Fakultät Informatik/Mathematik

Lehrstuhl für Modellierung / Simulation

Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Vorlesungsmitschrift

PROGRAMMIERUNG VON KOMPONENTENARCHITEKTUREN

Mitschrift von

Falk-Jonatan Strube

Vorlesung von

Prof. Dr. rer. nat. Sebastian Aland

4. Juli 2018

INHALTSVERZEICHNIS

1 Einführung **9**
1.1 Microsoft .NET Platform . . . 9
1.1.1 .NET Versionen und Komponenten . . . 9
1.1.2 Features . . . 10
1.1.3 Compilierung in .NET . . . 11
1.1.3.1 C# Code . . . 11
1.1.3.2 Visual Basic Code . . . 11
1.1.3.3 Entsprechender IL Code . . . 11
1.1.4 Ausführung des Programms . . . 12
1.1.5 Teile von .NET . . . 12
1.1.6 Namespaces . . . 13
1.2 Die Sprache C# . . . 13
1.2.1 Features . . . 13
1.2.2 .NET Core . . . 15
1.3 Gliederung des Kurses . . . 15

2 Grundlagen C# **16**
2.1 Intrinsische Datentypen . . . 16
2.2 Boxing / Unboxing . . . 19
2.2.1 Member der intrinsischen Datentypen . . . 19
2.2.2 Arbeiten mit Strings . . . 19
2.2.3 Textformatierung . . . 20
2.2.4 Escape Characters . . . 21
2.2.5 Widening / Narrowing . . . 21
2.2.6 Schlüsselwort var . . . 21
2.3 Schleifen . . . 22
2.4 Garbage Collector . . . 22
2.5 Operatorüberladung . . . 23

3 Methoden und Parameter Modifizierer **24**
3.1 Ausgabeparameter (out) . . . 24
3.2 Referenzparameter (ref) . . . 24
3.3 Beliebig viele Parameter (params) . . . 25
3.4 pass-by-value . . . 25
3.5 Veranschaulichung . . . 25

4 Array / Struct / Enum **27**
4.1 Enumeration (Aufzählung) . . . 27
4.1.1 Eigenschaften . . . 27
4.1.2 Einen enum durchlaufen . . . 27
4.2 Struct . . . 27
4.3 Array . . . 28

5 Objekt-orientierung in C# **29**
5.1 Vererbung . . . 29

5.2 Der Konstruktor . . . 30
5.3 Polymorphie . . . 31
5.4 Schlüsselwort-Optionen für Klassenmethoden . . . 31
5.5 Optionen für Klassendeklaration . . . 32
5.6 Optionen für Variablen . . . 32
5.7 Zugriffsmodifizierer für Sichtbarkeit . . . 32
5.8 Casting von Klassen . . . 32
5.9 Properties (Eigenschaft) . . . 33
5.9.1 Auto-generated properties . . . 34
5.10 Object initialization Syntax . . . 34
5.11 Anonyme Objekte . . . 35

6 Interfaces (Schnittstellen) 36
6.1 Nutzen: Ersatz für fehlende Mehrfachvererbung . . . 36

7 Generik 38
7.1 Generische Klasse definieren . . . 38
7.2 Generische Klasse benutzen . . . 38
7.3 Generische Methoden . . . 39
7.4 Template Constraints . . . 39
7.4.1 Interface Constraint . . . 39
7.4.2 Klassen Constraint . . . 39
7.4.3 Vererbungs Constraints . . . 40
7.4.4 Instanzierbarkeits Constraint . . . 40
7.5 Beispiel . . . 40
7.5.1 Aufgabe . . . 40
7.5.2 Antwort . . . 41

8 Exceptions 42
8.1 Behandlung einer Ausnahme mit try / catch . . . 42
8.2 Arten von Exceptions . . . 42
8.3 Ausnahmen werfen mit throw . . . 43
8.4 Finally Block . . . 43
8.5 Eigenschaften der Klasse Exception . . . 44
8.6 Innere Exceptions . . . 44

9 Delegates und Events 45
9.1 Delegate . . . 45
9.1.1 Multicast Delegates . . . 46
9.1.2 Generische Delegates . . . 46
9.1.3 Lambda Funktionen . . . 46
9.1.4 Beispiele . . . 47
9.2 Ereignisse . . . 48
9.2.1 Ereignisse mit Übergabeparametern . . . 49

10 Windows Forms 51
10.1 Windows Forms Anwendung erstellen . . . 51
10.1.1 Form.cs . . . 51
10.1.2 Form.Designer.cs . . . 52
10.1.3 Program.cs . . . 52
10.2 Ein neues Control generieren . . . 52
10.3 Hosting . . . 53

10.4 Auf Events reagieren . . . 53
10.5 Benutzerdefiniertes Zeichnen . . . 53
10.6 Transformation von Graphikobjekten . . . 54
10.7 Persistenz . . . 54
10.8 Events . . . 55
10.8.1 Mausereignisse . . . 55
10.8.2 Timerereignisse . . . 57
10.8.3 Tastaturereignisse . . . 57
10.8.4 Parallelität von Ereignissen . . . 58

11 Sockets **59**
11.1 Netzwerkprogrammierung mit der Klasse Socket . . . 59
11.1.1 Kommunikation mit Sockets . . . 59
11.1.2 Funktionalität . . . 59
11.2 Arbeiten mit Socket-Objekten . . . 60
11.2.1 Synchrone und Asynchrone Socketfunktionen . . . 60
11.2.2 Portwahl bei Client und Server . . . 61
11.2.3 Interprozesskommunikation . . . 62
11.2.4 Socketanzahl . . . 62
11.3 Synchroner Datenaustausch . . . 63
11.4 Synchron VS Asynchron . . . 64
11.4.1 Beispiele . . . 64
11.5 Sockets trennen und beenden . . . 65

12 DLLs/COM **66**
12.1 Externe DLLs einlesen/nutzen . . . 66
12.2 COM-Komponenten . . . 68

13 Parallele Programmierung **70**
13.1 Einleitung . . . 70
13.1.1 CPU Entwicklung . . . 70
13.1.2 Traditionelle Sequentielle Programmierung . . . 70
13.1.3 Parallele Programmierung . . . 71
13.1.4 Anwendungen paralleler Programmierung . . . 71
13.1.5 Moderne supercomputer . . . 72
13.1.6 Deutsche Supercomputer . . . 73
13.1.7 Supercomputer nach Ländern . . . 73
13.2 Multithreading . . . 74
13.3 Multithreading . . . 74
13.3.1 Verschiedene Methoden zur Parallelisierung: . . . 74
13.3.2 Thread . . . 74
13.3.3 Threadzustände . . . 75
13.3.4 Threads starten . . . 75
13.3.5 Beispiel . . . 75
13.3.6 Weitere Methoden von Thread . . . 76
13.4 Synchronisation von Threads . . . 76
13.4.1 Beispiel . . . 76
13.4.2 Datenabhängigkeiten . . . 76
13.4.3 Race Condition (Wettlaufsituation) . . . 77
13.4.4 Race Conditions vermeiden: . . . 77
13.4.5 Atomare Operationen . . . 77
13.4.6 Mutual exclusion (Mutex) . . . 78

13.4.7 Monitor . . . 78
13.4.8 Beispiel . . . 78
13.4.9 Lock . . . 79
13.5 Deadlock, Starvation . . . 79
13.5.1 Visualisierung im Wartegraph (ressource dependency graph) . . . 80
13.5.2 Verhindern von Deadlocks . . . 81
13.5.3 Starvation (Verhungern) . . . 81
13.6 Asynchrone Methoden . . . 81
13.6.1 Abgrenzung zu Multithreading . . . 81
13.6.2 Async Patterns . . . 82
13.6.2.1 Event-based Asynchronous Programming Design Pattern (EAP) 82
13.6.2.2 Vergleichen von Mustern . . . 82
13.6.3 Zusammenfassendes Beispiel . . . 83
13.7 Task Parallel Library . . . 85
13.7.1 Tasks vs Threads . . . 85
13.7.2 Laufzeitvergleich Task / Thread . . . 85
13.7.3 Die Klasse Task . . . 86
13.7.3.1 Task anlegen . . . 86
13.7.3.2 Task starten . . . 86
13.7.3.3 Task.Factory . . . 86
13.7.3.4 Rückgabewert eines Tasks . . . 86
13.7.3.5 Auf einen Task warten . . . 87
13.7.3.6 Auf mehrere Tasks warten . . . 87
13.7.3.7 Parallele Schleifen . . . 87
13.7.3.8 Lokale Schleifenvariablen . . . 87
13.7.3.9 Schleife abbrechen . . . 88
13.7.3.10 Parallelisierung von GUI Aktivitäten . . . 88
13.7.3.11 Wiederholung Parallel.For mit lokalen Schleifenvariablen . . . 89
13.8 Asynch/Await . . . 89
13.8.1 Modifizierer async . . . 90
13.8.2 Schlüsselwort await . . . 90
13.8.3 Rückgabewerte . . . 91
13.8.4 Rückgabewerte mit Tupel . . . 91

14 Iteratoren: IEnumerable, IEnumerator, yield return 93
14.1 IEnumerable/IEnumerator . . . 93
14.2 yield return . . . 94
14.2.1 Generischer IEnumerator . . . 95

15 C# Features: Attribute, Reflections, Extension methods 97
15.1 Attribute . . . 97
15.1.1 Attribut definieren . . . 97
15.1.2 Attribut nutzen . . . 97
15.1.3 Reflection (Attribute abfragen) . . . 97
15.2 Extension methods . . . 98
15.2.1 Verfahren . . . 98
15.2.2 Definition . . . 99

16 LINQ 100
16.1 Nutzen von Link . . . 100
16.2 Beispielszenario . . . 101
16.2.1 Ohne LINQ . . . 101

16.2.2 Mit LINQ . . . 102
16.3 LINQ Abfrage . . . 102
16.3.1 Syntax . . . 102
16.3.2 Gegenüberstellung . . . 103
16.3.3 Schlüsselwärter und (Standard-)Abfrageoperatoren . . . 103
16.3.4 Struktur . . . 104
16.3.5 from und select – Datenquelle und Auswahl . . . 104
16.3.6 Selektor . . . 105
16.3.7 Unterabfragen . . . 105
16.3.8 Select vs SelectMany . . . 106
16.3.9 Where . . . 107
16.3.10 Ausführen der Abfrage . . . 108
16.3.11 GroupBy . . . 110
16.3.11.1 Composite-Keys . . . 111
16.4 LINQ-Provider/Implementierungen/Flavors . . . 111
16.5 Weiteres . . . 112
16.5.1 Abfrageoperatoren . . . 112
16.5.2 Anwendungsbeispiel . . . 112

17 Wiederholung **113**
17.1 Lambda Expressions . . . 113

ORGANISATORISCHES

Gliederung

- Grundlagen C#
- Vererbung, Polymorphie, partielle Klassen
- Interfaces
- Generische Programmierung
- Fehlerbehandlung Exceptions
- Delegates und Ereignisse
- Windows Forms
- Sockets / LINQ / ASP.NET
- Parallele Programmierung
- Test-basierte Entwicklung
- Agile / emergente Entwicklung

PRÜFUNG

- Prüfung am PC (PC-Pool) → hauptsächlich Programmierung, ggf. ein wenig Verständnisfragen
- Skripte/Notizen/Bücher erlaubt (jedes nicht-elektronische Hilfsmittel)

THEMEN DER PRÜFUNG

- Klassen und Konstruktoren anlegen
- Properties
- Polymorphie / Methoden überschreiben
- Interfaces
- Exceptions
- generische Collections (List)
- Windows Forms
- Parallelisierung
  - Parallel.For
  - Threads / Tasks

  - Locking / Thread-Safety
  - Async / Await
- Iteratoren (IEnumerable, IEnumerator)
- Lambda Expressions

NICHT THEMEN DER PRÜFUNG

- Test-Driven Development
- Sockets, DLL & COM
- Attribute, Reflections
- LINQ

UMFANG DER PRÜFUNG

- 2 große Aufgaben (jeweils ca. 1 Stunde Programmierzeit)

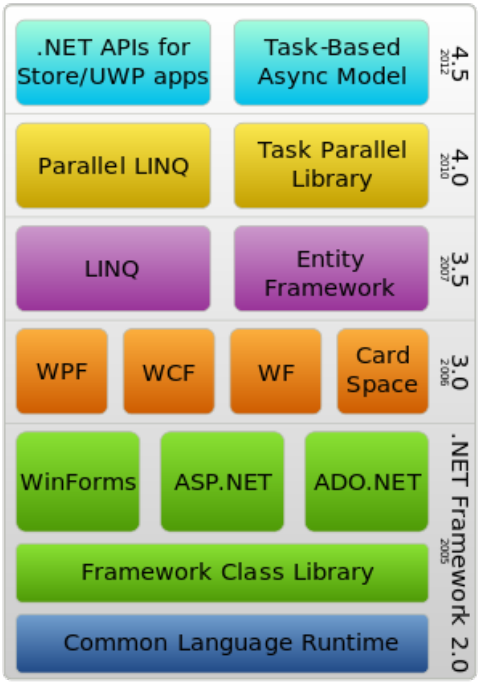
1 EINFÜHRUNG

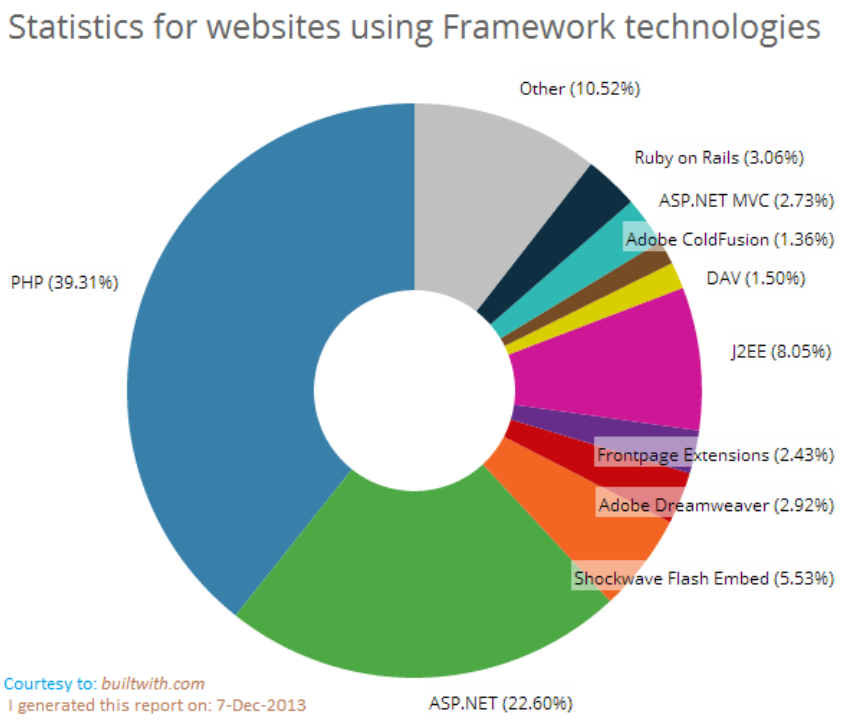
1.1 MICROSOFT .NET PLATFORM

Vorlesung
24.03.2017

- 2002 eingeführt
- Hauptprogrammierplattform für Windows

1.1.1 .NET VERSIONEN UND KOMPONENTEN





1.1.2 FEATURES

- Komponentenbasiert:
  Unterschiedliche Komponenten aus unterschiedlichen Programmiersprachen:
  - C#, Visual Basic, F#, JavaScript, C++ (in Visual Studio)
  - Ruby, Python, COBOL, Pascal (externe Compiler)
  - Liste von CLI Sprachen
  - Vererbung, Exception Handling, Debugging (über mehrere Programmiersprachen hinweg)
- Versionskompatibilität: mehrere Versionen der gleichen .dll können nebeneinander existieren
- .NET Distributionen:
  - Microsoft, included in Visual Studio
  - Andere Distributionen (www.mono-project.com, www.dotgnu.org) stellen Compiler und Runtime Environment für Linux Android, IOS, ...
  - .NET Core (plattformunabhängig, schlank)
- Erstellte Anwendungen sind Plattform-unabhängig, und laufen unter verschiedenen Betriebssystemen (Windows, MacOS, Linux, Android, ...)

Warum mehrere Programmiersprachen?

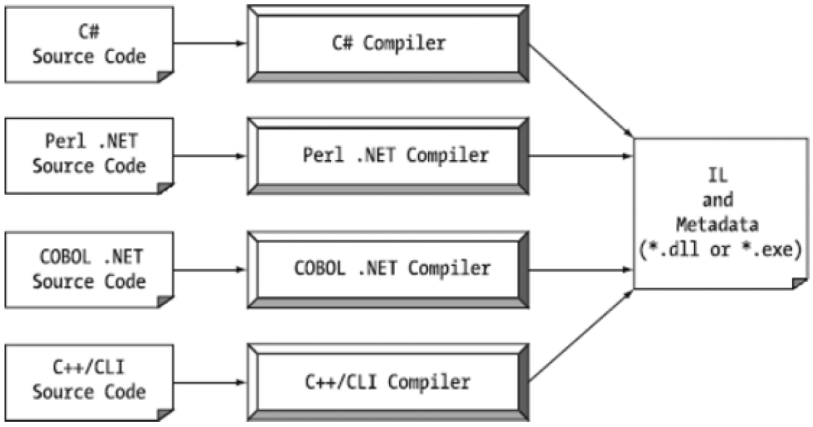
- Nutzung von Vorteilen der unterschiedlichen Sprachen

Wie tauschen sich die unterschiedlichen Programmiersprachen untereinander aus?

- über CLI

1.1.3 COMPILIERUNG IN .NET

- Erzeugt *.dll Bibliotheken und *.exe Dateien → heißen Assembly
- Diese Dateien haben bis auf ihre Endung nichts mit normalen .exe, .dll Dateien zu tun
- Dateien enthalten Code in einer Intermediate Language (IL) + Metadaten + Manifest
- Enthält keine Plattform-spezifischen Informationen → plattformunabhängig
- Metadaten beschreiben Typen, bspw. von wem eine Klasse erbt, welche Methoden sie hat, Version, ...



1.1.3.1 C# CODE

```
class Calc{
  public int Add(int x, int y){
    return x + y;
  }
}
```

1.1.3.2 VISUAL BASIC CODE

```
Class Calc
  Public Function Add(ByVal x As Integer, ByVal y As Integer) As
     Integer
    Return x + y
  End Function
End Class
```

1.1.3.3 ENTSPRECHENDER IL CODE

https://en.wikipedia.org/wiki/List_of_CIL_instructions

```
.method public hidebysig instance int32 Add(int32 x, int32 y) cil
   managed{
// Code size 9 (0x9)
```

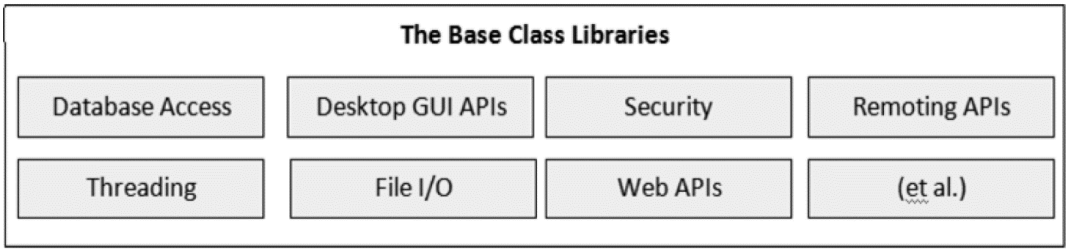
```
.maxstack 2
.lacals init (int32 V_0)
IL_0000:  nop
IL_0001:  ldarg.1
IL_0002:  ldarg.2
IL_0003:  add
IL_0004:  stloc.0
IL_0005:  br.s IL_0007
IL_0007:  ldloc.0
IL_0008:  ret
} //end of method Calc::Add
```

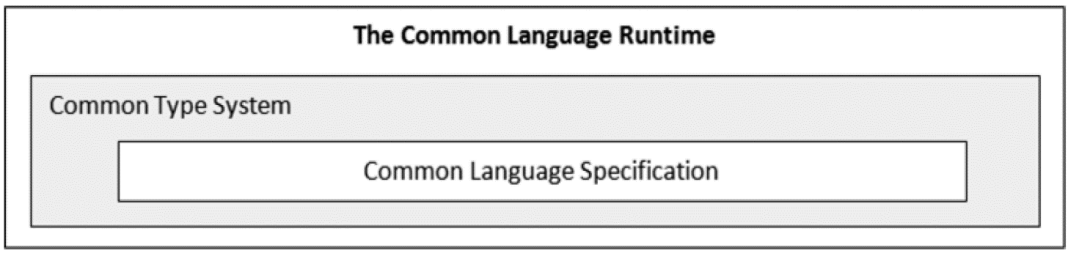
1.1.4 AUSFÜHRUNG DES PROGRAMMS

- ruft just-in time compiler (JIT, jitter) auf der den code compiliert und ausführt
- Methoden werden nur einmal kompiliert und im Speicher gelagert, ($\rightarrow$ kein Interpreter) so dass mehrfacher Aufruf der gleichen Methode nur eine einzige Kompilierung benötigt.

