value 20 now
```

- Pointer können nur Wertetypen im Stack referenzieren, das Referenzieren auf Wertetypen im Heap ist nicht möglich
- eine Konvertierung von Pointer in Integertypen (am sinnvollsten uint) und andersrum ist ohne Probleme explizit möglich, da beide 4 Bytes belegen
- eine Konvertierung von Pointern in Pointer, die auf andere Datentypen zeigen, ist ebenfalls möglich
- falls der zu referenzierende Datentyp nicht spezifiziert werden soll, kann ein void-Pointer erzeugt werden
- mit Pointern können arithmetische Operationen durchgeführt werden: addiert man zu einem Double-Pointer 1 dazu, so wird die Speicheradresse um 1 * Größe des Datentyps = 8 Bytes erhöht
- der sizeof-Operator liefert die Größe eines Wertetyps zurück
- ein Pointer kann nur auf ein Struct zeigen, wenn dieses keine Referenztypen beinhaltet
- über einen Pointer kann auch auf Member eines structs zugegriffen werden

```
(*pStruct).X = 4;
pStruct->X = 4; //means the same
```

- Pointer können direkt auf Member in Structs zeigen

```
long* pL = &(pStruct->X);
```

- wenn Pointer auf Wertetypen innerhalb von Klassen referenzieren, muss das fixed-Schlüsselwort benutzt werden, welches dem Garbage Collector mitteilt, das Objekt nicht zu löschen, falls Code nach einem fixed-Pointer folgt