1.1.5 TEILE VON .NET

- Common Language Runtime (CLR)
  - Findet, lädt und managed .Net Objekte
  - Nutzt Metadaten der Assembly um Typen zu finden
  - Memory management, thread coordination, ...
  - Compiliert nötigen IL Code und führt ihn aus
- Common Type System (CTS)
  - Spezifiziert Datentypen und Programmkonstrukte und deren Interaktion
- Common Language Specification (CLS)
  - Definiert eine Teilmenge von CTS, die über alle .NET Sprachen geteilt ist
  - Denn nicht jede .NET Sprache unterstützt alle features und Datentypen der CTS
  - Programmierung nur mit CLS features erzeugt Code Bibliotheken die garantiert mit jeder anderen .NET Sprache kommunizieren können
  - Festgelegt in Regeln
    - Erste Regel: Alle Regeln beziehen sich nur auf die Teile eines Typs, der außerhalb eines Assemblys sichtbar sind
  - C# Compiler kann prüfen, ob die Regeln eingehalten wurden, also ob der Code der CLS entspricht
- Base Class Library
  - Existiert für jede .NET Sprache
  - Beinhaltet Behandlung von I/O, Threads, Grafik, Hardware-Schnittstellen





Wie behält man die Übersicht bei vielen verschiedenen Klassen? → Namespaces und/oder Vererbung

1.1.6 NAMESPACES

- Zur Organisation aller Typen der .NET Platform über alle Sprachen hinweg
- Namespaces können subspaces (Unterräume) enthalten
  - System, System.IO, ...
- Jedes Assembly kann beliebig viele Namespaces enthalten
- Bsp: System namespace enthält System.Int32, System.String, ...
- Zugriff auf Typen durch
  - `Namespace.Subnamespace.Subnamespace.Typname`
  - using keyword: `using Namespace.Subnamespace.Subnamespace Typname`

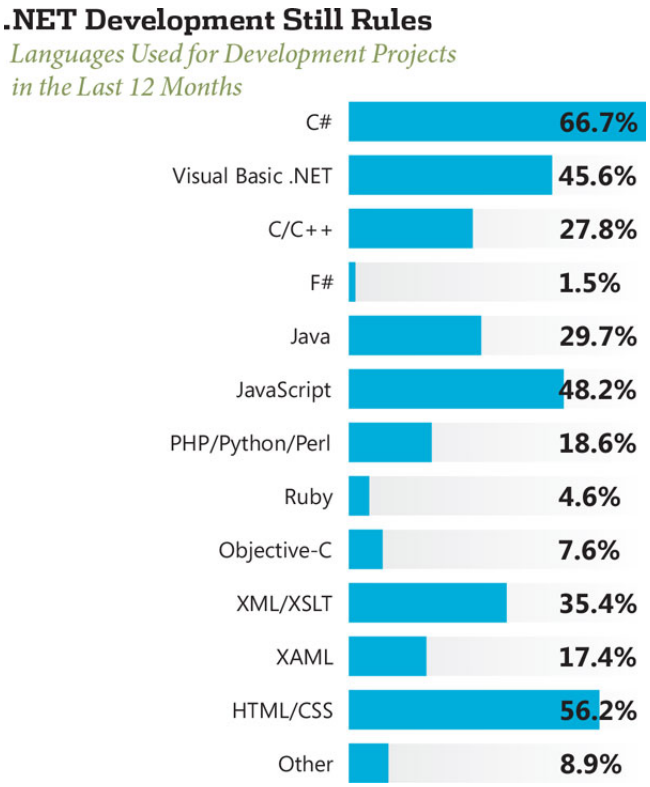
1.2 DIE SPRACHE C#

- Syntax sehr ähnlich zu Java
- Vereint Konzepte von Java, Visual Basic, C++
- Idee: die Vorteile aller Sprachen vereinigen
  - Syntaktisch sauber wie Java
  - Einfach wie Visual Basic
  - Flexibel und Mächtig wie C++

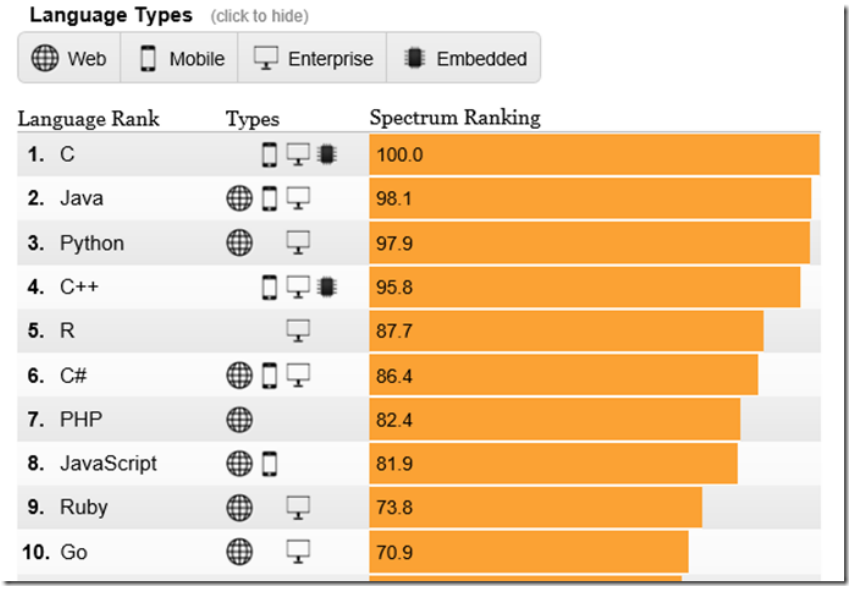
1.2.1 FEATURES

- Keine Pointer nötig
- Automatisches Speichermanagement (Garbage Collector)

- Operatorüberladung
- Attribut-basierte Programmierung
- Generische Programmierung
- Lambda Operatoren
- Einfaches Multithreading mit Async/Await Keywords



MOST POPULAR CODING LANGUAGES



1.2.2 .NET CORE

.NET Core ist eine freie und quelloffene Software-Plattform (innerhalb) der .NET-Plattform

- 2016 veröffentlicht
- Plattform-unabhängig!
- Abgespeckte Version von .NET
- Fast alles was wir hier lernen kann direkt in .NET Core verwendet werden

1.3 GLIEDERUNG DES KURSES

- Grundlagen C#
- Vererbung, Polymorphie, Schnittstellen
- Generische Programmierung
- Fehlerbehandlung Exceptions
- Delegates und Ereignisse
- Windows Forms
- Parallele Programmierung
- Test-basierte Entwicklung
- Agile / emergente Entwicklung

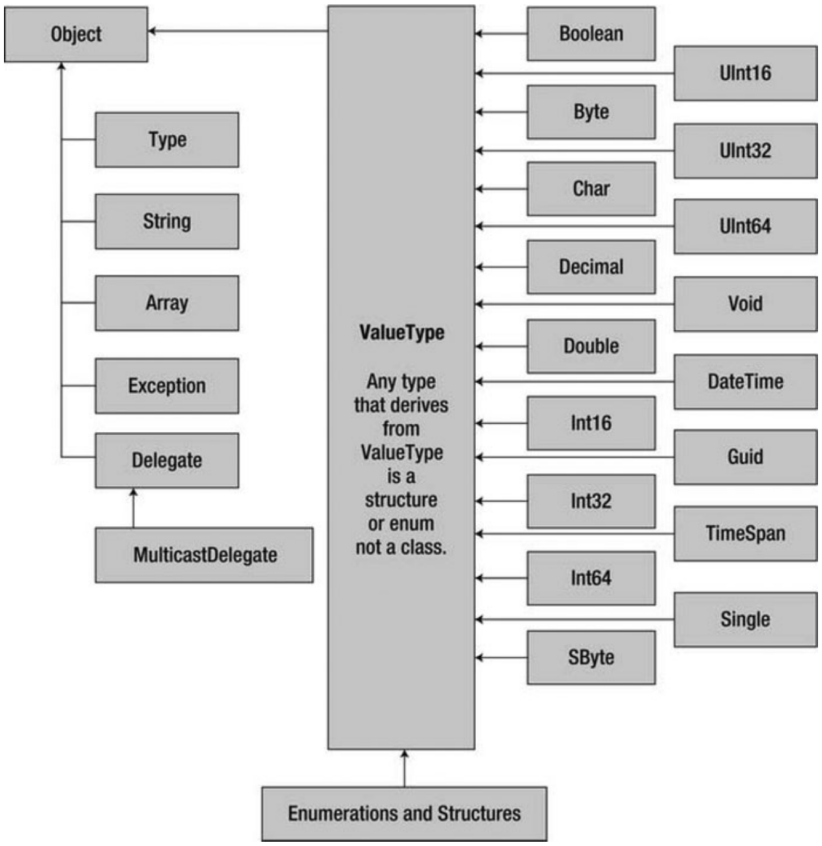
2 GRUNDLAGEN C#

```
// The topmost class in the .NET univeres: System.Object
namespace System{
  public class Object{
    public Object();  // Konstruktor, wird auch von Konstruktoren abgeleiteter Klassen gerufen
    public virtual Boolean Equals(Object obj);  // aktuelles Objekt und Objekt gleich?
    public static Boolean Equals(Object o1, Object o2); // Objekte o1 und o2 gleich?
    public virtual int GetHashCode(); // Eindeutiger Wert, override seitens abgeleiteter Klasse
    public Type GetType();  // liefert exakten Typ des Objektes zur Laufzeit
    pubilc virtual String ToString(); // gibt String zurück, der das aktuelle Objekt darstellt
    protected virtual void Finalize();  // gibt einem Objekt Gelegenheit zum Versuch, Ressourcen freizugeben, bevor Objekt vom Garbage Collector freigegeben wird
    protected Object MemberwiseClone(); // erstellt eine flache Kopie des aktuellen Objektes
    public static bool ReferenceEquals(object objA, object objB); // stellt fest, ob die angegebenen Objekte identischen Speicherplatz belegen, d.h. nicht nur gleiche Werte besitzen. Aus ReferenceEquals(o1,o2) == true folgt immer Equals(o1, o2) == true bzw. o1.Equals(o2) == true. Umgekehrt gilt die Aussage jedoch nicht.
  }
}
```

2.1 INTRINSISCHE DATENTYPEN

C# Shorthand	CLS Compliant?	System Type	Range	Meaning
`bool`	yes	`System.Boolean`	`true` oder `false`	logical value
`sbyte`	no	`System.SByte`	$-128 \ldots 127$	signed 8-bit number
`byte`	yes	`System.Byte`	$0 \ldots 255$	unsigned 8-bit number
`short`	yes	`System.Int16`	$-32\,768 \ldots 32\,767$	signed 16-bit number
`ushort`	no	`System.UInt16`	$0 \ldots 65\,535$	unsigned 16-bit number
`int`	yes	`System.Int32`	$-2\,147\,483\,648 \ldots 2\,147\,483\,647$	signed 32-bit number
`uint`	no	`System.UInt32`	$0 \ldots 4\,294\,967\,295$	unsigned 32-bit number

`long`	yes	`System.Int64`	$-9\ 223\ 372\ 036\ 854\ 775\ 808$ ... $9\ 223\ 372\ 036\ 854\ 775\ 807$	signed 64-bit number
`ulong`	no	`System.UInt64`	0 ... $18\ 446\ 744\ 073\ \ 709\ 551\ 615$	unsigned 64-bit number
`char`	yes	`System.Char`	`U+0000` ... `U+ffff`	single 16-bit Unicode character
`float`	yes	`System.Single`	$-3,4 \cdot 10^{38}$ ... $+3,4 \cdot 10^{38}$	32-bit floating-point number
`double`	yes	`System.Double`	$\pm 5,0 \cdot 10^{-324}$ ... $\pm 1,7 \cdot 10^{308}$	64-bit floating-point number
`decimal`	yes	`System.Decimal`	$(-7,0 \times 10^{28}$ ... $7,9 \times 10^{28})$ / $10^{0 \cdots 28}$	
`string`	yes	`System.String`	Limited by system memory	Represents a set of Unicode characters
`Object`	yes	`System.Object`	Can store any data type in an object variable	The base class of all types in the .NET universe



- Object(s) hat Referenztypen Heap
- ValueType(s) hat Wertetypen Stack

BEISPIEL

```
static class A{
  static double d=0;
}

```

```
static void MyMethod (double d){
  int k=0;
  A a = new A();
}

Main(...){
  double d=0;
  MyMethod();
}
```

Die in dem Code befindlichen Daten werden im Stack/Heap abgespeichert:

Stack

Stackframe: MyMethod d: 00000000 (aus Stackframe Main) k: 00000000 a: Referenz auf Heap
Stackframe: Main d: 00000000

Heap

A: d: 00000000 ⋮

Beim Beenden der Methode `MyMethod()`(Scope) wird der entsprechende Stack gelöscht. Der Heap wird erst vom Garbage Collector gelöscht, wenn die Einträge nicht mehr in Verwendung sind.

AM BEISPIEL:

```
static public void setToOne(int a, ref int b, int[] c, int[] d){
  a = 1;
  b = 1;
  c[0] = 1;
  d = new int[1];
  d[0] = 1;
}

static void Main(string[] args){
  int a = 0;
  int b = 0;
  int[] c = new int[1];
  c[0] = 0;
  int[] d = new int[1];
  d[0] = 0;
  setToOne(a, ref b, c, d);
  Console.WriteLine("{0} {1} {2} {3}", a, b, c[0], d[0]);
}
```

Ergebnis ist 0 1 1 0:

- a wird nicht überschrieben, ist nur lokal
- b wird überschrieben, da Referenz übergeben wurde
- c wird überschrieben, da das Array eine Referenz (kein ValueType) ist (ähnlich wie in C)

- d wird nicht überschrieben, da die Variable neu initialisiert wird

2.2 BOXING / UNBOXING

BOXING: Konvertierung (Einschachteln) eines Wertetyps in ein object (implizit)
UNBOXING: Extraktion eines Wertetyps aus einem object (explizit)

BEISPIEL BOXING

Boxing:

```
Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
Console.WriteLine("12.ToString() = {0}", 12.ToString());
Console.WriteLine("12.GetType() = {0}", 12.GetType());
```

Ausgabe:

```
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Unboxing:

```
object o = 123;
int i = (int) o;
```

2.2.1 MEMBER DER INTRINSISCHEN DATENTYPEN

```
int.MaxValue  // liefert größten int
int.MinValue  // liefert kleinsten int
double.Epsilon  // liefert kleinste darstellbare Zahl

char myChar = {}'a';
char.isDigit(myChar)  // liefert True wenn myChar eine Zahl ist
char.isLetter(myChar) // liefert True wenn myChar eine Zahl ist
char.isWhiteSpace({}'Hallo du',5) // steht an 5-ter Stelle ein
   Leerzeichen?

// Konvertierung von Strings:
bool b = bool.Parse(inputString); // liefert True wenn inputString
   = "True"
double d = double.Parse(inputString); // convertiert inputString in
    double
int i = int.Parse(inputString); // convertiert inputString in int
```

2.2.2 ARBEITEN MIT STRINGS

String Member	Meaning

`Length`	This property returns the length of the current string.
`Compare()`	This static method compares two strings.
`Contains()`	This method determines whether a string contains a specific substring.
`Equals()`	This method tests whether two string objects contain identical character data.
`Format()`	This static method formats a string using other primitives (e.g., numerical data, other strings) and the `{0}` notation.
`Insert()`	This method inserts a string within a given string.
`PadLeft()` `PadRight()`	These methods ar used to pad a string with some characters
`Remove()` `Replace()`	Use these methods to receive a copy of a string with modifications (characters removed or replaced)
`Split()`	This method returns a String array containing the substrings in this instance that are delimited by elements of a specified char array or string array.
`Trim()`	This method removes all occurrences of a set of specified characters from the beginning and end of the current string.
`ToUpper()` `ToLower()`	These methods create a copy of the current string in uppercase or lowercase format, respectively.

BEISPIEL

```
string s = Console.ReadLine();  // z.B. "Hallo Welt"
s = s.Replace("Hallo", "Tschüss");
Console.WriteLine("Modified string: \{0\}", s); // Tschüss Welt
s = s + " ..."; // string concatenation
Console.WriteLine("Modified string: \{0\}", s); // Tschüss Welt ...
```

2.2.3 TEXTFORMATIERUNG

`Console.WriteLine("i = {0:F8}", i);`

ALLGEMEIN

`{index[,breite][::formatierung]}`

- `index`:
  Laufende Nummer des Ausdrucks (von 0 an 0(1)...)
- `breite`:
  Anzahl Zeichen für Ergebnis der Formatierung:
  - > 0: Rechtsbündig
  - < 0: Linksbündig

  wenn fehlt: kürzeste Ganzzahl (Länge)

- `formatierung`:

Zeichen	Beschreibung	Defaultergebnis
`C,c`	Währung	€, $ xx,xx.xx
`D,d`	dezimal	[-] xxxx
`E,e`	exponentiell	[-] x,xxxxE±xxx
`F,f`	Festkomma	[-] xxxx,xx
`G,g[E,e]`	exponentiell	variabel
`N,n`	Nummer mit Trennzeichen	[-] xx,xxx,xx
`X,x`	hexa, ganz	variabel

2.2.4 ESCAPE CHARACTERS

Character	Meaning
`\'`	Inserts a single quote into a string literal.
`\"`	Inserts a double quote into a string literal.
`\\`	Inserst a backslash into a string literal. This can be quite helpful when defining a file or network paths.
`\a`	Triggers a system alert (beep). For console programs, this can be an audio clue to the user.
`\n`	Inserts a new line (on Windows platforms).
`\r`	Inserts a carriage return.
`\t`	Inserts a horizontal tab into the string literal.

2.2.5 WIDENING / NARROWING

Im folgenden Code nimmt die Add Methode nur Argumente vom Typ int, bekommt aber short übergeben. Was passiert? Welcher Teil des Codes könnte Probleme geben?

```
public static void Main(string[] args){
  short a = 10, b = 10;
  short c = Add(a, b);  // Fehler: int kann keinem short zugewiesen werden! Müsste gecastet werden. Wenn dann allerdings a+b größer ist als der Wertebereich von short, würde der Overflow eine negative Zahl erzeugen.
  Console.WriteLine("{0} + {1} = {2}", a, b, c);
}

public static int Add(int a, int b){
  return a + b;
}
```

2.2.6 SCHLÜSSELWORT VAR

Typen werden automatisch aus der anfänglichen Zuweisungen geschlussfolgert.
→ implizite Typisierung

BEISPIEL

```
var myInt = 0;
var myBool = true;
var myString = "Time, marches on ... ";
Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
Console.WriteLine("myString is a: {0}", myString.GetType().Name);
```

2.3 SCHLEIFEN

Wie in C# gibt es die folgenden Schleifen:

- `do {...} while (...)`
- `while (...){...}`
- `for (...; ...; ...){...}`

Zusätzlich gibt es die foreach Schleife:

- `foreach` (Typ element in Menge)
- jedes Element einer Menge wird genau 1 mal durchlaufen
- Die Menge kann alles sein was das Ienumerable Interface implementiert (Array, Liste, ...)
- entspricht einem Iterator in C++
- durchlaufene Elemente können nicht verändert werden
- Beispiel:

```
string[] carTypes = {"Ford", "BMW", "Trabbi", "Honda"};
foreach (string c in carTypes)
Console.WriteLine(c);
```

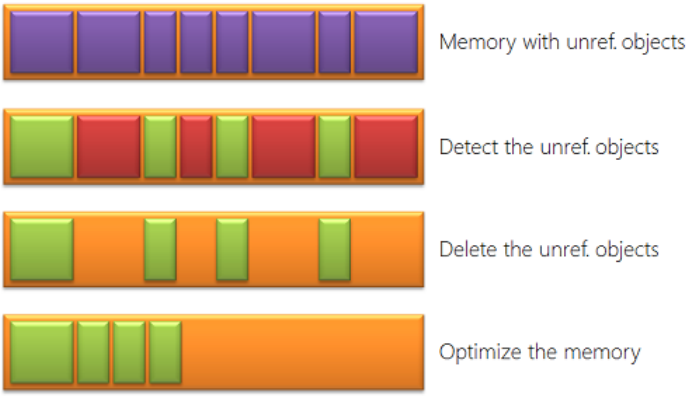
2.4 GARBAGE COLLECTOR

- Objekte benötigen Speicher im Heap (Speicherallokation mit new)
- werden Objekte nicht mehr benötigt sollte der Speicher wieder freigegeben werden
- Bsp:

```
Circle kreis1 = new Circle();
Circle kreis2 = new Circle();
[...]
kreis1 = null2;
```

- Speicherfreigabe ist in C# nicht explizit möglich, sondern geschieht durch den Garbage Collector

FUNKTIONSWEISE



Garbage Collector wird ausgelöst:

1. wenn Programm wenig ausgelastet
2. wenn Speicherressurcen knapp
3. bei Programmende
4. manuell mit `GC.collect();`

2.5 OPERATORÜBERLADUNG

Beispiel: a+b ist nun a-b:

```
public static int operator +(int a, int b){
  return a-b;
}
```

3 METHODEN UND PARAMETER MODIFIZIERER

```
public int myFunction(int a, out string s, ref double d, params double[] d);
```

3.1 AUSGABEPARAMETER (OUT)

- müssen in der Methode zugewiesen werden
- müssen nicht vor Aufruf der Methode initialisiert werden
- pass-by-reference

BEISPIEL

```
public static int myFunction(out double d){
  d = 1;
  return 0;
}

// Aufruf:
double d;
int i = myFunction(out d);
// d=1
```

3.2 REFERENZPARAMETER (REF)

- Wert-Änderungen ändern den Parameter auch nach Abarbeitung der Methode
- pass-by-reference

BEISPIEL

```
public static void myFunction(ref double d){
  // d=0
  d = 1;
}

// Aufruf:
double d = 0.0;
int i = myFunction(ref d);
// d=1
```

3.3 BELIEBIG VIELE PARAMETER (PARAMS)

- Erlaubt beliebig viele Parameter des gleichen Typs zu übergeben ohne dass explizit ein Array erzeugt werden muss
- nur ein einzelner params Modifizierer pro Methode, stets an letzter Stelle der Argumentliste

BEISPIEL

```
public static double add(params double[] d){
  double sum = 0;
  for (int i=0; i<dd.Length; i++) sum += d[i];
  return sum;
}

// Aufruf:
double summe = add(0.1, 0.2, 0.3);
```

3.4 PASS-BY-VALUE

- Wert der übergebenen Variable wird in eine neue lokale Variable kopiert

BEISPIEL

```
public static void myFunction(double d){
  d = 1;
}

// Aufruf:
double d = 0;
int i = myFunction(d);
// still d = 0
```

3.5 VERANSCHAULICHUNG

Wie viele Fehler finden Sie (angenommen die Signatur der Funktion ist korrekt)?