```
fixed (long* pObject = &(myObject.X))
{
    //do something
}
```

Kapitel 12: Reflection

Means of Reflection

- Reflektion ermöglicht die folgenden Aktionen:
  - Durchlaufen der Member eines Typs
  - Instanziierung von neuen Objekten
  - Ausführen von Membern eines Objekts
  - Anzeigen von Typinformationen
  - Anzeigen von Assemblyinformationen (Manifest)
  - Untersuchen der benutzerdefinierten Attribute für einen Typ
  - Erzeugen und Kompilieren einer neues Assembly

Custom Attributes

- werden als Metadaten durch den Compiler in der Assembly gespeichert
- eigene Attribute müssen von der Klasse System.Attribute abgeleitet werden
- für jedes Attribut können die folgende Aspekte angegeben werden
  - Pflichtangabe: mit welchen Typen kann das Attribut angewendet werden (Klassen, Structs etc.)

    ```
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Class)]
    ```

  - kann das Attribut mehrfach einem Element zugeordnet werden

    ```
    AllowMultiple=true;
    ```

  - wird das Attribut von abgeleiteten Klassen und Interfaces vererbt

    ```
    Inherited=false;
    ```

  - welche obligatorischen und welche optionalen Parameter nimmt das Attribut auf

- eigene Parameter für ein Attribut können im Konstruktur des Attributs angegeben werden
- optionale Parameter können über Properties definiert werden
  ```
  [Fieldname(„SocialSecurityNumber", Comment="This is the primary key field")]
  ```
- bei Verwendung eines Attributs kann die Kurzform (z.B. FieldName) oder die Langform (z.B. FieldNameAttribute) angegeben werden

Reflection

- es gibt drei Arten, die Typ-Referenz eines gegebenen Datentyps zu ermitteln
  - typeof-Operator der Klasse Type
    ```
    Type t = typeof(double)
    ```
  - GetType()-Methode der Klasse System.Object
    ```
    Type t = d.GetType();
    ```
  - statische GetType()-Methode der Klasse Type
    ```
    Type t = Type.GetType(„System.Double");
    ```
- Type stellt eine Vielzahl von Properties bereit, mit denen der Name und der Namespace, der Basistyp und der darunterliegende .NET-Basistyp sowie weitere boolsche Eigenschaften zur Art und zu den Zugriffsmodifizierern des zu untersuchenden Typs ausgelesen werden können
- es werden viele Methoden bereitgestellt, die alle oder einige Member eines Datentyps zurückliefern, z.B. *GetMethod()/GetMethods()* mit dem dazugehörigen Objekt *MethodInfo* oder *GetMember()/GetMembers()* mit *MemberInfo*
- die Klasse *System.Reflection.Assembly* bietet analog eine Vielzahl von Methoden und Properties, um Metadaten über Assemblies auszulesen und Assemblies auszuführen
- zunächst muss aber die Assembly über *Assembly.Load()* geladen werden

Kapitel 13: Errors & Exceptions

Looking into Errors and Exception Handling

- Zwei Hauptklassen erben von der Klasse *System.Exception*
  - *System.SystemException*: hier werden allgemeine Exceptions wie *ArgumentException* oder *StackOverflowException* definiert
  - *System.ApplicationException*: hier werden Exceptions definiert, die nur in einzelnen Programmen auftreten
- viele abgeleitete Exception-Klassen fügen keine neue Funktionalität zur Basisklasse hinzu, es soll einfach nur der Fehlergrund spezifiziert werden
- die Fehlerbehanldung läuft im Allgemeinen in drei Schritten ab
  - *try*: kapselt den Codeabschnitt, in dem Fehler auftreten können
  - *catch*: behandelt die auftretenden Fehler
  - *finally* (optional): wird unabhängig vom Auftreten von Fehlern immer ausgeführt, dient meist zum Schließen und Beenden von externen Ressourcen
    ```
    try
    { /* code for normal execution */ }
    catch
    { /*error handling */ }
    finally
    { /* clean up */ }
    ```
- beim Auftreten eines Fehlers wird ein Fehlerobjekt erzeugt
  ```
  throw new Exception();
  ```
- die erste Fehlerbehandlungsmethode, die im catch-Block auf die geworfene Exception passt, wird durchgeführt (deshalb sollten die speziellsten Fehler immer zuerst gefangen werden)

- wird zunächst die allgemeinste Exception im catch-Block gefangen, und folgen dann noch andere Catch-Blöcke, so wird der Code nicht kompiliert, da die nachfolgenden Abschnitte nie erreicht werden können
- will man Exceptions fangen, die von anderen Sprachen erzeugt wurden und nicht von System.Exception erbene, so kann man dies in einem allgemeinen catch-Block tun
  `catch {}`
- wird eine Exception nicht vom eigenen Programm aufgefangen, so fängt die .NET-Runtime sie (das ganze Programm läuft in einem riesigen try-catch-Block)
- Tritt ein Fehler in einem geschachtelten try-Block auf, so wird zunächst versucht, im inneren catch-Block den Fehler zu fangen. Gelingt dies nicht, wird der innere finally-Block ausgeführt und versucht, den Fehler im äußeren catch-Block zu fangen.
- in geschachtelten try-Blöcken wird häufig der Exceptiontyp geändert oder Informationen zur Exception hinzugefügt

Kapitel 16: Assemblies

What are Assemblies?

- Nachfolger der DLLs, die globale Funktionen bereit stellen
- DLL-Hölle: alte DLL-Versionen werden von neueren überschrieben, ohne abwärtskompatibel zu sein, oder neuere Versionen werden durch alte überschrieben
- einige Lösungsversuche existieren, die aber lediglich die Symptome und nicht die Ursache auflösten (side-by-side DLLs, file protection für systeminterne DLLs)
- die Installation einer Assembly erfolgt durch Kopieren all ihrer enthaltenen Dateien
- das Laden von Assemblies erfolgt side-by-side, so dass innerhalb eines Programms auch mehrere Versionen einer Assembly geladen werden können
- die Versionierungsabhängigkeiten einer Assembly werden in seinem Manifest gespeichert
- private Assemblies brauchen keine Rücksicht auf Namenskonflikte oder Versionierungsprobleme zu nehmen, da sie im gleichen oder einem Unterverzeichnis des Programms gespeichert werden

Assembly Structure

- Assemblies sind selbstbeschreibende Installationseinheiten, die aus einer oder mehreren Dateien bestehen und beinhalten Metadaten, Ressourcen, DLLs und eine EXE-Datei
- ein Modul ist eine DLL ohne Assembly Metadaten und kann in eine Assembly geladen werden
- Module unterstützen die Mehrsprachigkeit, da jedes Modul in einer anderen .NET-Sprache geschrieben und dann in eine gemeinsame Assembly geladen werden kann

Cross-Language Support

- das CTS legt eine Reihe von Regeln fest, wie Compiler bei der Definition, der Referenzierung und dem Gebrauch von Referenz- und Datentypen beachten müssen
- bei Beachtung der CTS können Objekte, die in verschiedenen Sprachen geschrieben wurden, miteinander interagieren
- da nicht alle Datentypen von allen .NET-Programmiersprachen unterstützt werden (z.B. kennt Visual Basic kein uint), fasst das CLS die Sprachelemente zusammen, um von jeder Sprache auf den Code zugreifen zu können
- nur öffentliche und protected Member müssen CLS-konform sein, auf private Member kann von außerhalb gar nicht zugegriffen werden

XML-Serialisierung

- bei der XML-Serialisierung werden nur öffentliche Felder und Eigenschaftswerte von Objekten serialisiert, Methoden werden nicht serialisiert
- damit Properties serialisiert werden, muss neben dem get-Accessor auch der set-Accessor implementiert werden
- eine zu serialisierende Klasse muss immer einen Standardkonstruktur besitzen und öffentlich sein
- mit Hilfe der Klasse XMLSerializer wird ein XML-Stream erzeugt, der dem XML-Schema entspricht
- Serialisieren von Klassen

```
MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new XmlSerializer(typeof(MySerializableClass));
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
```

- Deserialisieren von Klassen

```
MySerializableClass myObject;
XmlSerializer mySerializer = new XmlSerializer(typeof(MySerializableClass));
FileStream myFileStream = new FileStream("myFileName.xml", FileMode.Open);
myObject = (MySerializableClass)mySerializer.Deserialize(myFileStream);
```

Refactoring

- Refaktorisieren ist der Prozess, ein Softwaresystem so zu verändern, dass das exteren Verhalten nicht geändert wird, der Code aber eine bessere interne Struktur erhält
- die Bedeutung des Refactorings stieg mit der Einführung von agilen Softwareentwicklungsprozessen (mehr Kommunikation, kürzere Entwicklungszeiten, weniger Dokumentation)
- der erste Schritt muss immer das Erstellen von validen Testfällen sein
- die Tests sollten selbstüberprüfend sein
- Zerlegen und Verschieben von langen Methoden, Umbennen von lokalen Variablen zum bessern Verständnis des Codes, temporäre Variablen durch Abfragen ersetzen (allerdings zu Lasten der Performance)