```
public static int myFunction(int a, out int o, ref int r){
  // o muss erst ein Wert zugewiesen werden!
  int result = a+o+r;
  return result;
}

static void Main(string[] args) {
  double a;
  myFunction(a, 1, 2);
  // a: nicht korrekt, muss erst initialisiert werden und ist
     falscher Typ (double)
```

```
11    // b: nicht möglich, muss eine Variable sein und mit "out" ü
         bergeben werden
12    // 2: nicht möglich, muss eine Adresse/Referenz zu einer
         Variablen sein und mit "ref" übergeben werden
13 }
```

4 ARRAY / STRUCT / ENUM

4.1 ENUMERATION (AUFZÄHLUNG)

Vorlesung
31.03.2017

BEISPIEL

```
public enum Karte{
  Karo = 9,
  Herz = 10,
  Pik = 11,
  Kreuz = 12
}
```

4.1.1 EIGENSCHAFTEN

- gruppiert Konstanten die thematisch verwandt sind
- Konstanten sind (wenn nicht anders angegeben) vom Typ int
- ohne Angabe von Werten werden automatisch Werte vergeben (0, . . . , n)
- ein Objekt von Typ enum hat genau einen Wert

```
Karte card1;
card1 = Karte.Herz;
if (card1 == Karte.Herz)
  Console.WriteLine("Sie haben Herz.");
```

4.1.2 EINEN ENUM DURCHLAUFEN

```
foreach (Karte karte in Enum.GetValues(typeof(Karte)))
  Console.WriteLine(karte);
```

4.2 STRUCT

- ähnlich wie class, aber Wertetyp (auf dem Stack abgelegt)
- Unterschiede zu class:
  - kann nicht abgeleitet werden (erbt nur von ValueType)
  - besitzt immer parameterlosen Konstruktor
  - Speicherallokation mit new nicht nötig, wenn auf Initialisierung der Felder des struct verzichtet werden kann

Beispiele: alle primitiven Datentypen

4.3 ARRAY

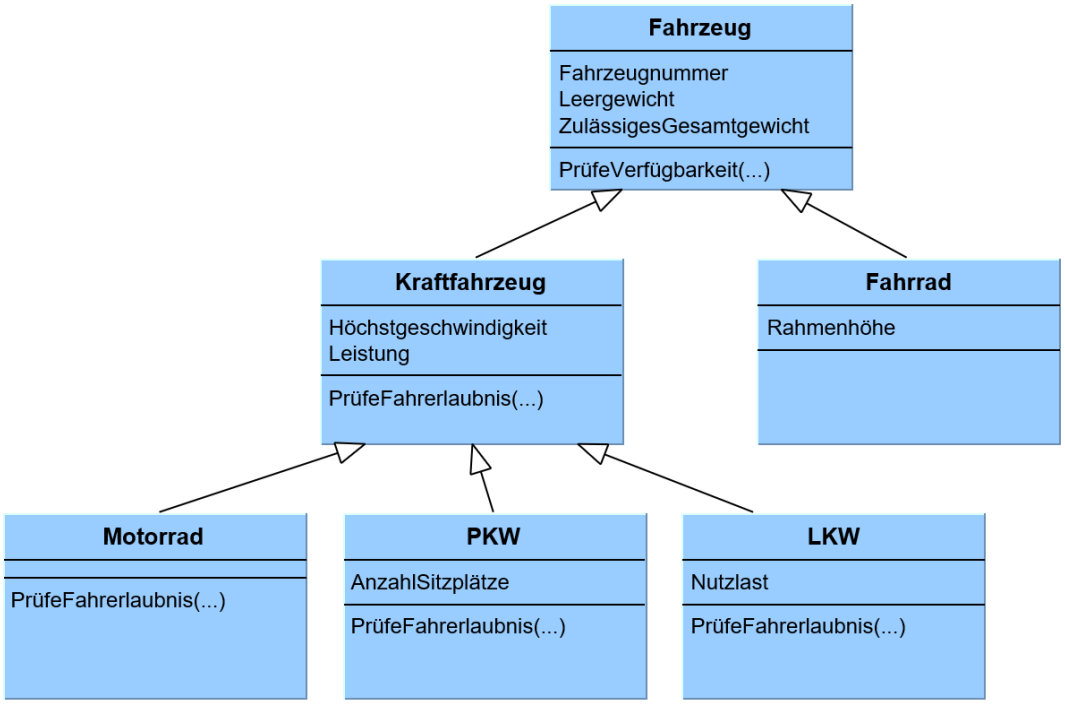
Arrays erben direkt von `System.Object`, sind also Referenztypen!

```
int[] elements = new int[3];
elements = {23, 9, 7};

// mehrdimensionale Arrays:
int[,] points = new int[4,3];
int[,] points = {{1,2,3},{4,5,6}};

// verzweigte Arrays:
int[][] myArray = new int[3][];
myArray[0] = new int[3];
myArray[1] = new int[4];
myArray[2] = new int[2];
```

5 OBJEKT-ORIENTIERUNG IN C#

5.1 VERERBUNG

Alle Member der Basisklasse werden quasi vom Compiler in die Subklasse kopiert.



- In C# nur einfach Vererbung (genau eine Mutterklasse)
- dafür beliebige Anzahl von Schnittstellen
- BASE-Schlüsselwort zum Aufrufen der überschriebenen Basisklassenmember von abgeleiteten Klassen
- SEALED-Schlüsselwort verhindert Vererbung von dieser Klasse

BEISPIEL

```
class Fahrzeug{
  int Fahrzeugnummer;
  ...
}

class Fahrrad : Fahrzeug {
  int Rahmenhöhe;
  int Fahrzeugnummer;
```

```
  // Zugriff auf Fahrzeugnummer Fahrzeugnummer der Oberklasse bspw.: base.Fahrzeugnummer
}

Fahrzeug[] meineFahrzeuge = new Fahrzeug[3];
meineFahrzeuge[0] = new Fahrrad();  // möglich, da Fahrrad Fahrzeug ist
meineFahrzeuge[1] = new Fahrrad();
meineFahrzeuge[2] = new PKW();  // auch möglich, da PKW auch Fahrzeug ist
```

5.2 DER KONSTRUKTOR

- spezielle Methode die automatisch bei Speicherallokation mit NEW aufgerufen wird
- kann nicht explizit aufgerufen werden
- wenn kein Konstruktor angegeben wird ein Standard-Konstruktor zur Verfügung gestellt
- spezielle Syntax: kein Rückgabewert, Name = Klassenname

- sind mehrere Konstruktoren vorhanden kann man diese nutzen mit THIS
- auf Konstruktoren der Mutterklasse kann mit BASE zugegriffen werden

Beispiel Was tun die 3 Konstruktoren von Employee?

```
class Person {
  public string name;
  public Person(string name) {
    // perform some checks here
    this.name = name;
  }
}

class Employee : Person {
  public int id;
  public Employee() : this("unbekannt", 0) { }  // ruft den 3. Konstruktor mit dem Namen "unbekannt" und ID 0 auf.
  public Employee(int id) : this("unbekannt", id) { }   // ruft den 3. Konstruktor mit dem Namen "unbekannt" und der übergebenen ID auf
  public Employee(string name, int id) : base(name) {
    // perform some checks here
    this.id = id;
  } // ruft den Konstruktor des Parents mit dem übergebenen Namen auf (der den Namen zuweist) und weist dann die übergebene ID zu
}
```

5.3 POLYMORPHIE

Auf einer Referenz der Basisklasse kann die spezifische Methode der Sub-Klasse aufgerufen werden.
Funktioniert in C# durch die Schlüsselwörter `virtual` und `override`.

```
class Basisklasse {
  public virtual void test() {
    Console.WriteLine("Basis");
  }
}

class Subklasse : Basisklasse {
  public override void test() {
    Console.WriteLine("Sub");
  }
}

// Aufruf:
Basisklasse b = new Subklasse;
b.test(); // Ausgabe: Sub
```

5.4 SCHLÜSSELWORT-OPTIONEN FÜR KLASSENMETHODEN

- VIRTUAL: Die Methode kann Inhalt enthalten und kann in einer Abgeleiteten Klasse polymorph überschrieben werden
- OVERRIDE: Muss bei einer überschriebenen Methode angegeben werden um eine virtual Methode der Basisklasse zu überschreiben
- SEALED: Eine überschriebene virtuelle Methode darf in Tochterklassen nicht weiter überschrieben werden
- ABSTRACT: Die Methode ist leer und muss überschrieben werden
- NEW: Wenn die zu überschreibende Methode nicht virtual in der Basisklasse ist kann override nicht verwendet werden → new nötig, Polymorphismus funktioniert dann nicht
- STATIC: Methode muss von der Klasse aus aufgerufen werden, nicht von einer Instanz
- CONST: Der Rückgabewert kann nicht verändert werden (Aufruf function()=1 geht nicht)

BEISPIEL

```
class A {
  public virtual void test() {
    Console.WriteLine("A");
  }
}

class B : A {
  public sealed override void test() {
```

```
    Console.WriteLine("B");
  }
}

class C : B {
  public override void test() { // führt zu Fehler, da test() von B "sealed" ist!
    Console.WriteLine("C");
  }
}
```

5.5 OPTIONEN FÜR KLASSENDEKLARATION

- STATIC: Klasse darf nur statische Member enthalten und kann nicht instanziiert werden
- ABSTRACT: wenn eine abstracte Methode enthalten ist muss die Klasse als abstract markiert werden, verhindert Instanziierung

5.6 OPTIONEN FÜR VARIABLEN

- STATIC: die Variable oder Klasse existiert einmalig für die Funktion oder Klasse und nicht für eine Instanz)
- CONST: (die Variable ändert sich nicht)
  - const vor Parametern: Die Parameter können nicht verändert werden
  - const vor Membervariable: Variablenwert muss sofort zugewiesen werden, Variable ist dann automatisch auch static, beide Einschränkungen sind aufgehoben wenn READONLY statt CONST verwendet wird

5.7 ZUGRIFFSMODIFIZIERER FÜR SICHTBARKEIT

	Überall	Bibliothek	abgeleitete Klassen	
private				→ Standard in einem Typ (z.B. in einer Klasse)
protected			X	
internal		X	X	
public	X	X	X	

5.8 CASTING VON KLASSEN

- implicit cast in eine Basisklasse:

```
object peter = new Person("Peter");
```

- explicit cast in Subklasse:

```
Person p = (Person) peter; // braucht cast, da peter erst mal nur object ist.
```

- AS keyword:

```
string s = peter as string; // "as-cast" peter auf string möglich, führt nicht zu Laufzeitfehler
if (s == null) // wenn peter kein string ist, bleibt s null
  Console.WriteLine("Peter ist kein String");
```

- IS keyword:

```
if (peter is string)  // ähnlich wie "peter as string" mit Abfrage, ob null
  Console.WriteLine("Peter ist kein String");
```

BEISPIEL

Vorlesung 07.04.2017

```
class MyClass{
  public virtual void Write(string text){
    Console.Write(" du bis, {0}", text);
  }
}
class MySubClass : MyClass{
  public override void Write(string text){
    Console.Write("{0} Fisch", text);
  }
}

public static void Main(string[] args){
  MyClass a = new MyClass();
  MyClass b = new MySubClass();
  b.Write("Egal wie");
  a.Write("Helene ist");
  ((MyClass) b).Write("");
  Console.Write("er:");
  // Text bis hier: "Egal wie Fisch du bist, Helene ist Fischer"
  ((MySubClass) a).Write("Ende"); // funktioniert nicht! Programm stürzt ab.
}
```

5.9 PROPERTIES (EIGENSCHAFT)

Property = Member-Variable die in einem Objekt definiert ist und auf die von außen zugegriffen werden soll.

Zugriff auf Eigenschaften:

C++ Style:

```
private int myVariable;
public int GetMyVariable() {
  return myVariable;
}
private void SetMyVariable(int
    value) {
  myVariable = value;
}
public void AccessVariables(int
    a) {
  int b = GetMyVariable();
      SetMyVariable(a);
}
```

C# Style:

```
private int myVariable;

public int MyVariable{
  get {
    return myVariable;
  }
  private set {
    myVariable = value;
  }
}

int b = MyVariable;
MyVariable = a;
```

- Schreibaufwand reduziert
- Direkter Zugriff im Aufrufer

5.9.1 AUTO-GENERATED PROPERTIES

Will man nur den schreibenden Zugriff auf eine Variable einschränken, kann man noch kürzer schreiben:

```
public int MyVariable {get; private set;}
// Variable myVariable wird intern angelegt.
```

- Dabei wird die zugrundeliegende Variable (vorher `myVariable`) nicht mehr benutzt
- Intern wird eine eigene Variable angelegt auf die nun kein direkter Zugriff mehr möglich ist

5.10 OBJECT INITIALIZATION SYNTAX

Objekte / Felder einer Klasseninstanz können direkt zugewiesen werden, ohne benutzerdefinierten Konstruktor.

```
class Person {
  public string Nachname { get; set;}
  public string Vorname { get; set;}
  public Person() {}
}

...

Person p1 = new Person() {Nachname = "Hook"};
Person p2 = new Person() {Vorname = "Wendy"};
Person p3 = new Person() {Vorname = "Peter", Nachname =  "Pan"};
```

5.11 ANONYME OBJEKTE

- Klasse ohne Bezeichner-Namen
- zur schnellen Gruppierung von Elementen
- Der Typ dieser Objekte ist kompliziert → var keyword benutzen

```
var anonymous = new { Name = "Florian", Alter = 28 };
Console.WriteLine(anonymous.Name);
```

6 INTERFACES (SCHNITTSTELLEN)

- Vertragsvereinbarung für Klassen, welche Pflichten sie erfüllen müssen (welche Member sie implementieren müssen)
- Klassen dürfen in C# nur von einer Basis-Klasse erben, aber beliebig viele Schnittstellen implementieren
- Schnittstellen sind wie abstrakte Klassen, definieren nur Member aber spezifizieren keine Werte / Implementierung
- erlaubte Member:
  - Methoden
  - Properties
  - Events

BEISPIEL

```
public interface ICopy {
  string Caption {get; set;}
  void Copy();
}

// in der Klasse, die das Interface implementiert müssen alle(!)
   Methoden implementiert sein
class Document : BaseDocument, ICopy {
  public void Copy() {
    // ...
  }
  public string Caption {
   get{ /* ... */ }
   set{ /* ... */ }
  }
  // ...
}
```

6.1 NUTZEN: ERSATZ FÜR FEHLENDE MEHRFACHVERERBUNG

BEISPIEL

- Sie schreiben ein Grafikprogramm und haben viele Objekte die alle eine Methode `Draw()` implementieren sollen, die das Objekt auf dem Bildschirm zeichnet.
- Sie haben alle instanziierten Objekte in einem Array gespeichert
- Dieses Array soll durchlaufen werden um alle Objekte zu zeichnen

Wie muss der folgende Code ergänzt werden, damit man ein Array der mal-baren Objekte durchlaufen und dabei `Draw()` ausführen kann?

```
public interface IDrawable {
  void Draw();
}

public class Rectangle : GeometricObject, IDrawable {
  public void Draw()  {  /* ... */  }
}

public class Cylinder : GeometricObject, IDrawable {
  public void Draw()  {  /* ... */  }
}

public class Transporter : Car, IDrawable {
  public void Draw()  {  /* ... */  }
}

IDrawable [] graphicObjects = new IDrawable[3];
graphicObjects[0] = new Rectangle();
graphicObjects[1] = new Cylinder();
graphicObjects[2] = new Transporter();
foreach (IDrawable item in graphicObjects)
  item.Draw();
```

7 GENERIK

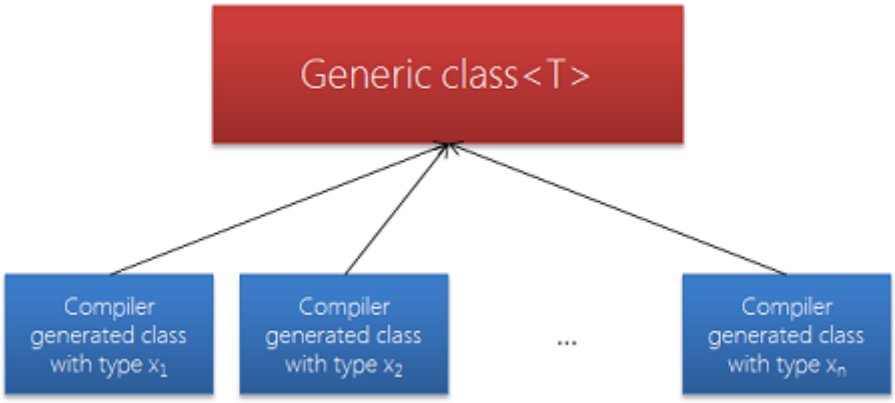
- Oft müssen ähnliche Operationen / Funktionalitäten für unterschiedliche Datentypen bereitgestellt werden
- Um nicht gleichen Code mehrfach zu schreiben nutzt man Platzhalter (template) für Datentypen
- Beispiel: List<T>

7.1 GENERISCHE KLASSE DEFINIEREN

```
class GenericClass<T> {
  T myVariable;
  public GenericClass(T myVar){
    myVariable = myVar;
  }
  public T MyVariable{
    get {
      return myVariable;
    }
  }
}
```

7.2 GENERISCHE KLASSE BENUTZEN

```
GenericClass<int> a = new GenericClass<int>(5);
```



Ähnlich für:

- struct

- interfaces
- delegates

7.3 GENERISCHE METHODEN

Generik funktioniert genauso für Methoden

```
class MyClass {
  public static void Swap<T>(ref T l, ref T r) {
  // auch möglich mit mehreren Generiks:
  // public static void Swap<T1, T2>(ref T1 l, ref T2 r) {
    T temp = r;
    r = l;
    l = temp;
  }
}

int a = 3;
int b = 4;
MyClass.Swap(ref a, ref b);
```

7.4 TEMPLATE CONSTRAINTS

Constraints = Vorgaben an die Templates

7.4.1 INTERFACE CONSTRAINT

```
public static void Swap<T>(ref T l, ref T r) where T : IComparable{
  if (l.CompareTo(r) < 0)  {
    T temp = r;
    r = l;
    l = temp;
  }
}
```

7.4.2 KLASSEN CONSTRAINT

```
class MyGenericClass<T> where T : IComparable{

}
```

7.4.3 VERERBUNGS CONSTRAINTS

```
class MyGenericClass<T> where T : Random

class MyGenericClass<T, V> where T : List<V>

class MyGenericClass<T, V> where T : List<V> where V : Random
```

7.4.4 INSTANZIERBARKEITS CONSTRAINT

```
T CreateNew<T>() where T : new() {
  return new T(); // geht nur wegen " : new()", sonst:
     Compilerfehler
}
```

7.5 BEISPIEL

7.5.1 AUFGABE

Zur Verwaltung der Fahrzeuge eines Konzerns existiert die Klasse `Car`, die alle Daten eines Fahrzeuges enthält.

```
public class Car {
  private int gewicht;     // in kg
  public int Gewicht{
    get {return gewicht; }
    private set {gewicht = value;}
  }
  // ...
}
```

Sie möchten nun alle Fahrzeuge des Konzerns in einer Liste Sammeln. C# stellt dafür den generischen Typ `List<T>` zur Verfügung. Auf einem Objekt des Typs `List` kann die Methode `Add (T element)` aufgerufen werden um Objekte hinzuzufügen. Außerdem wollen Sie alle Fahrzeuge der Liste automatisch nach ihrem Gewicht sortieren lassen, ohne etwas an der sensiblen Klasse `Car` zu ändern. Sie finden dazu die Methode `Sort` der Klasse `List`:

```
public void Sort(IComparer<T> comparer);
```

Sie suchen nach dem Interface IComparer und finden folgende Definition:

```
public interface IComparer<in T> {
  // Returns: 1   if y>x
  //          -1  if y<x
  int Compare(T x, T y);
}
```

Was müssen sie schreiben um...

1. ... eine Liste anzulegen und die bereits erzeugten Autos `ferrari, trabbi, tesla` einzufügen?
2. ... die Liste automatisch sortieren zu lassen?

3. Was ist der Vorteil der Generik in `List<T>`? Was wäre schlecht daran, wenn die `List` nicht generisch wäre?

7.5.2 ANTWORT

1.

```
List<Car> list = new List<Car>();
list.Add(ferarri);
list.Add(trabbi);
list.Add(tesla);
```

2.

```
class Comparer : IComparer<Car> {
  int Compare(Car x, Car y){
    if (x.Gewicht < y.Gewicht)
      return 1;
    else
      return -1;
  }
}
list.Sort(new Comparer());
```

3. Die Liste weiß, was sie für einen Datentyp enthält. Sonst wäre es nur eine Liste von Objekten. Damit könnten auch „fremde“ Objekte (Tiere, Fahrräder, Strings, Zahl, ...) eingefügt werden. Diese könnten beispielsweise nicht mehr so einfach verglichen werden.

8 EXCEPTIONS

Ziel: Behandlung von Ausnahmefehlern, Vermeidung von Programmabsturz

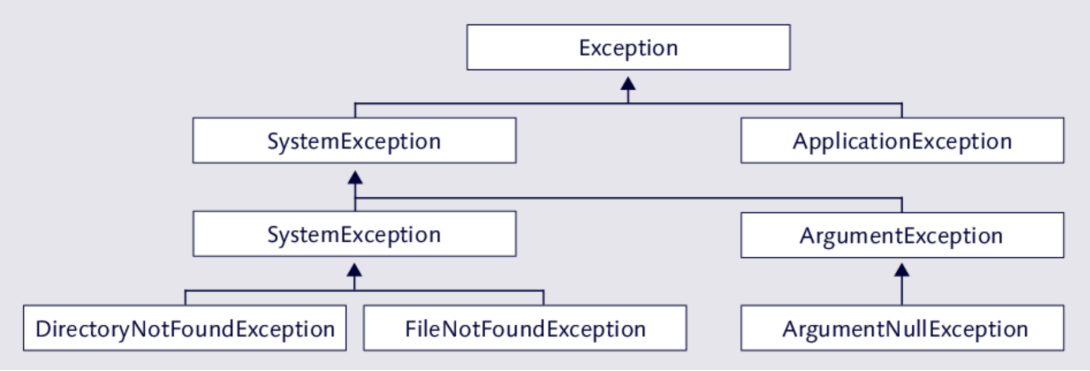
8.1 BEHANDLUNG EINER AUSNAHME MIT TRY / CATCH

Beispiel:

```
using System.IO;

class Program {
  static void Main(string[] args) {
    try { // erzeugt ggf. Ausnahme
      StreamReader stream = new StreamReader(@"C:\Text.txt");
      Console.WriteLine(stream.ReadToEnd());
      stream.Close();
    } catch (Exception e){  // behandelt Ausnahme
      Console.WriteLine(e.Message);
    }
  }
}
```

8.2 ARTEN VON EXCEPTIONS

Auszug der Vererbungshierarchie:



Mehrere Exceptions-Typen können abgefangen werden um gesondert auf sie zu reagieren

```
StreamReader stream = null;
Console.Write("Welche Datei soll geöffnet werden? ... ");
string path = Console.ReadLine();
try {
  stream = new StreamReader(path);
  Console.WriteLine(stream.ReadToEnd());
```

```
  stream.Close();
} catch (FileNotFoundException ex) {
  Console.WriteLine(ex.Message);
} catch (DirectoryNotFoundException ex) {
  Console.WriteLine(ex.Message);
} catch (ArgumentException ex) { //Pfadangabe leer
  Console.WriteLine(ex.Message);
} catch (Exception ex) {
  Console.WriteLine(ex.Message);
}
// ...
```

8.3 AUSNAHMEN WERFEN MIT THROW

Zum Test von Code kann man auch selbst ausnahmen erzeugen (werfen) `throw AbcException;`

8.4 FINALLY BLOCK

Falls nach Behandlung einer Exception etwas unbedingt ausgeführt werden muss (z.B. Freigabe einer Datei, Schließen einer Datenbankverbindung,...)

```
try {
  // ...
} catch {
  // ...
} finally {
  // ...
  // wird auch noch behandelt, wenn im catch schon ein return steht
    .
}
```

Der `finally`-Block wird immer ausgeführt unabhängig ob:

- Exception geworfen
- catch block ein return statement enthielt
- in catch noch eine Exception geworfen wurde

8.5 EIGENSCHAFTEN DER KLASSE EXCEPTION

Eigenschaft	Beschreibung
`Data`	Stellt zusätzliche Informationen zu der Ausnahme bereit.
`HelpLink`	Verweist auf eine Hilfedatei, die diese Ausnahme beschreibt.
`InnerException`	Falls bei der Behandlung einer Ausnahme eine weitere Exception ausgelöst wird, beschreibt diese Eigenschaft die neue (innere) Ausnahme.
`Message`	Liefert eine Zeichenfolge mit der Beschreibung des aktuellen Fehlers. Die Information sollte so formuliert sein, dass sie auch von einem Anwender verstanden werden kann.
`Source`	Liefert einen String zurück, der die Anwendung angibt, in der die Ausnahme ausgelöst worden ist.
`StackTrace`	Beschreibt in einer Zeichenfolge die aktuelle Aufrufreihenfolge aller Methoden.
`TargetSite`	Liefert zahlreiche Informationen zu der Methode, in der die Ausnahme ausgelöst worden ist.

Am nützlichsten sind `Message` und `StackTrace`, sie liefern beide einen String.

8.6 INNERE EXCEPTIONS

Tritt innerhalb einer Ausnahmebehandlung (catch) eine Exception auf so sollte auch diese Behandelt werden (innere Exception)

Beispiel: Was passiert hier? Was passiert wenn der (int) cast in calculateSomething weggelassen wird?

```
public static void calculateSomething() {
  double x = 1 / 2 * 2;    // 1/2 = 0, 0*2 = 0
  int y = 1 / (int) x ; // 1/0 => Exception
}

public static void Main(string[] args) {
  try{
    calculateSomething();
  } catch (Exception ex) {
    Console.WriteLine(ex.Message);
    try{
      StreamWriter file = new StreamWriter("error\_output.txt");
      file.WriteLine("Fehler beim Aufruf von calculateSomething");
      file.Close();
    } catch {
      Console.WriteLine("Fehler-Output konnte nicht geschrieben
         werden.");
    }
  }
}
```

9 DELEGATES UND EVENTS

9.1 DELEGATE

Delegate (Delegierter = Weiterleiter eines Auftrags) ist ein Typ der den Zeiger auf eine Methode beschreibt

```
public delegate double CalculateHandler(double value1, double
    value2);
```

Danach kann eine Variable vom Typ des Delegate deklariert werden:
`CalculateHandler calculate;`
Diese ist ein Zeiger auf eine beliebige Methode die in Argumentliste und Rückgabewert mit dem Delegate übereinstimmt.
Der Zeiger kann wie folgt auf eine vorhandene Methode gesetzt werden:
`calculate = someMatchingFunction;`
Danach wird die zugewiesene Methode aufgerufen durch :
`calculate(value1, value2)` ⇒ ruft dann `someMatchingFunction(value1, value2)` auf.

BEISPIEL

```
public delegate double CalculateHandler(double value1, double
    value2);

class Functions {
  public static double Add(double x, double y) {return x + y;}
  public static double Subtract(double x, double y) {return x - y;}
}
// ...
static void Main(string[] args) {
  CalculateHandler calculate;
  Console.Write("Geben Sie den ersten Operanden ein: ");
  double input1 = Convert.ToDouble(Console.ReadLine());
  Console.Write("Geben Sie den zweiten Operanden ein: ");
  double input2 = Convert.ToDouble(Console.ReadLine());
  Console.Write("Operation: Addition - (A) oder Subtraktion - (S)?
      ");
  string wahl = Console.ReadLine().ToUpper();

  if (wahl == "A")
    calculate = Functions.Add;
  else if (wahl == "S")
    calculate = Functions.Subtract;
  else
    return;

  double result = calculate(input1, input2);
```

```
  Console.WriteLine("Ergebnis = {0}\n\n", result);
}
```

Soweit ist das ganze erstmal äquivalent zu:

```
double result;
if(wahl == "A")
  result = Functions.Add(input1, input2);
else if(wahl == "S")
  result = Functions.Subtract(input1, input2);
```

Nur mit dem Unterschied das die Ausführung der Funktion innerhalb der if-Abfrage steht.

9.1.1 MULTICAST DELEGATES

Mehrere Delegates werden in einem kombiniert.
Aufruf des Delegates führt also mehrere Funktionen hintereinander aus:

```
calculate = Functions.Subtract;
calculate += Functions.Add;
double result = calculate(input1, input2);  // nur der return-Wert
   der letzten Delegate-Funktion (hier: Add) wird übergeben. In
   diesem Fall ist der Multicast also sinnfrei, man könnte ihn aber
    sehen, wenn in den beiden Funktionen noch etwas auf die Konsole
    geschrieben wird.
```

9.1.2 GENERISCHE DELEGATES

C# stellt die meistbenutzten Delegates zur Verfügung

```
delegate void Action();
delegate TReturn Func<TReturn>();
delegate TReturn Func<TArg, TReturn>(TArg arg);
delegate TReturn Func<TArg1, TArg2, TReturn>(TArg1 arg1, TArg2 arg2
   );
```

Diese sind vordefiniert und können überall benutzt werden.
Man könnte also auf die Deklaration von CalculateHandler verzichten und diesen überall ersetzen durch:

```
CalculateHandler calculate;
// kann damit ersetzt werden:
Func<double, double, double> calculate;
```

9.1.3 LAMBDA FUNKTIONEN

(anonyme Funktionen)
Syntax: (Argumente) => Funktionsaufruf

BEISPIEL

```
if (wahl == "A")
  calculate = (x, y) => x+y;
else if (wahl == "S")
  calculate = (x, y) => x-y;
```

Einschränkungen für lambda Funktionen:

- Zugriff nur auf lokale Variablen
- keine Sprunganweisungen (break, goto)

WEITERE BEISPIELE

Was ist der Typ dieser Funktionen?

```
Func<double, double> square = // Typ
  x => x * x;
Func<double> answer = // Typ
  () => 42;
Action sayHello = // Typ
  () => Console.WriteLine("Hallo");
Func<double> helloAnswer =  // Typ
  () => {
    Console.WriteLine("Hallo");
    return 42;
  }
```

9.1.4 BEISPIELE

Vorlesung
28.04.2017

BEISPIEL 1

Was für Methoden beschreibt del?

```
public delegate int del(int x, int y);

del summe = (x,y) => x+y;
int result = summe(1,2); // result = 3
summe = (x,y) => x-y;
result = summe(1,2);  // result = -1
del bxy;
{ // lokaler Scope
  double b = 1;
  bxy = (x,y) => {return b*x+y;};
  b = 2;
}
// b=1; // würde zu Compiler-Fehler führen (b existiert nicht mehr)
// double b=1;  // Compiler-Fehler, obwohl sie eigentlich nicht
   mehr existiert

Console.WriteLine(bxy(1,0));  // output: 2
```

BEISPIEL 2

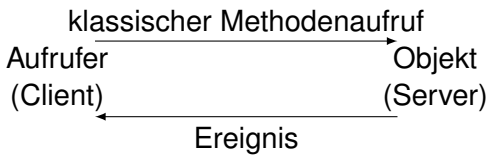
Sie haben ein int-Array `int[] myArray = new int[]{ 13, 39, 42, 53, 72, 23, 65};` und wollen alle Werte finden, die größer als 60 sind.
Dafür gibt es die statische Methode `Array.FindAll`, die alle Werte eines Arrays findet die eine bestimmte gegebene Bedingung erfüllen und als neues Array zurückgegeben:
`T[] Array.FindAll(T[] array, Predicate<T> match);//hier: T=int`
Die Find-Methode bekommt ein Objekt des Delegate `Predicate` übergeben der wie folgt definiert ist:
`delegate bool Predicate<T>(T obj);`

1. Was für eine Funktion beschreibt `Predicate`?
2. Was müssen wir schreiben um mit `Array.FindAll` alle Werte zu finden, die größer als 60 sind?

Antwort:

1. Ein Delegate das überprüft, ob das Objekt im Array einer Bedingung entspricht.
2.
```
int[] values2 = Array.FindAll(myArray, isGreater60 );
public bool isGreater60(int x){
  return (x>60);
}
// alternativ mit Lambda:
int[] values = Array.FindAll(myArray, (x) => x>60 );
```

9.2 EREIGNISSE



Ereignis: Aufruf einer Methode im Client
→herausragende Rolle bei GUI Programmierung

BEISPIEL

```
public class Circle {
  private double radius;
  public double Radius {
    get { return radius; }
    set { radius = value; }
  }
}

public class Program {
  static void Main(string[] args) {
    Circle kreis = new Circle();
    kreis.Radius = -1;
```

```
13    }
14
15    static void kreis_InvalidRadius() {
16      Console.WriteLine("Invalid Radius.");
17    }
18  }
```

ZIEL: Änderung des Radius zu einem negativen Wert soll die Methode `kreis_InvalidRadius()` aufrufen.

- `kreis_InvalidRadius()` ist ein EventHandler (behandelt ein auftretendes Ereignis)
- Dazu muss Circle die Methode `kreis_InvalidRadius()` übergeben werden
- Wie kann man eine Methode übergeben?

```
1  public class Circle {
2    // ...
3    public delegate void InvalidRadiusEventHandler();
4    public event InvalidRadiusEventHandler InvalidRadius;
5    public double Radius {
6      get { return radius; }
7      set {
8        if (value >= 0) Radius = value;
9        else if (InvalidRadius != null)  InvalidRadius();
10     }
11   }
12 }
13
14 public class Program {
15   static void Main(string[] args) {
16     Circle kreis = new Circle();
17     kreis.InvalidRadius += kreis_InvalidRadius;
18     kreis.Radius = -1;
19   }
20
21   static void kreis_InvalidRadius() {
22     Console.WriteLine("Invalid Radius.");
23   }
24 }
```

Schlüsselwort EVENT: kann prinzipiell auch weggelassen werden, ändert nur den Zugriff auf den delegate: für events nur mit += oder -= möglich ist

9.2.1 EREIGNISSE MIT ÜBERGABEPARAMETERN

Oft sollte der Ereignishandler noch Argumente übergeben bekommen

- wer hat das Ereignis ausgelöst (hier: kreis): SENDER
- zusätzliche ereignisspezifische Daten: EVENTARGS

Standard-Signatur für .NET Ereignishandler enthält diese 2 Argumente:
`static void kreis_InvalidRadius(object sender, EventArgs e)`

BEISPIEL

```
static void kreis_InvalidRadius(object sender,
    InvalidRadiusEventArgs e) {
  Console.Write("Ein Radius von {0} ist nicht zulässig.", e.
      invalidRadius);
  Console.Write("Neueingabe: ");
  ((Circle)sender).Radius = Convert.ToDouble(Console.ReadLine());
}

public class Circle {
  public delegate void InvalidRadiusEventHandler(object o,
      InvalidRadiusEventArgs e);
  // ...
  public double Radius {
    get { return radius; }
    set {
      if (value >= 0)
        radius = value;
      else
        InvalidRadius( this, new InvalidRadiusEventArgs(value) );
    }
  }
}

public class InvalidRadiusEventArgs : EventArgs {
  public double invalidRadius;
  public InvalidRadiusEventArgs(double invalidRadius_) {
    invalidRadius = invalidRadius_;
  }
}
```

Frage: Was passiert wenn nach einem falschen Radius, bei Neueingabe wieder etwas negatives eingegeben wird?

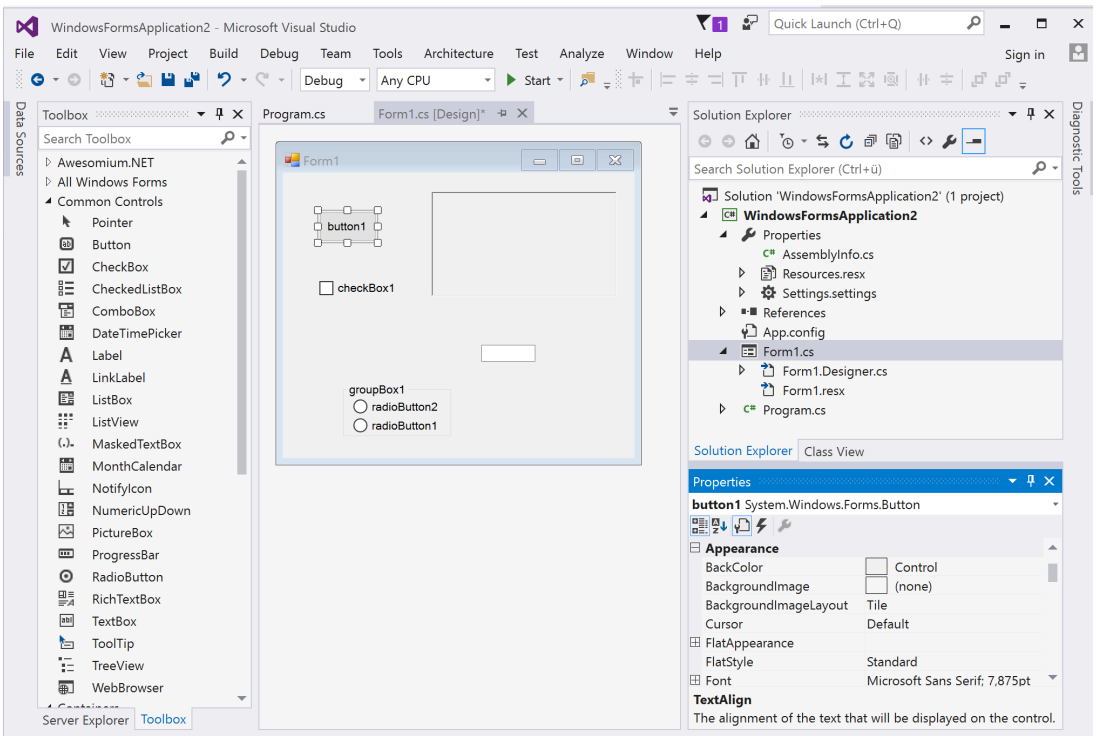
- Das Event wird immer wieder im `set` aufgerufen.

10 WINDOWS FORMS

Windows Forms ist ein GUI Toolkit für .NET.
Alternative GUI Schnittstelle: Windows Presentation Foundation (schwieriger)

10.1 WINDOWS FORMS ANWENDUNG ERSTELLEN

New → Project → C# → Windows Forms Application



10.1.1 FORM.CS

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace MyFirstForm {
  public partial class Form1 : Form { // partial: Definition von
      Klasse Forme1 ist auf mehrere .cs-Dateien verteilt.
    public Form1() {
      InitializeComponent();
    }
  }
}
```

10.1.2 FORM.DESIGNER.CS

```
namespace MyFirstForm {
  partial class Form1 { // partial: hier geht's weiter
    // ...
    private void InitializeComponent() {
      this.button1 = new System.Windows.Forms.Button();
      this.SuspendLayout();
      //
      // button1
      //
      this.button1.Location = new System.Drawing.Point(84, 95);
      this.button1.Name = "button1";
      this.button1.Size = new System.Drawing.Size(75, 23);
      this.button1.TabIndex = 0;
      this.button1.Text = "button1";
      this.button1.UseVisualStyleBackColor = true;
      // ...
    }
  }
}
```

10.1.3 PROGRAM.CS

```
using System;
using System.Windows.Forms;

namespace MyFirstForm {
  static class Program {
    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
  }
}
```

10.2 EIN NEUES CONTROL GENERIEREN

```
public Form1() {
  InitializeComponent();

  var myLabel = new Label();
  //Set some properties
  myLabel.Text = "Some example text";
  myLabel.Location = new Point(100, 60);
  myLabel.ForeColor = Color.Red;
```

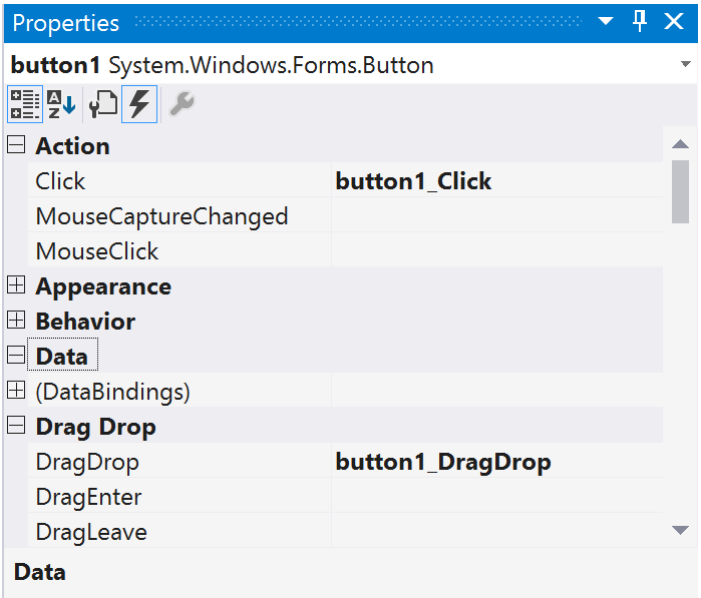
```
  // Without the following line we would see nothing
  this.Controls.Add(myLabel);
}
```

10.3 HOSTING

- alle Controls (außer Form) werden in einem anderen Control gehostet
- Sichtbarkeit und Position des Controls sind dann fest mit dem Host verbunden

10.4 AUF EVENTS REAGIEREN

Beispiele für Events: Click, DragDrop, MouseMove, KeyDown
Beim Auftreten dieser Events können EventHandler aufgerufen werden die Code ausführen.
EventHandler können für jedes Control in den Properties definiert werden



10.5 BENUTZERDEFINIERTES ZEICHNEN

Zentrales Object: `System.Drawing.Graphics`

```
System.Drawing.Graphics graphics = this.CreateGraphics();
```

Enthält viele Befehle die mit Draw oder Fill anfangen.

Draw-Befehle benötigen einen Pen (definiert Zeichen-Stil und Farbe):

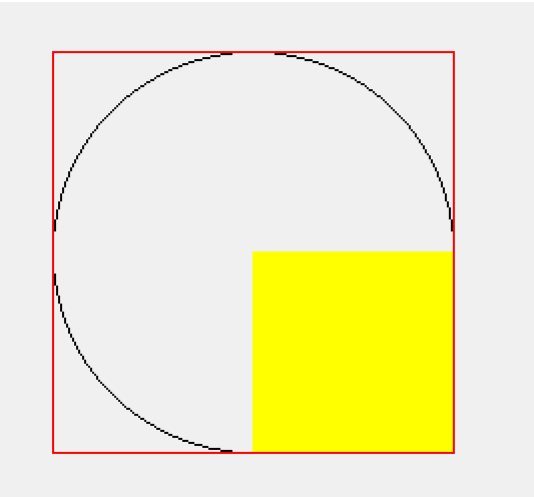
```
Pens.Red // einfacher vordefinierter roter Stift
new Pen(Color.Red, 4f) // Farbe Rot, Stiftdicke 4
```

Fill-Befehle benötigen einen Brush (z.B. SolidBrush, TextureBrush, ...):

```
Brushes.Yellow // Vordefinierter gelber SolidBrush
```

BEISPIEL

```
System.Drawing.Rectangle rectangle = new System.Drawing.Rectangle
    (100, 100, 200, 200);
graphics.DrawEllipse(Pens.Black, rectangle);
graphics.DrawRectangle(Pens.Red, rectangle);
graphics.FillRectangle(Brushes.Yellow, new Rectangle(200,200, 100,
    100));
```



10.6 TRANSFORMATION VON GRAPHIKOBJEKTEN

Graphikobjekte können

- skaliert (`graphics.ScaleTransform(x_factor, y_factor);`)
- verschoben (`graphics.TranslateTransform(x, y);`)
- rotiert (`graphics.RotateTransform(angle);`)

werden.

10.7 PERSISTENZ

Graphikobjekte, die mit einfach bei einer Aktion gemalt werden, können wieder verschwinden, wenn das Fenster verkleinert und wieder vergrößert wird, das Fenster aus dem Bildschirm raus und wieder rein geschoben wird usw.
Damit sie persistent bleiben, müssen sie in der Funktion `OnPoint()` gezeichnet werden.

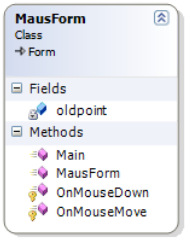
10.8 EVENTS

10.8.1 MAUSEREIGNISSE

Wir müssen uns (bei fast allen Ereignissen!) jeweils für einen dieser beiden Wege entscheiden:

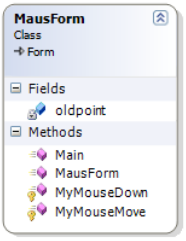
Es gibt zwei Möglichkeiten für solche Ereignishandler:

1. → Wir überschreiben die Ereignishandler der Basisklasse …



ist objektorientierter ☺

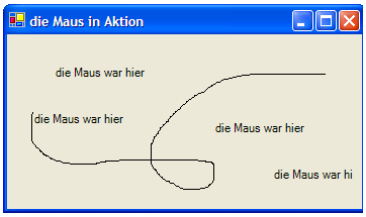
2. → Wir definieren eigene Ereignishandler in unserer Klasse …

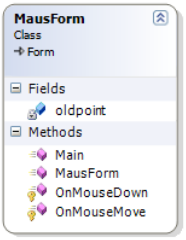


ist ereignisorientierter ☺
wird auch im Visual Studio automatisch genutzt

Wie macht man das ???

1. → Wir überschreiben die Ereignishandler der Basisklasse …






```
class MausForm : Form
{
    Point oldpoint;
    public static void Main()
    {   // the "main() - Funktion"
        Application.Run(new MausForm());
    }
    public MausForm()
    {           // Konstruktor
        Text = "die Maus in Aktion";
        oldpoint = new Point(0, 0);
    }
    protected override void OnMouseMove(MouseEventArgs mea) {
        Graphics grfx = CreateGraphics();
        if (mea.Button == MouseButtons.Left) {
            grfx.DrawLine(Pens.Black, oldpoint, new Point(mea.X, mea.Y));
            oldpoint = new Point(mea.X, mea.Y);
        }
        grfx.Dispose();
      }
    protected override void OnMouseDown(MouseEventArgs mea) {
        Graphics grfx = CreateGraphics();
        grfx.DrawString("die Maus war hier", Font, Brushes.Black,
                        mea.X, mea.Y);
        grfx.Dispose();
        oldpoint = new Point(mea.X, mea.Y);
    }
}
```

2. → Wir definieren eigene Ereignishandler in unserer Klasse …

```
class MausForm : Form
{
    Point oldpoint;
    public static void Main()
    {   // the "main() - Funktion"
        Application.Run(new MausForm());
    }
    public MausForm()
    {         // Konstruktor
        Text = "die Maus in Aktion";
        oldpoint = new Point(0, 0);
        this.MouseDown += new MouseEventHandler(MyMouseDown);
        this.MouseMove += new MouseEventHandler(MyMouseMove);
    }
    protected void MyMouseDown(object sender, MouseEventArgs mea)
    {
        Graphics grfx = CreateGraphics();
        grfx.DrawString("die Maus war hier", Font, Brushes.Black,
                        mea.X, mea.Y);
        grfx.Dispose();
        oldpoint = new Point(mea.X, mea.Y);
    }
    protected void MyMouseMove(object sender, MouseEventArgs mea)
    {
        Graphics grfx = CreateGraphics();
        if (mea.Button == MouseButtons.Left)
        {
            grfx.DrawLine(Pens.Black, oldpoint, new Point(mea.X, mea.Y));
            oldpoint = new Point(mea.X, mea.Y);
        }
        grfx.Dispose();
    }
}
```

nutzbare Ereignisse (Auswahl)

Ereignis	Methode	Argument
MouseDown	OnMouseDown	MouseEventArgs
MouseUp	OnMouseUp	MouseEventArgs
MouseMove	OnMouseMove	MouseEventArgs
MouseWheel	OnMouseWheel	MouseEventArgs ...

MouseEventArgs

Typ	Name	Zugr.	Beschreibung
int	X	get	x-Position
int	Y	get	y-Position
MouseButtons	Button	get	Maustaste
int	Clicks	get	bei Doppelklick ==2
int	Delta	get	Mausradbewegung

MouseButtons

None, Left, Right, Middle, ...

10.8.2 TIMEREREIGNISSE

Timer-Ereignisse können durch die Timerklasse behandelt werden ...

wozu?
- Erzeugen von Ereignissen in festen Abständen
- Steuerung von Animationen

Merke:
- es gibt maximal ein Timerereignis in der Warteschlange
- minimale Auflösung ca. 10 ms
- nicht sonderlich genau (+-10 ms)

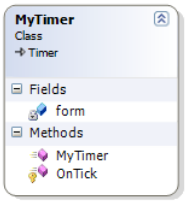
die Timer-Klasse (Auswahl)

- Eigenschaften:

Typ	Name	Zugriff	Beschreibung
int	Interval	get/set	Timerperiode in ms
bool	Enabled	get/set	möchte ich den Timer benutzen?

- Methoden:

Typ	Name	Beschreibung
void	Start()	startet den Timer
void	Stop()	stoppt den Timer



1.

```
class MyTimer: Timer {
   RandomRectangle form;
   public MyTimer (RandomRectangle f) {
      form=f;
   }
   protected override void OnTick(EventArgs ea) {
      form.DrawIt();
   }
}
```

10.8.3 TASTATUREREIGNISSE

Consolenprogramme warten auf Tastaturereignisse ☹
Windowsprogramme reagieren auf Tastaturereignisse ☺ ...

nutzbare Ereignisse (Auswahl)

Ereignis	Methode	Argument
KeyDown	OnKeyDown	KeyEventArgs
KeyUp	OnKeyUp	KeyEventArgs
KeyPress	OnKeyPress	KeyPressEventArgs

KeyEventArgs

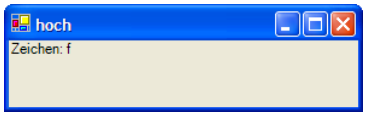
Typ	Name	Zugr.	Beschreibung
Keys	KeyCode	get	Identifiziert die Taste
Keys	Modifiers	get	Identifiziert den Umschaltstatus
bool	Shift	get	true bei gedrückter Shift-Taste
bool	Control	get	true bei gedrückter Control-Taste
bool	Alt	get	true bei gedrückter Alt-Taste

...

KeyPressEventArgs

Typ	Name	Zugr.	Beschreibung
char	KeyChar	get	Unicode-Zeichen (16 bit) ...

Keys
- **Buchstaben:** `Keys.A ... Keys.Z`
- **Zahlen:** `Keys.D0 ... Keys.D9`
- **Funktionstasten:** `Keys.F1 ... Keys.F24`
- **Steuer- und Cursortasten:**
  `Keys.Home, Keys.End, Keys.PageUp, Keys.PageDown,`
  `Keys.Up, Keys.Down, Keys.Left, Keys.Right, ...`



Beispiel 1: Arbeit mit den Cursortasten

1. Überschreiben des geerebten Ereignishandlers

```
protected override void OnKeyDown(KeyEventArgs kea)  {
     switch(kea.KeyCode) {
      case Keys.Right: Text = "rechts";      break;
      case Keys.Left:  Text = "links";       break;
      case Keys.Up:    Text = "hoch";        break;
      case Keys.Down:  Text = "nach unten"; break;
  }
}
```

2. einen eigenen Ereignishandler definieren

```
this.KeyDown += new KeyEventHandler(MyKeyDown);
  ...

protected void MyKeyDown(object sender, KeyEventArgs kea) {
    ... // wie bei Variante 1
}
```

10.8.4 PARALLELITÄT VON EREIGNISSEN

professionelle Anwendungen halten die Applikation jederzeit offen für „parallel" anfallende Ereignisse!

Das Problem: Ereignisse werden nicht bearbeitet, während ein anderes Ereignis behandelt wird

```
class Parallel: Form {
  public static void Main() {   // the "main() - Funktion"
    Application.Run(new Parallel());
  }
  public Parallel() {           // Konstruktor
    Text = "Parallele Ereignisse";
  }
  protected override void OnMouseDown(MouseEventArgs mea) {
     Graphics grfx = CreateGraphics();
     for(int i=0; i<50000; i++) {
         grfx.DrawString("Blockiert", Font, Brushes.Black,
                         mea.X, mea.Y);
         // Application.DoEvents();
     }
     grfx.DrawString("Blockierung beendet", Font, Brushes.Black,
                         mea.X, mea.Y+20);
     grfx.Dispose();
  }
}
```

Die Lösung: Ereignishandler wird explizit aufgerufen

```
Application.DoEvents();
```

Beachten Sie dies, sobald ein Ereignishandler länger als 250 ms für seine Aufgaben benötigt!!!

11 SOCKETS

11.1 NETZWERKPROGRAMMIERUNG MIT DER KLASSE SOCKET

11.1.1 KOMMUNIKATION MIT SOCKETS

wozu?

- als "standardisierte" Schnittstelle zur Programmierung von Client-Server-Anwendungen (nicht nur!)

Generelles:

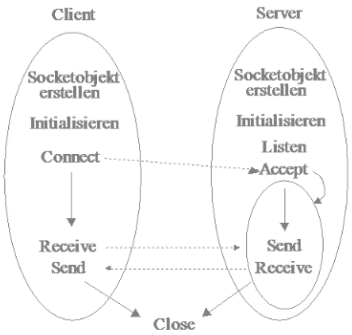
- *Definition:*
  Ein Socket (deutsch: Steckdose) ist ein Kommunikationsendpunkt innerhalb einer
  Anwendung. Er wird durch eine IP-Adresse und eine Portnummer adressiert.
  Windows-Sockets implementieren Funktionalität von Berkley Sockets (BSD)

- Sockets sind *protokoll-unabhängig*.
  Die Programmierung des Datenaustausch erfolgt unabhängig vom Protokoll.

- Sockets sind aber *nicht protokoll-transparent:*
  Beim Erzeugen eines Socket wird das Protokoll festgelegt (TCP, UDP, ...).

Prinzip:

- dieses Prinzip gilt für Sockets generell
  (unabhängig von der Programmiersprache oder dem Betriebssystem)



11.1.2 FUNKTIONALITÄT

wir arbeiten mit 2 Namespaces:

```
using System.Net;
using System.Net.Sockets;
```

wir benötigen 3 Klassen:

- `IPAddress` stellt eine Internetprotokolladresse (IP) bereit:

```
IPAddress myIP      = IPAddress.Parse("192.168.1.2");
IPAddress[] myIP2   = Dns.GetHostAddresses("iaix1.informatik.htw-dresden.de");
IPAddress[] myIP2   = Dns.GetHostAddresses("iaix1.informatik.htw-dresden.de");
IPAddress[] hostadd = Dns.GetHostAddresses("localhost");
```

- `IPEndPoint` repräsentiert eine Internet-Adresse incl. Port-Nummer:

```
IPEndPoint EPhost = new IPEndPoint(hostadd[0], 20000);
```

- Was bedeutet Portnummer?
  -> *Service Datei:*

```
/etc/services                              /* UNIX */
\windows\services                          /* Win95, 98, Me */
\windows\SYSTEM32\DRIVERS\ETC\SERVICES     /* WinNT, XP, Vista, Win7 */
```

```
# This file contains port numbers for well-known services as defined by
# RFC 1060 (Assigned Numbers).
# Format:
# <service name>  <port number>/<protocol>  [aliases...]   [#<comment>]
#
echo                7/tcp
echo                7/udp
...
ftp                21/tcp              # ftp port
telnet             23/tcp              # telnet port
smtp               25/tcp       mail   # email port
...
```

- Socket
  repräsentiert einen synchronen bzw. asynchronen Socket (egal ob Client oder Server)

```
// Creates the Socket for sending/receiving data over TCP.
Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
                      ProtocolType.Tcp );
```

  - AddressFamily

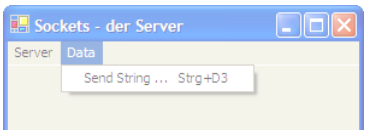
```
AddressFamily.InterNetwork     // IP Version 4
AddressFamily.InterNetworkV6   // IP Version 6
AddressFamily.AppleTalk        // Protokoll für MACOS
...
```

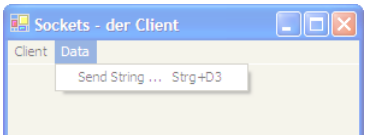
  - SocketType

```
SocketType.Stream  // TCP/IP
SocketType.Dgram   // UDP/IP
```

  - ProtocolType

```
ProtocolType.Tcp   // TCP/IP
ProtocolType.Udp   // UDP/IP
```





11.2 ARBEITEN MIT SOCKET-OBJEKTEN

11.2.1 SYNCHRONE UND ASYNCHRONE SOCKETFUNKTIONEN

synchrone Memberfunktionen:

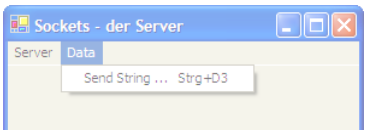
- das Programm wird unterbrochen, bis die Methode beendet wird
- Vorteil: nach dem Methodenaufruf steht das Ergebnis zur Verfügung
- Nachteil: das Programm blockiert bis z.B. ein Datenblock gesendet wurde (praktisch nur bei Consolenprogrammen sinnvoll !!!)
- Beispiele: `Connect, Send, Receive, Close, ...`

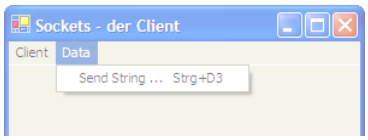
asynchrone Memberfunktionen:

- Methodenabarbeitung erfolgt in einem extra Thread
- Vorteil: kein Blockieren der Anwendung durch Socket-Methoden
  d.h. Ereignisbehandlung parallel zu Netzwerkoperationen möglich
- Nachteil: zusätzlicher Programmieraufwand (Begin-End-Methoden, Callback-Mechanismus)
- Beispiele: `s.BeginConnect(); ... s.EndConnect();`
  `s.BeginSend(); ... s.EndSend();`

Merke:

- das Programm kann asynchrone mit synchronen Funktionsaufrufen mischen !!!!
- wir arbeiten (wenn möglich) mit den asynchronen Methoden !!!!
- Asynchrone Sockets lassen sich nicht debuggen ☹ !!!!!! (*Ctrl-F5*)

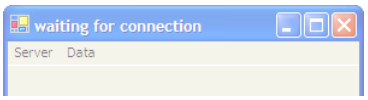




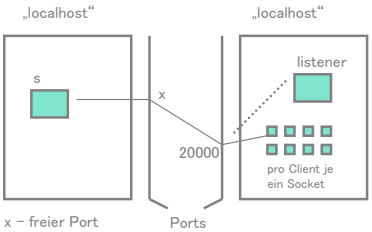
11.2.2 PORTWAHL BEI CLIENT UND SERVER

den **Server-Socket** müssen wir an einen Port binden und aktivieren:

```
// Creates the Socket for sending/receiving data over TCP.
Socket listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
                             ProtocolType.Tcp );
// IP-Adresse des Rechners ermitteln
IPAddress[] hostadd = Dns.GetHostAddresses("localhost");
// IP-Adresse und Portnummer verbinden
IPEndPoint EPhost = new IPEndPoint(hostadd[0], 20000);
try {
   listener.Bind(EPhost); // Socket an Port binden
   ...
} catch (Exception e) {   // "Fehlerbehandlung"
   Text = (e.ToString()); return;
}
Text = "waiting for connection";
```



ein **Client-Socket** wird erst dann aktiviert, wenn er mit dem Server verbunden wird,
-> er sucht sich einen "freien" Port ...



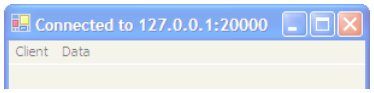
das **Client-Socketobjekt** wird über einen freien Port der aktuellen Maschine und dem spezifizierten Port der Zielmaschine (`remoteEP`) mit dem Serversocket verbunden

wie?

- **Synchron** mit `void Socket::Connect(EndPoint remoteEP);`

```
// create a new IPEndPoint (sockets destination)
IPAddress[] hostadd = Dns.GetHostAddresses("localhost");
IPEndPoint EPhost = new IPEndPoint(hostadd[0], 20000);
try {  // Connects to the host using IPEndPoint.
    s.Connect(EPhost);
} catch (Exception e) {    // "Fehlerbehandlung"
    Text = (e.ToString()); return;
}
if (s.Connected)
      Text = String.Format("Connected to {0}",
                           s.RemoteEndPoint.ToString());
```

Blockiert bis zur Verbindung (oder „Timeout") ...



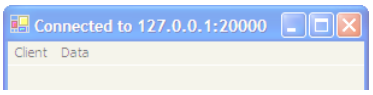
11.2.3 INTERPROZESSKOMMUNIKATION

- **Asynchron**
  mit `IAsyncResult Socket::BeginConnect(EndPoint remoteEP, AsyncCallback callback, object state);`
  `void Socket::EndConnect(IAsyncResult asyncResult);`

wir benötigen hierzu einen *Ereignishändler* (Delegaten, Callbackfunktion):

```
// create a new IPEndPoint (sockets destination)
IPAddress[] hostadd = Dns.GetHostAddresses("localhost");
IPEndPoint EPhost = new IPEndPoint(hostadd[0], 20000);

// Connects to the host using IPEndPoint.
s.BeginConnect(EPhost, new AsyncCallback(ConnectCallback),s);
...

private void ConnectCallback(IAsyncResult ar) {
     try {  // Complete connecting to the remote device.
       s.EndConnect(ar);
       // Begin to receive data.
       ...
     } catch (Exception e) {
       Text = e.ToString();
     }
}
```

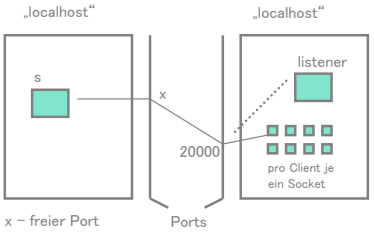
Die Funktion gibt die Verbindung lediglich „in Auftrag" ...



Bemerkung: damit das funktioniert, muss auf der Serverseite ein Socket an dem entsprechenden Port gebunden sein ("lauschen", "listen") ...

11.2.4 SOCKETANZAHL

Beachten Sie: Für jeden Client, der sich am Server meldet, wird ein neuer Socket zur Kommunikation verwendet !!!!



- "Bereitschaftszustand" (listening state) setzen:
  nur für Serverseite – setzt "listening state" ... blockiert nicht!

  `void Socket::Listen(int backlog);`

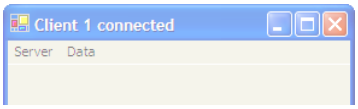
  Parameter `backlog`:
  max. Anzahl der Client-Anforderungen (in der Socket-Queue)

- Beispiel:

```
IPAddress[] hostadd = Dns.GetHostAddresses("localhost");
IPEndPoint EPhost = new IPEndPoint(hostadd[0], 20000);
try {
      listener.Bind(EPhost);
      listener.Listen(10); // Allows a queue of 10 connections.
      ...
```

Ein Server muss auf Clients reagieren:

- meldet sich ein Clientsocket am Serversocket an, so wird ihm jeweils ein neuer Socket zur Kommunikation zugeordnet
- **Synchron**: `Socket Socket::Accept()`
- **Beispiel**:
```
...
Socket handler = listener.Accept();
```



asynchron ???

Ein Server muss auf Clients reagieren:

- meldet sich ein Clientsocket am Serversocket an, so wird ihm jeweils ein neuer Socket zur Kommunikation zugeordnet
- **Asynchron**:

```
IAsyncResult Socket::BeginAccept(
                          AsyncCallback callback,
                          object state);
Socket EndAccept(IAsyncResult asyncResult);
```

- **Beispiel**:
  wir benötigen hierzu einen *Ereignishändler* (Delegaten, Callbackfunktion):

```
try {
   listener.Bind(EPhost);
   listener.Listen(10); // Allows a queue of 10 connections.
   listener.BeginAccept(new AsyncCallback(acceptCallback),s);
}  catch (Exception e) {
   Text = (e.ToString()); return;
}
...

public void acceptCallback(IAsyncResult ar) {
   Socket listener = (Socket) ar.AsyncState;
   sc = listener.EndAccept(ar); // sc is the new socket
   ...
   // get ready for next acception
   listener.BeginAccept(new AsyncCallback(acceptCallback),
                        listener);
   ...
}
```

11.3 SYNCHRONER DATENAUSTAUSCH

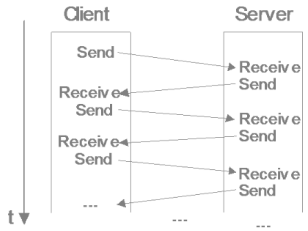
Das Prinzip des Datenaustauschs:

- Senden und Empfangen zwischen Client und Server ist wechselseitig über ein **Anwendungsprotokoll** zu definieren
- Protokoll muss klar sein: Festzulegen ist die Semantik der übertragenen Daten

Die hierfür **notwendigen Methoden der Socketklasse**:

```
int Socket::Send(byte[] buffer);
int Socket::Send(byte[] buffer, int size, SocketFlags sf);
int Socket::Send(byte[] buffer, int offset, int size, SocketFlags sf);

int Socket::Receive(byte[] buffer);
int Socket::Receive(byte[] buffer, int size, SocketFlags sf);
int Socket::Receive(byte[] buffer, int offset, int size, SocketFlags sf);
```

Parameter:	`buffer`	– klar ☺
	`size`	– Anzahl der Bytes
	`offset`	– Position innerhalb des Vektors (0-basiert)
	`flags`	– i.a. SocketFlags.None (entspricht dem Standardwert, siehe Online-Hilfe)
return-Wert:	Anzahl der übertragenen Zeichen kann kleiner sein als size (!!!!!) –> in Zyklus einbinden	



11.4 SYNCHRON VS ASYNCHRON

Beispiele für synchronen Datenaustausch (praktisch nur für Consolenprogramme sinnvoll):

1. Übertragen einer Testzeichenkette (konvertiere Unicode zu BYTE/ASCII)

```
byte[] msg = System.Text.Encoding.ASCII.GetBytes("This is a test");
int bytesSent = s.Send(msg);
```

2. maximal 1024 Bytes empfangen (beachte: `Receive` blockiert die Anwendung bis Daten anliegen !!!!!)

```
byte[] bytes = new byte[1024];
int bytesRec = s.Receive(bytes, 1024);
Console.WriteLine("Echoed text = {0}",
                  System.Text.Encoding.ASCII.GetString(bytes, 0, bytesRec));
```

Besser ist asynchroner Datenaustausch:

die Anwendung blockiert nun nicht mehr, vielmehr wird der Delegat gestartet, sobald Daten zum Empfang anliegen ...

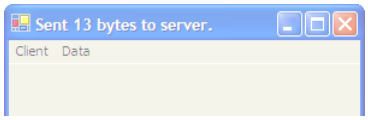
```
IAsyncResult Socket::BeginSend(byte[] buffer, int offset, int size,
                               SocketFlags socketFlags, AsyncCallback callback, object state);
int Socket::EndSend(IAsyncResult asyncResult);

IAsyncResult Socket::BeginReceive(byte[] buffer, int offset, int size,
                               SocketFlags socketFlags, AsyncCallback callback, object state);
int Socket::EndReceive(IAsyncResult asyncResult);
```

Parameter:	siehe Online-Hilfe
return-Wert:	`IAsyncResult`-Interface zur Beschreibung des Vorgangs `int`-Wert mit der Anzahl der gesendeten, gelesenen Bytes

11.4.1 BEISPIELE

1. Übertragen einer Testzeichenkette:

```
const int BufferSize = 256;            // Size of buffer.
byte[] buffer = new byte[BufferSize];  // buffer.

String str = String.Format("Datenpacket {0}", ++i);
buffer = Encoding.ASCII.GetBytes(str);
s.BeginSend(buffer, 0, buffer.Length, SocketFlags.None,
            new AsyncCallback(SendCallback), s);
...

private void SendCallback(IAsyncResult ar) {
  try {
    // Complete sending the data to the remote device.
    int bytesSent = s.EndSend(ar);  // keep in mind: a loop could be necessary
    // Signal that all bytes have been sent.
    Text = String.Format("Sent {0} bytes to server.", bytesSent);
  } catch (Exception e) {
    Text = e.ToString();
  }
}
```



2. Empfangen der Testzeichenkette:

```
const int BufferSize = 256;            // Size of buffer.
byte[] buffer = new byte[BufferSize];  // buffer.

s.BeginReceive(buffer, 0, buffer.Length, 0,
               new AsyncCallback(ReadCallback), s);
...
```

```
public void ReadCallback(IAsyncResult ar) {
      try {   int bytesRead = s.EndReceive(ar);
              if (bytesRead > 0) {
                  Text = Encoding.ASCII.GetString(buffer, 0, bytesRead);
                  s.BeginReceive(buffer, 0, buffer.Length, 0,
                                 new AsyncCallback(ReadCallback), s);
              }
      } catch (Exception e) {
          Text = "Client disconnected"; s.Close();
      }
}
```

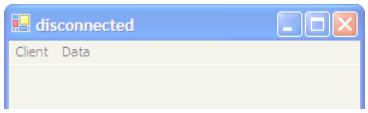
wir bereiten gleich wieder die nächste Leseoperation vor ··· ☺

wenn eine Socketverbindung geschlossen wird, bekommen wir für offene Lesevorgänge eine Exception und müssen unseren Socket auch schließen !!!!

11.5 SOCKETS TRENNEN UND BEENDEN

Sockets trennen und beenden:

```
s.Close();
...
if (!s.Connected) Text = "disconnected";
```



Beachte:
Sockets eignen sich nicht nur zur Kommunikation zwischen verteilten Client-Server-Anwendung, sondern auch zur Interprozesskommunikation ... d.h. wir haben Server- und Clientsocket auf einem Rechner laufen …

```
// IP-Adresse des Rechners ermitteln
IPAddress hostadd = Dns.Resolve("localhost").AddressList[0];
```

12 DLLS/COM

12.1 EXTERNE DLLS EINLESEN/NUTZEN

Dynamic Link Library (DLL) bezeichnet allgemein eine dynamische (d.h. zur Laufzeit nutzbare) Bibliothek ...

Unterscheidung zu .dll Assemblies

- DLL Dateien die mit einem .NET Compiler erstellt wurden liegen in intermediate language vor können einfach .NET Programm hinzugefügt werden (Bsp: BombLibrary)
- Allgemeine DLL Dateien die nicht von .NET stammen liegen jedoch in Maschinencode vor, müssen anders eingebunden werden
- übliche Dateiendungen:
  - DLL klar
  - OCX für Bibliotheken mit ActiveX Steuerelementen, DirectX-Componenten

Welche Vorteile hat das?

- Jeglicher Programmcode, der von mehr als einer Anwendung benötigt werden könnte, wird nur in einer einzelnen Datei auf der Festplatte gespeichert …
- … und bei Bedarf nur einmal in den Hauptspeicher geladen.
- Bug-Fixing, Verbesserungen usw. betreffen „nur" eine DLL-Datei (kleine „Patches" für große Softwareprodukte
- einfache Programmerweiterungen über Plugins (-> Drittanbieter, Freeware, …)
- modularer Aufbau
- „alter", getesteter, optimierter Programmcode (auch MFC, C, Assembler, …) kann in moderne APIs (.NET) integriert werden …

Welche Nachteile sind bekannt?

- mehrere Anwendungen benötigen verschiedene Versionen der gleichen DLL

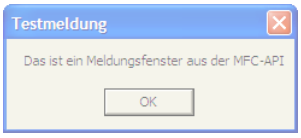
-> **Lösungen:**
- COM-Technologie auf der Basis von versionsbasierten Interfaces
- .NET umgeht Versionskonflikte, indem es die gleichzeitige Existenz von mehreren Versionen einer Programmbibliothek ermöglicht

wir arbeiten im Namespace: **System.Runtime.InteropServices**

das Attribut `[DllImport]`
- benennt die Dll-Datei
- deklariert die C#-Funktion, an die eine DLL-Funktion gebunden wird …

Beispiel 1: Variante ohne Parameter

```
[DllImport("user32.dll")]
 public extern static int MessageBox(uint HWnd, string lpText,
                                     string lpCaption, uint uType);
 ...
 MessageBox(0, "Das ist ein Meldungsfenster aus der Windows-API",
               "Testmeldung", 0);
```



Beispiel 2: die Funktion kann einen neuen Namen bekommen

`EntryPoint`: „Eintrittspunkt" in die DLL-Funktion,
d.h. der Name der gewünschten DLL-Funktion …

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
 public extern static int APIMessageBox(uint HWnd, string lpText,
                                        string lpCaption, uint uType);
```

beachte die Eigenschaften der deklarierten Methode:

- **static**:
  eine Methode aus einer DLL-Datei kann niemals Bestandteil einer Instanz sein, weil sie per Definition global verfügbar ist ...
- **extern**:
  klar! Es handelt sich ja um eine externe Methode ...

Wir haben noch ein Problem:

- die Daten der DLL-Funktionsparameter müssen auf C#-Datentypen abgebildet werden !!!
- folgende Regeln helfen bei einfachen Datentypen:

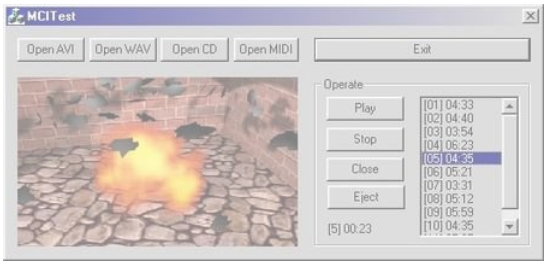
C/C++ -Datentyp	C#-Datentyp
float	float
double	double
signed char	sbyte
unsigned char	byte
short	short
unsigned short	ushort
long	int
unsigned long	uint
int64	long
unsigned int64	ulong
* (beliebige Pointer)	das bleibt ein Problem !!! IntPtr kann eine Lösung sein, bedingt aber zusätzliche Konvertierungen

!

```
IntPtr ipr = teste2();  // liefert eine Zeichenkette (char*)
string s = System.Runtime.InteropServices.Marshal.PtrToStringAnsi(ipr);
MessageBox.Show("Die Eingabe war: " + s);
```

auch auf Multimedia-Funktionen des Microsoft-Betriebssystems kann einfach zugegriffen werden ...

```
[DllImport("winmm.dll")]
private static extern long mciSendString(
                    string strCommand, StringBuilder strReturn,
                    int iReturnLength, IntPtr hwndCallback);

sCommand = "open \"datei.mpg\" type mpegvideo alias MediaFile";
mciSendString(sCommand, null, 0, IntPtr.Zero);

sCommand = "play MediaFile";
mciSendString(sCommand, null, 0, IntPtr.Zero);

sCommand = "stop MediaFile";
mciSendString(sCommand, null, 0, IntPtr.Zero);
```



12.2 COM-KOMPONENTEN

mit Hilfe der COM-Technologie kann zur Laufzeit auf externe Software-Komponenten zugegriffen werden ...

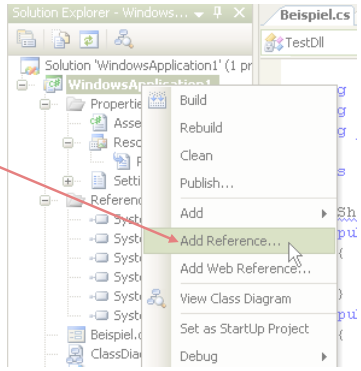
wie kann ich mir das vorstellen?

- es gibt Software-Produkte (Word, Excel, Internet Explorer, DirectX, Multimedia-Treiber, ··· die ihre Funktionalität als COM-Objekte bereitstellen ···
- wir binden nicht nur eine Funktion sondern eine Funktionssammlung als Objekt an ein Interface
- .NET stellt eine Vielzahl an COM-Wrappern für COM- Komponenten bereit ···

Beispiel 1: wir nutzen die Shell32.dll über einen .NET-Wrapper

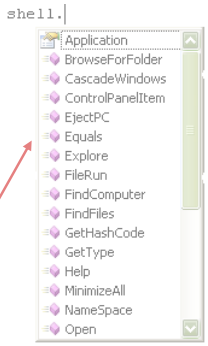
```
using Shell32;
Shell shell = new Shell();
...
```

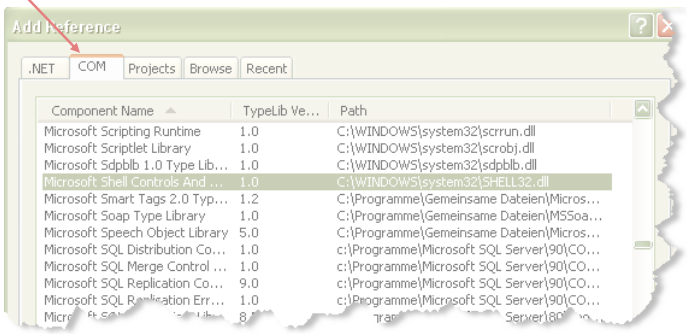
Vorsicht:
wir müssen eine Referenz auf das COM-Objekt in unser Projekt aufnehmen ...



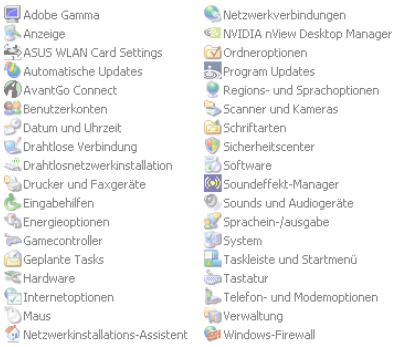

```
using Shell32;
Shell shell = new Shell();
...
```

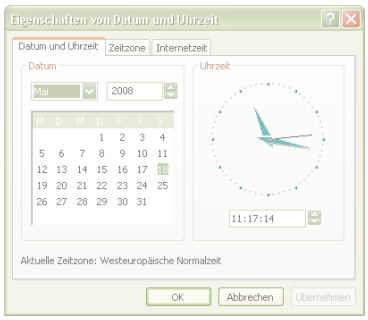
Bemerkung:
- über das COM-Objekt sind zum Übersetzungszeitpunkt Typprüfungen möglich
- mögliche Zugriffe/Methoden des Objekts werden wie bei .NET-Objekten angezeigt





```
using Shell32;
Shell shell = new Shell();

shell.Explore("c:\\");
shell.ControlPanelItem("");
shell.SetTime();
...
```

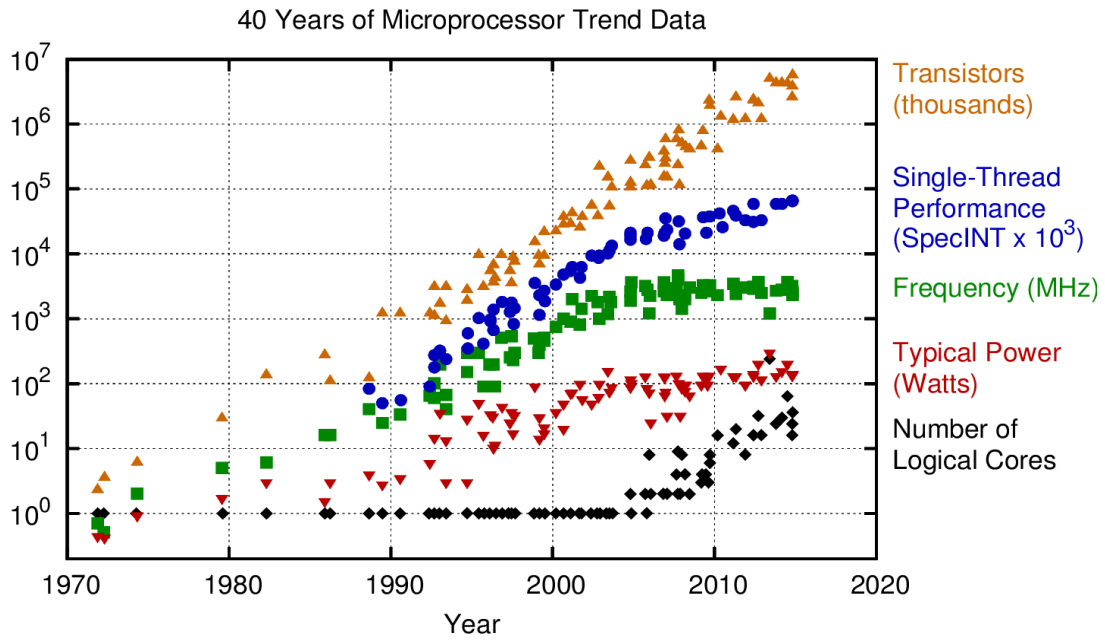




13 PARALLELE PROGRAMMIERUNG

13.1 EINLEITUNG

13.1.1 CPU ENTWICKLUNG



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

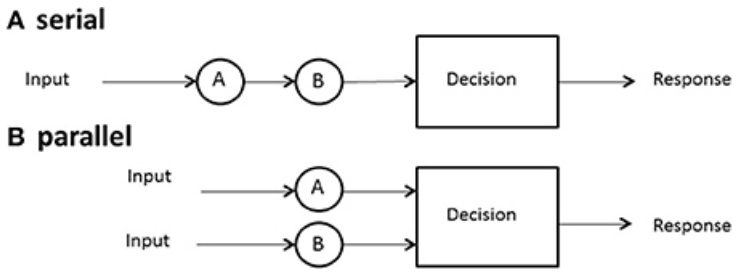
- Moore's law: alle 2 Jahre verdoppelt sich die Anzahl von Transistoren
- früher ging damit verdopplung der Taktfrequenz einher
- heute Taktfrequenz stagniert weil höhere Frequenz zu viel Hitze in der CPU erzeugt
- Performancesteigerung nur durch mehr Kerne möglich
- Neue Technologien wie Intel Hyper-Threading bilden auf jedem physikalischen Core noch 2 oder mehr virtuelle Cores
- Anzahl von physikalischen und virtuellen Kernen steigt rasant

13.1.2 TRADITIONELLE SEQUENTIELLE PROGRAMMIERUNG

- Software läuft sequentiell (nicht parallel) ab
- ein Algorithmus besteht aus einer Kette von Anweisungen, die nacheinander ablaufen
- ein einzelner CPU verarbeitet die Befehle
- mit mehreren Prozessoren wird die Software nicht schneller oder besser, dazu ist parallele Programmierung nötig

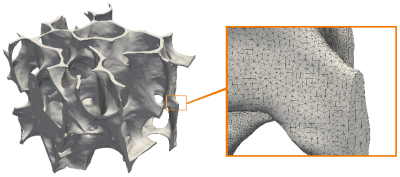
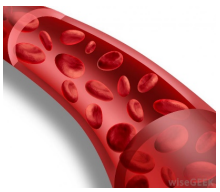
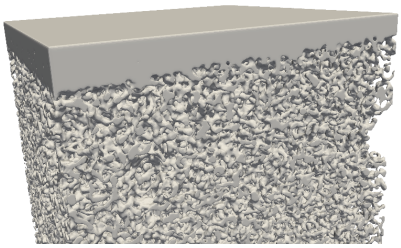
13.1.3 PARALLELE PROGRAMMIERUNG

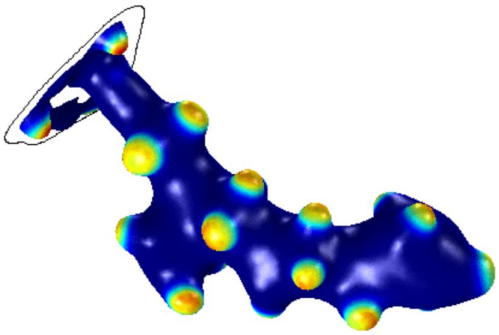
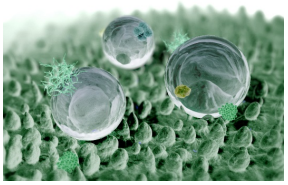
- mehrere Befehle werden parallel (simultan) abgearbeitet
- durch mehrere Threads, mehrere Prozessoren oder mehrere Rechner
- Parallel Programmierung
  - beschleunigt moderne Anwendungen oder ermöglicht sie erst
  - ermöglicht simultane Ausführung mehrer Anwendungsteile
- ist komplexer als sequentielle Programmierung



13.1.4 ANWENDUNGEN PARALLELER PROGRAMMIERUNG

- Bildverarbeitung, Computergraphik
- Aufwendige Berechnungen (Data-Mining)
- Webroboter (Indizierung des Internets)
- Künstliche Intelligenz
- Simulationen (Wetterprognose, Physikalische Simulationen)
- ...

13.1.5 MODERNE SUPERCOMPUTER

Aus https://de.wikipedia.org/wiki/Supercomputer

10,649,600 Number of cores in the Sunway TaihuLight supercomputer at the National Supercomputing Center in Wuxi, China as of June.
— TOP500 list of the world's top supercomputers, June 2016

Rank	Site	System	Cores	Rmax (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi, *China*	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway - *NRCPC*	10,649,600	93,014.6	15,371
2	National Super Computer Center in Guangzhou, *China*	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P - *NUDT*	3,120,000	33,862.7	17,808
3	DOE/SC/Oak Ridge National Laboratory, *United States*	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x - *Cray Inc.*	560,640	17,590.0	8,209
4	DOE/NNSA/LLNL *United States*	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom - *IBM*	1,572,864	17,173.2	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) *Japan*	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect - *Fujitsu*	705,024	10,510.0	12,660

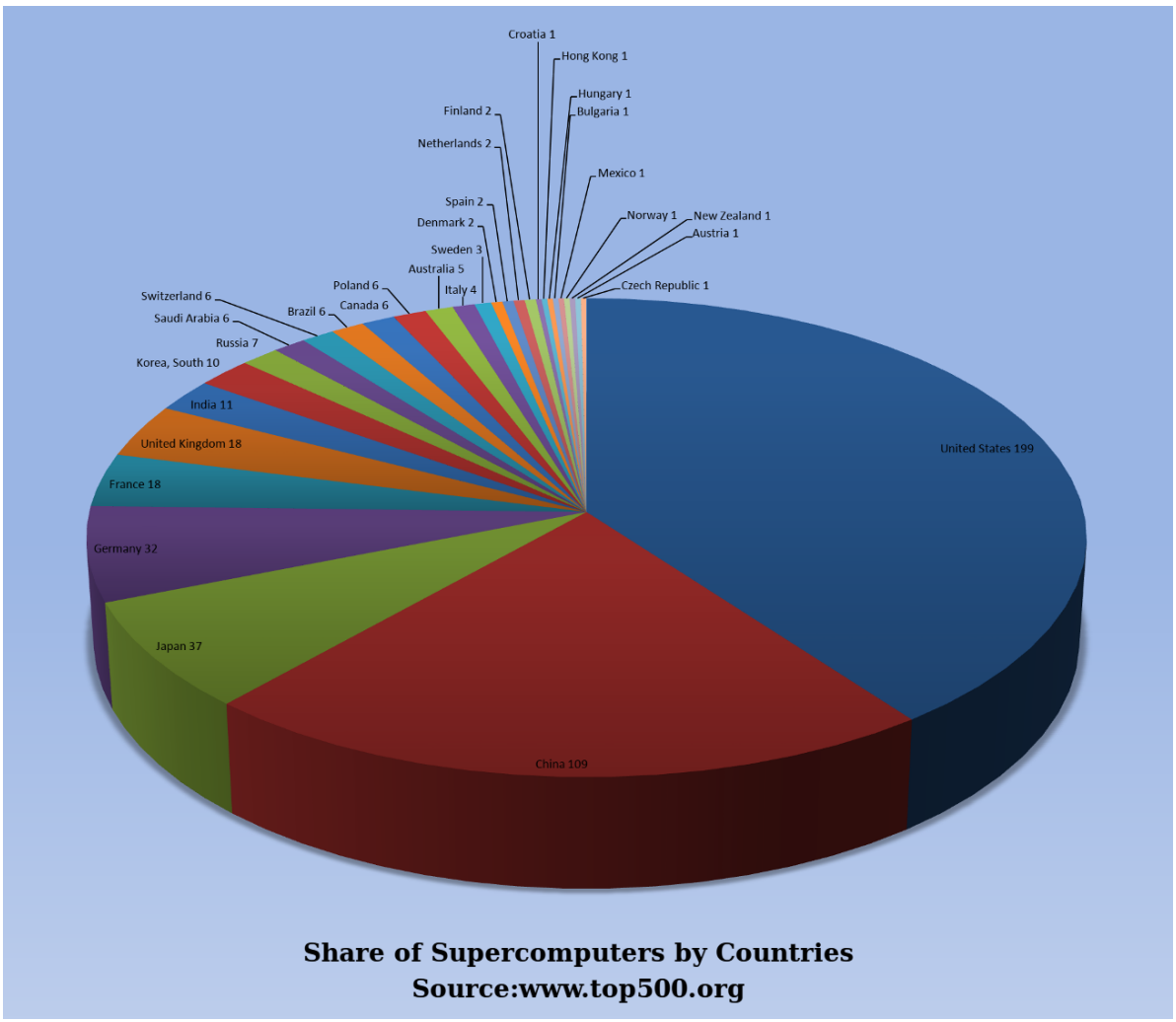
Aus http://www.digitaleng.news/de/wp-content/uploads/2016/11/BtN5.jpg

13.1.6 DEUTSCHE SUPERCOMPUTER

Name	Standort	TeraFLOPS	Konfiguration	TB RAM	Energiebedarf	Zweck
Hazel Hen	Höchstleistungsrechenzentrum Stuttgart	7.420,00	Cray Aries Netzwerk; 7712 Nodes mit je 24 Kernen (Intel Xeon E5-2680 v3, 30M Cache, 2,50 GHz)	987	˜3.200 kW	
JUQUEEN	Forschungszentrum Jülich	5.900,00	IBM BlueGene/Q, 28.672 Power BQC-Prozessoren (16 Kerne, 1,60 GHz)	448	2.301 kW	Materialwissenschaften, theoretische Chemie, Elementarteilchenphysik, Umwelt, Astrophysik
SuperMUC	Leibniz-Rechenzentrum (LRZ) (Garching bei München)	2.897,00	18.432 Xeon E5-2680 CPUs (8 Kerne, 2,7 GHz), 820 Xeon E7-4870 CPUs (10 Kerne, 2,4 GHz)	340	3.423 kW	Kosmologie über die Entstehung des Universums, Seismologie und Erdbebenvorhersage
HRSK-II	ZIH, TU Dresden	1.600,00	43.866 CPU Kerne, Intel Haswell-EP-CPUs (Xeon E5 2680v3), 216 Nvidia Tesla-GPUs	130		Wissenschaftliche Anwendungen
HLRE-3 "Mistral"	Deutsches Klimarechenzentrum Hamburg	1.400,00	1. Ausbaustufe (Juli 2015): 1.500 Knoten bullx B700 DLC, 36.000 Kerne (Haswell), 20 PB Lustre-Festplattensystem, 12 Visualisierungsknoten (je 2 Nvidia Tesla K80 GPUs), Warmwasserkühlung	120		Klimamodellierung
Cray XC40	Deutscher Wetterdienst (Offenbach)	1.100,00	Cray Aries Netzwerk; 1.952 CPUs Intel Xeon E5-2680v3/E5-2695v4	122	407 kW	Numerische Wettervorhersage und Klimasimulationen

Aus https://de.wikipedia.org/wiki/Supercomputer

13.1.7 SUPERCOMPUTER NACH LÄNDERN



13.2 MULTITHREADING

13.3 MULTITHREADING

Vorlesung
12.05.2017

13.3.1 VERSCHIEDENE METHODEN ZUR PARALLELISIERUNG:

- HOCHLEISTUNGSRECHNER: Verteilter Speicher, Kommunikation zwischen Rechnern durch Nachrichten (Message Passing, MPI)
- EINZELRECHNER/SMARTPHONES: gemeinsamer Speicher, Aufteilung des Programmablauf in Threads

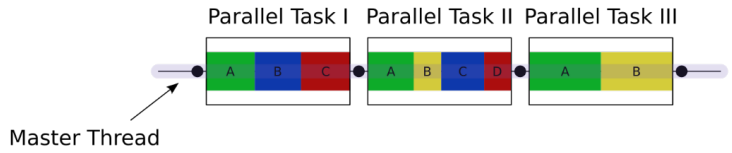
Wir beschränken uns auf letzteres!

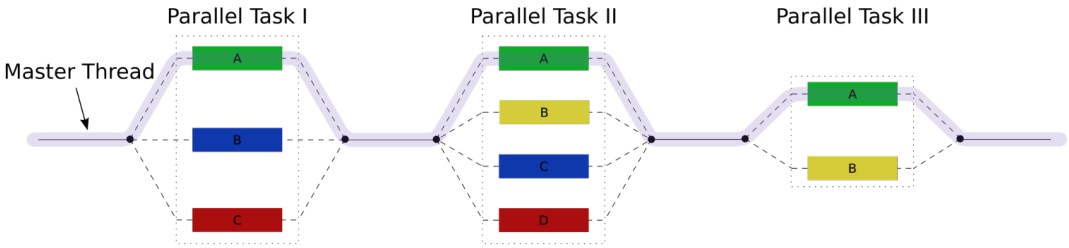
13.3.2 THREAD

- Anwendung
  - Prozess (Task)
    - Thread 1
    - Thread 2
    - Thread 3

Thread (Faden) = Teilprozess

- zum Start des Programms: 1 Thread (Masterthread)
- serielle Programmierung: es gibt nur diesen einen Thread
- Multithreading: es werden neue Threads gebildet die parallel abgearbeitet werden können





veranschaulicht:

→ fork (verzweigen)

→ join (zusammenführen)

- In einem Prozessor kann zu jedem Zeitpunkt nur 1 Thread bearbeitet werden
- sind mehrere Threads → Warteschlange, Austausch in kurzen Zeitintervallen (Standard: 20ms), Scheduling durch das System
- Threads haben verschiedene Prioritätsstufen (Windows: 1..31) die beim Scheduling berücksichtigt werden

13.3.3 THREADZUSTÄNDE

- LAUFEND (running) - läuft gerade in der CPU
- BEREIT (ready) - wartet in der Warteschlange auf CPU-Zeit
- WARTEND (waiting) - im Leerlauf, benötigt im Moment keine CPU-Zeit, kann erst durch einen anderen Thread wieder in den Zustand BEREIT

13.3.4 THREADS STARTEN

Eine gegebene Methode TestMethod, soll nebenläufig in einem neuen Thread abgearbeitet werden:

```
using System.Threading;
// ...
ThreadStart del = new ThreadStart(TestMethod);
Thread thread = new Thread(del);
thread.Start();
```

Alternativ: Thread starten mit Lambda-Expressions

```
Thread thread = new Thread( () => TestMethod() );
Thread thread = new Thread( () => TestMethod(param1, param2, ...) )
    ;
// Achtung: Parameter werden erst zum Start des Threads ausgewertet
    und dürfen sich bis dahin nicht ändern
```

13.3.5 BEISPIEL

```
using System.Threading;

class Program {
  static void Main(string[] args) {
    Thread thread = new Thread(new ThreadStart(TestMethod));
    thread.Start();
    for(int i = 0; i <= 100; i++) {
      for(int k = 1; k <= 5; k++)
        Console.Write(".");
      Console.WriteLine("Primär-Thread " + i);
    }
  }

  public static void TestMethod() {
    for(int i = 0; i <= 100; i++) {
      for(int k = 1; k <= 5; k++)
      Console.Write("X");
      Console.WriteLine("Sekundär-Thread " + i);
    }
  }
}
```

Ausgabe (Beispiel):

```
...XXXXXXSekundär-Thread 0
XX..Primär-Thread 0
.XXXSekundär-Thread 1
XX.. \\ usw.
```

13.3.6 WEITERE METHODEN VON THREAD

- `Thread.Sleep(int time)` Thread schläft für time Millisekunden
- `thread.Abort()` Thread bricht ab, nicht mehr unterstützt in .Net Core
- `thread.Join()` Blockiert den aufrufenden Thread (!=thread), bis der Thread thread beendet ist

13.4 SYNCHRONISATION VON THREADS

Problem: Mehrere Threads können auf gleiche Daten zugreifen

13.4.1 BEISPIEL

2 Threads, jeder soll die Variable V um 1 erhöhen

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

Sei anfänglich V = 0. Was passiert bei folgenden Abläufen?

- 1A, 2A, 3A, 1B, 2B, 3B
- 1A, 1B, 2A, 2B, 3A, 3B
- 1A, 2A, 1B, 3A, 2B, 3B

Es kann als Wert "V+1ßtatt "V+2"raus kommen, wenn beide den Wert "Väuslesen und dann darauf addieren.

13.4.2 DATENABHÄNGIGKEITEN

Einige Berechnungen beruhen auf vorher berechneten Daten $\rightarrow$ können erst ausgeführt werden wenn diese Daten berechnet sind.

Bernstein conditions: 2 Prozesse P_i, P_j sind unabhängig und können parallel ausgeführt werden, wenn alle Input (I) und Output (O) Variablen die folgenden Bedingungen erfüllen:

- $I_j \cap O_i = \varnothing$
- $O_j \cap I_i = \varnothing$
- $O_j \cap O_i = \varnothing$

Bsp: Können diese beiden Zeilen parallel ausgeführt werden?

```
c = func1(a, b) { ... };  d = func2(a, b) { ... };  // Ausgabevariablen unabhängig: ok
c = func1(a, b) { ... };  a = func2(a, b) { ... };  // Eingabevariable der nächsten Zeile ist abhängig von der Ausgabe der vorherigen: nicht ok
c = func1(a, b) { ... };  c = func2(a, b) { ... };  // Ausgabevariable gleich: nicht ok
```

13.4.3 RACE CONDITION (WETTLAUFSITUATION)

Das Ergebnis der Anwendung ist abhängig vom Timing der Threads → zufälliges, falsches Ergebnis
Treten auf, wenn datenabhängige Threads gleichzeitig ausgeführt werden
Fun Fact: Gibt es Studien zufolge auch im menschlichen Gehirn ... Grund für Zufälligkeiten

13.4.4 RACE CONDITIONS VERMEIDEN:

Der kritische Codeblock darf nur betreten werden wenn kein anderer Prozess darauf zugreift. (= mutual exclusion)

Funktioniert das vielleicht wie folgt?
Was passiert wenn 2 Threads (fast) gleichzeitig `addOne()` aufrufen?

```
static int flag = 0;
static int V = 0;
// thread-safe execution of V = V + 1

public void addOne() {
  do {
    if (flag == 0) {
      // lock the code section by setting a flag
      flag = Thread.CurrentThread.ManagedThreadId;
      V = V + 1;
      flag = 0;
      return;
    }
  } while (true);
}
```

Es können immer noch zwei Threads gleichzeitig in den kritischen Bereich kommen, wenn sie beide gleichzeitig das flag lesen, bevor der andere es geschrieben hat.

13.4.5 ATOMARE OPERATIONEN

- unzerteilbare Operation die nicht durch Ende der Zeitscheibe eines Threads auf der CPU unterbrochen werden können.
- können nur vollständig oder gar nicht stattfinden kann
- kleinste atomare Operation: Maschinenbefehl
- Hochsprachenbefehle sind nicht atomar, Bsp i++ (lesen, inkrementieren, schreiben)

- Bsp: Was wird ausgegeben?

```
static void Main(string[] args) {
  long counter = 0;
  Thread a = new Thread( () =>  {
    for (long i = 0; i < 100000; i++)
      counter++;
  });
  Thread b = new Thread( () =>  {
    for (long i = 0; i < 100; i++)
      counter++;
  });
  a.Start(); b.Start();
  a.Join(); b.Join();
  Console.WriteLine("{0}", counter);

 }
```

Ausgabe ist zufällig zwischen $100\,001$ und $100\,100$.

13.4.6 MUTUAL EXCLUSION (MUTEX)

= gegenseitiger Ausschluss
Race conditions werden vermieden dadurch dass zu jedem Zeitpunkt nur 1 Thread eine kritischen Codeabschnitt abarbeiten darf, der andere Thread muss solange warten bis der eine Thread fertig ist.
Realisierungen:

- lock
- semaphore
- monitor
- message passing

13.4.7 MONITOR

- Definiert kritische Codeabschnitte, die nur von einem Thread betreten werden dürfen.
- Ein kritischer Codeabschnitt wird durch eine Speicheradresse identifiziert, dazu kann ein Referenztyp (z.B. ein Object) verwendet werden, z.B. das object, wessen Zugriff kritisch ist.
- `Monitor.Enter(object)` = Beginn des kritischen Abschnitts.
- `Monitor.Exit(object)` = Ende des kritischen Abschnitts.

13.4.8 BEISPIEL

Was passiert nun wenn addOne() von 2 Threads gleichzeitig aufgerufen wird?

```
static object lockObject = new object();
static int V = 0;
// thread-safe execution of V = V + 1

public void addOne() {
  Monitor.Enter(lockObject);
  V = V + 1;
  Monitor.Exit(lockObject);
}
```

13.4.9 LOCK

```
lock (x){
  // ...
}
```

ist äquivalent zu

```
Monitor.Enter(x);
try {
  //...
} finally {
  Monitor.Exit(x);
}
```

13.5 DEADLOCK, STARVATION

Deadlock = Verklemmungssituation die bei ungünstiger Anwendung von Sperren (Locks, Barriers) auftreten kann
Chinese Chopstick Problem – Dining Philosophers Problem

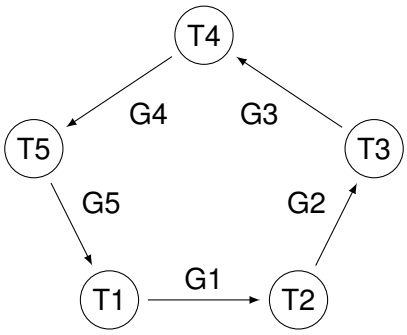
- jeder Philosoph braucht 2 Gabeln zum Essen
- Philosophen denken über die Probleme der Welt nach und essen wenn sie hungrig sind
- sie greifen zuerst nach der linken Gabel, dann nach der rechten, essen dann und legen die Gabeln danach zurück
- Problem: Was passiert wenn alle 5 gleichzeitig essen wollen?

Aus https://en.wikipedia.org/wiki/Dining_philosophers_problem

13.5.1 VISUALISIERUNG IM WARTEGRAPH (RESSOURCE DEPENDENCY GRAPH)

- statischer Graph stellt Zustand zu einem konkreten Zeitpunkt des Programmablaufs dar
- gerichteter Graph mit Threads als Knoten, Sperren als Kanten, Richtung: Besitzer der Sperre
- Zyklen im Graph bedeuten einen Deadlock

T1 ... T5: Philosophen/Threads
G1 ... G5: Gabeln/Ressourcen
Deadlock, wenn jeder seine linke Gabel nimmt und auf die rechte wartet:

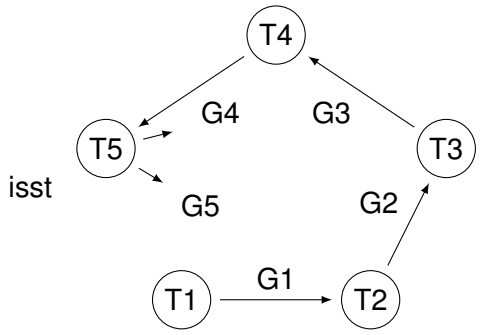


Beispielcode für T1:

```
Monitor.Enter(G5);
Monitor.Enter(G1);
// essen
Monitor.Exit(G1);
Monitor.Exit(G5);
```

13.5.2 VERHINDERN VON DEADLOCKS

- Reihenfolge von Sperranweisungen geschickter wählen
- stets gleiche Reihenfolge für Sperren
- Beispiel:
  Aufnahmepriorisierung: G1>G2>G3>G4>G5 (jeder Thread nimmt linke Gabel zuerst, außer Thread 1).



13.5.3 STARVATION (VERHUNGERN)

- Eine Aufgabe wird nie abgearbeitet (verhungert)
- Ursache: niedrige Priorität der Aufgabe, schnelle Erzeugung von neuen Aufgaben höherer Priorität
- z.B. immer neue Aufgaben werden generiert und in LIFO Queue abgelegt, nur die obersten werden abgearbeitet, die untersten verhungern
- Lösung: Prioritäts-Management der Aufgaben, FIFO Queue, ...

13.6 ASYNCHRONE METHODEN

13.6.1 ABGRENZUNG ZU MULTITHREADING

Beim Threading geht es um Arbeiter, bei Asynchronität um Aufgaben (Tasks)

ANALOGIE: Sie kochen in einem Restaurant, es werden Eier und Toast bestellt
Sie haben die Wahl wie Sie diese Bestellung ausführen:

- SYNCHRON: Sie kochen die Eier, wenn diese fertig sind machen Sie den Toast
- ASYNCHRON, EINZELNER THREAD: Sie kochen die Eier und starten eine Eieruhr, Sie schieben den Toast in den Toaster, während Sie warten reinigen Sie die Küche. Wenn die Wecker klingeln, nehmen Sie die Eier, bzw. den Toast raus
- ASYNCHRON, MULTITHREAD: Sie besorgen sich 2 weitere Köche, einer kocht die Eier der andere macht den Toast. Sie müssen die Köche koordinieren, damit sie in der Küche nicht die gleichen Küchengeräte gleichzeitig verwenden und sie bezahlen

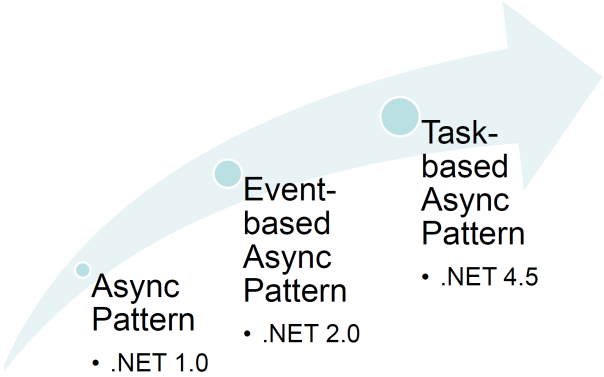
Viele Tasks belegen den Prozessor nicht dauerhaft mit Arbeit

- I/O Aufgaben
- Zugriffe auf Datenbanken, Internet

- Zugriff auf Peripherie

Für Prozessor-gebundene Aufgaben macht es Sinn Multithreading zu verwenden, für andere Aufgaben sind mehrere Threads nicht nötig, solange in der Wartezeit andere Aufgaben erledigt werden können.

13.6.2 ASYNC PATTERNS



13.6.2.1 EVENT-BASED ASYNCHRONOUS PROGRAMMING DESIGN PATTERN (EAP)

Das NET Framework stellt drei Muster zum Ausführen von asynchronen Vorgangen bereit:

- Das APM (ASYNCHRONOUS PROGRAMMING MODEL)-Muster (auch als IAsyncResult-Muster bezeichnet), bei dem asynchrone Vorgange die Begin-Methode und die End-Methode erfordern (z. B. BeginWrite und EndWrite für asynchrone Schreibvorgange). Dieses Muster wird für Neuentwicklungen nicht mehr empfohlen.
- Das EREIGNISBASIERTE ASYNCHRONE MUSTER (EVENT-BASED ASYNCHRONOUS PATTERN, EAP), das eine Methode mit dem Async-Suffix sowie ein oder mehrere Ereignisse, Ereignishandlerdelegattypen und von EventArg abgeleitete Typen erfordert. EAP wurde in .NET Framework 2.0 eingefuhrt. Es wird für Neuentwicklungen nicht mehr empfohlen.
- Das AUFGABENBASIERTE ASYNCHRONE MUSTER (TASK-BASED ASYNCHRONOUS PATTERN, TAP) stellt die Initiierung und den Abschluss eines asynchronen V organgs mit einer einzelnen Methode dar. TAP wurde in .NET Framework 4 eingeführt. Dieses ist das empfohlene Muster für die asynchrone Programmierung in .NET Framework.

13.6.2.2 VERGLEICHEN VON MUSTERN

Für einen schnellen Vergleich der Modellierung von asynchronen Vorgängen durch die drei Muster betrachtet man eine Read-Methode, die eine angegebene Menge von Daten ab einem angegebenen Offset in einen bereitgestellten Puffer liest:

```
public class MyClass{
  public int Read(byte [] buffer, int offset, int count);
}
```

Die APM-Entsprechung dieser Methode macht die BeginRead-Methode und die EndRead-Methode verfügbar:

```
public class MyClass{
  public IAsyncResult BeginRead(byte [] buffer, int offset, int
     count, AsyncCallback callback, object state) ;
```

```
  public int EndRead(IAsyncResult asyncResult);
}
```

Die EAP-Entsprechung macht den folgenden Satz von Typen und Membern verfügbar:

```
public class MyClass {
  public void ReadAsync(byte [] buffer, int offset, int count);
  public event ReadCompletedEventHandler ReadCompleted;
}
```

Die TAP-Entsprechung macht die folgende einzelne ReadAsync-Methode verfügbar:

```
public class MyClass {
  public Task<int> ReadAsync(byte [] buffer, int offset, int count)
      ;
}
```

13.6.3 ZUSAMMENFASSENDES BEISPIEL

Vorlesung 19.05.2017

Sei `double[] vec` ein gegebenes Array von Double Werten. Sie möchten gern die Summe aller Werte in `vec` berechnen.
Schreiben Sie ein Programm, dass diese Summel PARALLEL berechnet. Teilen Sie dafür die notwendige Schleife auf 2 Threads auf.
Idee: Schleife in der Mitte aufspalten: `i = 0,1,2, ..., n/2, n/2+1, ..., n-1, n.`

```
double summe = 0;
int n = vec.Length;

Thread t1  = new Thread(() => {
  for (long i=0; i < n/2; i++)
    summe += vec[i];
  }
);
Thread t2 = new Thread(() => {
  for (long i= n/2; i < n; i++)
    summe += vec[i];
  }
);

t1.Start();
t2.Start();
Console.WriteLine("Summe = {0} \n", summe);
```

2 Fehler:

- Summe muss zum Schreiben gelockt werden.
- Ausgabe passiert, bevor sichergestellt ist, dass Threads fertig sind.

Also:

```
double summe = 0;
object lockobj = new object();
int n = vec.Length;

```

```
Thread t1  = new Thread(() => {
  for (long i=0; i < n/2; i++)
    lock(lockobj) {
      summe += vec[i];
    }
  }
);
Thread t2 = new Thread(() => {
  for (long i= n/2; i < n; i++)
    lock(lockobj) {
      summe += vec[i];
    }
  }
);

t1.Start();
t2.Start();
t1.Join();
t2.Join();
Console.WriteLine("Summe = {0} \n", summe);
```

Fehler:

- Beide Threads tun nie etwas gleichzeitig, da der Lock auf Summe besteht.
  ⇒ Programm sogar langsamer!

Lösung:

```
double summe = 0, summe1 = 0, summe2 = 0;
int n = vec.Length;

Thread t1  = new Thread(() => {
  for (long i=0; i < n/2; i++)
    summe1 += vec[i];
  }
);
Thread t2 = new Thread(() => {
  for (long i= n/2; i < n; i++)
    summe2 += vec[i];
  }
);

t1.Start();
t2.Start();
t1.Join();
t2.Join();
summe = summe1 + summe2;
Console.WriteLine("Summe = {0} \n", summe);
```

Oder alternativ:

```
double summe = 0;
object lockobj = new object();
int n = vec.Length;
```

```
4
5 Thread t1  = new Thread(() => {
6   summe1 = 0;
7   for (long i=0; i < n/2; i++)
8     summe1 += vec[i];
9   lock (lockobj) {summe += summe1;}
10  }
11 );
12 Thread t2 = new Thread(() => {
13  summe2 = 0;
14  for (long i= n/2; i < n; i++)
15    summe2 += vec[i];
16  lock (lockobj) {summe += summe2;}
17  }
18 );
19
20 t1.Start();
21 t2.Start();
22 t1.Join();
23 t2.Join();
24 Console.WriteLine("Summe = {0} \n", summe);
```

13.7 TASK PARALLEL LIBRARY

- neu in .NET 4.0
- Im Zentrum steht die neue Klasse Task

13.7.1 TASKS VS THREADS

- zu viele Threads sind schlecht:
  - OS ist beschäftigt mit Scheduling.
  - jeder Thread bindet Speicher (im MB Bereich).
- Tasks können Threads erzeugen (max 1 Thread pro Task), wenn sie es für notwendig erachten, Concurrency Runtime sorgt für optimale Lastverteilung.
- Typische Probleme beim gleichzeitigen Zugriff auf Ressourcen (Race Conditions, Deadlock, ...) bleiben bestehen, müssen umgangen werden.

13.7.2 LAUFZEITVERGLEICH TASK / THREAD

Aufgabe: 64 Listen mit 2 000 000 Elementen sortieren

- 64 Threads: 14 sec (CPU Auslastung 60 %)
- 64 Tasks: 3 sec (CPU Auslastung 99 %)

Neue thread-sichere Datenstrukturen:

- `ConcurrentStack<T>`

- `ConcurrentQueue<T>`
- `BlockingCollection<T>`

13.7.3 DIE KLASSE TASK

- Anlegen und Verwendung ähnlich wie Thread
- wichtigste Zustände eines Task:

Task definieren	→	`Created`
Task starten	→	`WaitingToRun`
.NET startet den Task	→	`Running`
Task ohne Fehler abgearbeitet	→	`RanToCompletion`

13.7.3.1 TASK ANLEGEN

- mit lambda Functionen

```
Task t = new Task ( ()=> { /* ... */ } );
Task t = new Task ( (o) => { /* ... use o here */ }, /* value
    for o */ );
```

- ohne lambda Funktion

```
Task t = new Task( action );

Action<object> delegate = new Action<object>(DoWork);
Task t = new Task ( delegate, objectValue );

void DoWork( object o ){ /* ... */ }
```

13.7.3.2 TASK STARTEN

`Task.Start();`

13.7.3.3 TASK.FACTORY

`Task.Factory.StartNew( gleiche Parameter wie Konstruktor von Task );`
(startet den Task auch gleich, funktioniert ohne Konstruktor)

13.7.3.4 RÜCKGABEWERT EINES TASKS

```
// <int> ist der Typ des Rückgabewerts
Task<int> t = Task.Factory.StartNew(() => {
  Thread.Sleep(2000);
  return 13;
});
int i = t.Result;
// Code wartet, bis Task fertig ist und Zugriff auf Result möglich
    ist.
```

13.7.3.5 AUF EINEN TASK WARTEN

`t1.Wait();`

13.7.3.6 AUF MEHRERE TASKS WARTEN

`Task.WaitAll(Task[]); Task.WaitAny(Task[]);`
z.B. warten auf einen Task: `Task.WaitAny(new Task[] {t1, t2 } );`

13.7.3.7 PARALLELE SCHLEIFEN

- Die TPL stellt die Methoden `Parallel.For` und `Parallel.ForEach` zur Verfügung.
- intelligenter Aufteilungs-Algorithmus zerlegt Datenmenge, bzw. Schleifen-Interationen und erstellt selbstständig Tasks.

Beispiel:

```
// wie for (int i=0; i<5; i++):
Parallel.For(0,  5,  i => {
  Console.WriteLine("Durchlauf mit i = {0}", i);
  Thread.Sleep(1000);
});

Parallel.ForEach(new int[] {1, 2, 3, 4, 5}, item => {
  Console.WriteLine("Durchlauf mit i = {0}", item);
  Thread.Sleep(1000);
});
```

Wie könnte man die folgende Schleife parallelisieren ?

```
double result = 0;
for (int i = 0; i < vec1.GetLength(0) ; i++) {
  result += vec1[i] * vec2[i];
}
```

Lösung:

```
Parallel.For(0, vec.GetLength(0), i=> {
  result += vec1[i] * vec2[i];
});
```

Problem wieder ähnlich wie zuvor, dass Variable gelockt werden muss. Dafür gibt es dann besser lokale Schleifenvariablen:

13.7.3.8 LOKALE SCHLEIFENVARIABLEN

Jeder einzelne Thread hat eine eigene Instanz der Variable

- keine Sperre nötig
- am Ende können die lokalen Variablen zusammengeführt werden

```
// <TLocal>: Typ der lokalen Schleifenvariable
Parallel.For<TLocal> (
  Int32, // Startwert
  Int32, // Endwert
  Func<TLocal>, // Initialisierung der lokalen Schleifenvariable
  Func<Int32, ParallelLoopState, TLocal, TLocal>, //
     Iterationsvorschrift
  Action<TLocal> // Abschluss, nutze lokale Variable um globale
     Variable zu updaten
)
```

Beispiel:

```
Object myLock = new Object();
Parallel.For<double>(0, n, () => {return 0; },
  (i, pls, localSum) =>  {
    localSum += vec1[i] * vec2[i];
    return localSum;
  },
  x =>  {
    lock (myLock) result += x;
  }
);
```

13.7.3.9 SCHLEIFE ABBRECHEN

Nutzt das `ParallelLoopState` Objekt das übergeben wird

- `pls.Break();`
  Abbruch nachdem alle Iterationen bis zur aktuellen durchgeführt wruden
- `pls.Stop();`
  sofortiger Abbruch

13.7.3.10 PARALLELISIERUNG VON GUI AKTIVITÄTEN

- Die Controls der GUI gehört zu genau einem Thread, andere Threads haben keinen Zugriff auf GUI Aktivitäten
- wird ein Teil eines Programms auf einen anderen Thread ausgelagert, so müssen darin enthaltene GUI-Aktivitäten zurück an den GUI-Thread gegeben werden.
- Dies geschieht mit der Methode Invoke, die auf einem GUI Control aufgerufen wird

Beispiel:

```
Task.Factory.StartNew( () => {
  // aufwendige Berechnung hier
  this.Invoke(new Action(() => {
    label.Location = new Point(200,200);
  }));
}
```

13.7.3.11 WIEDERHOLUNG PARALLEL.FOR MIT LOKALEN SCHLEIFENVARIABLEN

Vorlesung 02.06.2017

Allgemeine Definition:

```
Parallel.For<T>(from, to, initialize, iterate, finalize)
```

Daraus macht der Compiler quasi den folgenden Code und führt ihn parallel(!) aus und spaltet die Schleife auf die Tasks auf:

```
T localResult = initialize();
for (long i=from; i<to; i++) {
  localResult = iterate(i, pls, localResult);
}
finalize(localResult);
```

Danach ist alles wieder seriell. Folglich ist auch `localResult` nicht mehr verfügbar.

BEISPIEL

Addiere Zahlen 1 bis 100:

```
int sum = 0;
Parallel.For<int>(1,101, () => 0,
  (i, pls, localSum) => localSum + i,
  (localSum) => lock(lockObj) { sum += localSum; }
);
```

VERSTÄNDNISTEST

Welchen Wert hat `result` am Ende?

```
Object myLock = new Object();
int result = 1;
Parallel.For<int>(10, 100, ()=> { return 0;}, % Initialisierung =>
   x=0
  (i, pls, x) => {
    x += (int)i % 2;
    return x;
  },
  z => {
    lock (myLock) result = (result + z) % 42;
  }
);
```

Lösung: 4 (result=1 + z=45 [45 gerade Zahlen]) % 42

13.8 ASYNCH/AWAIT

neu im .NET Framework 4.5

13.8.1 MODIFIZIERER ASYNC

- müssen ein await enthalten, ermöglichen dem Aufrufer an dieser stelle seine Arbeit fortzusetzen
- zulässige Rückgabewerte: `void`, `Task` oder `Task<TResult>`
- zulässige Parameter: kein `ref`, kein `out`

13.8.2 SCHLÜSSELWORT AWAIT

- Wartet auf Ausführung von Tasks
- Aufruf: `await TaskName`
- Ausdruck ist vom `void` wenn einfacher Task erwartet wird, vom Typ `TResult`, wenn `Task<TResult>` erwartet wird
- kann nur in einer Asynchronen Methode aufgerufen werden
- während des Wartens wird zum Aufrufer der asynchronen Methode zurückgekehrt
- ist der erwartete (await) Task fertig wird die asynchrone Methode über einen Rückruf an der Stelle der Unterbrechung fortgesetzt

Beispiel:

```
static void Main(string[] args) {
  DoSomethingAsync();
  for (int i = 0; i < 1000; i++)
    Console.Write(".");
  Console.ReadLine();
}

static async void DoSomethingAsync(){
  Console.Write("Start\n");
  Console.Write(await TestAsync());
  Console.WriteLine("Fertig\n");
}

static Task<string> TestAsync() {
  Task<string> t = new Task<string>(() => {
    Thread.Sleep(20); // lang dauernde Berechnung
    return "\n Das Ergebnis ist 42\n"; }
  );
  t.Start();
  return t;
}
```

Ausgabe:

```
Start
..........................  // viele Punkte
Das Ergebnis ist 42
Fertig
..........................  // viele Punkte
```

13.8.3 RÜCKGABEWERTE

Eine async Methode kann nur einen der folgenden Rückgabewerte haben

- `void`
- `Task`
- `Task<T>`

Dabei wird einzig für `Task<T>` ein return erwartet, dass einen Objekt vom Typ `T` zurückgibt. Es wird also kein Task explizit mit `return` zurückgegeben!

Der Aufrufer der async Methode erhält jedoch ein Task Objekt, dass genutzt werden kann um auf das Ergebnis der Methode zuzugreifen (es ist nicht der gleiche Task wie innerhalb der Methode!).

```
static void Main(string[] args) {
  Task<string> t = DoSomethingAsync();
  for (int i = 0; i < 1000; i++)
    Console.Write(".");
  Console.WriteLine(t.Result); // wartet auf DoSomethingAsync()
}

static async Task<string> DoSomethingAsync() {
  Task<int> t = new Task<int>(() => {
    Thread.Sleep(2000); // lang dauernde Berechnung
    return 42;
  });
  t.Start();
  int result = await t;
  string s = "Das Ergebnis ist" + result;
  return s;
}
```

13.8.4 RÜCKGABEWERTE MIT TUPEL

Tupel = generische Klasse, die Elemente in einer bestimmten Reihenfolge kapselt

Beispiel: Wir wollen einen string und einen int zurückgeben:

```
static async Task<Tuple<string, int>> TestAsync() {
  // ...
  return new Tuple<string, int>("Fertig",1);
}
```

Es gibt in .NET viele vordefinierte erwartbare (await-able) Methoden, z.B. für

- Internetzugriff
- Lesen und Schreiben von Dateien

```
class program {
  static bool finished = false;

  static async void ReadFromNetwork(string url)  {
```

```
    HttpClient client = new HttpClient();
    Console.WriteLine("Requesting the data . . .");
    var sw = Stopwatch.StartNew();
    string result = await client.GetStringAsync(url);
    finished = true;
    Console.WriteLine("{0}", result);
    Console.WriteLine("Finished in " + sw.ElapsedMilliseconds + "ms
        .");
    sw.Stop();
  }

  static void Main(string[] args)  {
    var  t = ReadFromNetwork("http://www.textfiles.com/music/kurt.
       txt");
    while (!finished)
      Console.Write(".");
    Console.ReadLine();
  }
}
```

Ausgabe:

```
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
.......................................................................
...............            BoddAH  <UNREADABLE>

Speaking from the tongue of an experienced simpleton who
obviously would rather be an emasculated, infantile
complaind. This note should be pretty easy to understand.
All the warnings from the Punk Rock 101 Courses over the
years, it's my first introduction to the, shall we say ethics
involved with independence and the embracement of your
community has been proven to be very true.  I haven't
felt the excitement of listening to, as well as creating
```

14 ITERATOREN: IENUMERABLE, IENUMERATOR, YIELD RETURN

Ziel: Durchlaufen von Elementen mit einer foreach Schleife
Im Programm soll aber eine einfache foreach Schleife definiert werden können bei der irgendeine Sammlung von Daten durchlaufen wird.

BEISPIEL

```
foreach (string s in StringList) {
  Console.WriteLine(s);
}
```

Die Sammlung der Daten die zu Durchlaufen sind, soll in StringList enthalten sein.
Welche möglichen Datentypen für StringList kennen Sie bis jetzt?
Nun wollen wir das erweitern: StringList soll z.B.

- aus einer Datenbank
- aus einer Datei
- aus dem Internet

lesen, und zwar dynamisch (erst bei Abfrage des nächsten Wertes)
Vorteil, sie brauchen das Auslesen nicht in ihrer Schleife implementieren, sondern in StringList (Trennung von Daten und Iteration)

Frage: Was muss StringList erfüllen damit das funktioniert?
Antwort: Es muss das Interface IEnumerable implementieren!

14.1 IENUMERABLE/IENUMERATOR

IEnumerable liefert die Methode GetEnumerator, die benutzt werden kann um eine Auflistung zu durchlaufen

```
public interface IEnumerable {
  IEnumerator GetEnumerator();
}
```

IEnumerator steuert das durchlaufen einer Aufzählung (entspricht einem Iterator in C++/Java)

```
public interface IEnumerator {
  T Current { get; }
  // Ruft das aktuelle Element in der Auflistung ab.
  bool MoveNext();
  // Setzt den Enumerator auf das nächste Element der Auflistung.
  // Rückgabewerte:
```

```
  // true, wenn der Enumerator erfolgreich auf das nächste Element gesetzt wurde,
  // false, wenn der Enumerator das Ende der Auflistung überschritten hat.

  void Reset();
  // Setzt den Enumerator auf seine anfängliche Position vor dem ersten Element
  // in der Auflistung.
}
```

Wie bekommt man so einen Iterator (IEnumerator)?

- Man schreib ihn selbst
- Man nutzt yield return

14.2 YIELD RETURN

yield return funktioniert ähnlich wie return in einer Methode

- Wird verwendet um jedes Element einer Auflistung einzeln zurück zu geben
- Der Rückgabewert der Methode ist damit automatisch ein Iterator (IEnumerator)
- Wenn eine yield return-Anweisung erreicht wird, wird ein Wert zurückgegeben, und die aktuelle Position im Code wird gespeichert. Wenn die Iteratorfunktion das nächste Mal aufgerufen wird, wird die Ausführung von dieser Position fortgesetzt

Beispiel:

```
static void Main(string[] args) {
  DayList daylist = new DayList();
  foreach (string s in daylist) { Console.WriteLine(s); }
}

public class DayList {
  public IEnumerator GetEnumerator() {
    yield return "Montag";
    yield return "Dienstag";
    yield return "Mittwoch";
    yield return "Donnerstag";
    yield return "Freitag";
  }
}

// oder äquivalent dazu:
public class DayList {
  string[] tage = { "Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag" };
  public IEnumerator GetEnumerator() {
    for (int i = 0; i < tage.Length; i++)
    yield return tage[i];
  }
}
```

Verständnistest: Wie würde man die Klasse DayList ohne yield return definieren?
→ Enumerator muss selbst erstellt werden:

```
public class Daylist : IEnumerator{
  int i = -1;
  string[] tage = { "Montag", "Dienstag", "Mittwoch", "Donnerstag",
      "Freitag" };

  public object Current {
    get {
      return tage[i];
    }
  }

  public bool MoveNext() {
    i++;
    if (i>4)
      return false;
    return true;
  }

  public void Reset() {
    i = -1;
  }

  public IEnumerator GetEnumerator(){
    return this;
  }
}

static void Main(string[] args){
  Daylist daylist = new Daylist();
  foreach (string s in daylist) Console.WritLine(s);
}
```

14.2.1 GENERISCHER IENUMERATOR

IEnumerator und IEnumerable gibt es auch als generische Typen:
`IEnumerator<T>, IEnumerable<T>`

BEISPIEL

Sie wollen eine Datei in einer foreach-Schleife Zeile für Zeile durchlaufen.
Dabei sollen nur Zeilen mit Inhalt (Length>1) berücksichtigt werden

```
int i = 0;
foreach (string line in new ExtractLinesFromFile(@"C:\eula.1028.txt
    ")) // @: es werden keine Escape-Charaktere behandelt (bspw. \n
    )
  Console.WriteLine("Zeile " + i++ +": " + line + "\n");

```

```
public static IEnumerable<string> ExtractLinesFromFile(string
    filename){
  StreamReader reader = new StreamReader(filename);
  string line = null;
  while ((line = reader.ReadLine()) != null){
    if (line.length > 1)
      yield return line;
  }
}
```

15 C# FEATURES: ATTRIBUTE, REFLECTIONS, EXTENSION METHODS

Vorlesung
09.06.2017

15.1 ATTRIBUTE

- sind Zusatzinformationen (Metadaten) zu Klassen, Methoden und Eigenschaften
- können zur Laufzeit ausgelesen werden mit einem Verfahren, dass Reflection heißt
- Syntax:
  `[Attributname]`
  `Typ, Klasse, Methode, Eigenschaft`

Beispiel für Attribute:

- `[Obsolete]`
- `[Obsolete("Nutzen Sie die neue Methode newMethod()")]`

15.1.1 ATTRIBUT DEFINIEREN

```
public class Author : System.Attribute {
  public string Name { get; set; }
  public Author(string name = "anonymous") {
    Name = name;
  }
}
```

15.1.2 ATTRIBUT NUTZEN

```
[Author("S.Aland")]
class MyClass {
  // ...
}
```

15.1.3 REFLECTION (ATTRIBUTE ABFRAGEN)

```
Attribute[] attrs = Attribute.GetCustomAttributes(typeof(MyClass));
foreach (System.Attribute attr in attrs) {
  if (attr is Author) {
    Author a = (Author)attr;
    Console.WriteLine("Author: " + a.Name);
  }
}
```

15.2 EXTENSION METHODS

fügen Methoden zu bestehenden Typen hinzu

Beispiel Sie möchten die Reihenfolge der Einträge einer Liste umkehren können, also eine neue Methode `Invert()` hinzufügen.

15.2.1 VERFAHREN

1. Hilfsklasse erstellen, Liste übergeben

```
List list;

Helpers.Invert(list);
```

   Bewertung: umständliche Hilfsklasse

2. Von der Klasse erben

```
class MyList : List

MyList list = ...
list.Invert();
```

   Bewertung: umständliche Vererbung, teils Vererbung nicht möglich (bei versiegelten Klassen, von denen nicht geerbt werden darf, bspw. `string`)

3. Methode direkt in Klasse List implementieren

```
List list;

list.Invert();
```

   unschön vorgegebene Klasse zu ändern!

4. ⇒ Extensions Methods nutzen, um `Invert()` zur Klasse List hinzuzufügen ohne List direkt zu ändern

```
List list;

list.Invert();
```

   Bewertung: gute Lösung

Am „schönsten“ ist also die letzte Methode mit einer Extension Methode
Aufruf:

```
List<int> myList = new List<int>() { 1, 2, 7, 8 };
myList.Invert();

for (int i = 0; i < tlist.Count; i++)
  Console.WriteLine(tlist[i]); // writes 8 7 2 1
// ...
```

Funktioniert durch Hinzufügen von

```
// um eine Extension Method zu finden, werden alle statischen Klassen mit statischen Funktionen durchsucht und geguckt, was passt
static class Extension { // statische Klasse mit statischer Methode! Name egal.
  public static void Invert<T>(this List<T> list){  // this als erstes Argument: Indikator, dass es sich um eine Extension Method handelt
    for (int i = 0; i < list.Count / 2; i++) {
      T temp = list[i];
      list[i] = list[list.Count {}- i {}- 1];
      list[list.Count {}- i {}- 1] = temp;
    }
  }
}
```

15.2.2 DEFINITION

- statische Methode in einer statischen Klasse
- erstes Argument: this ... gibt den Typ an, auf den die Extension Methode aufgerufen wird
- Name der Klasse die die Extension method definiert ist egal, die Klasse wird nie benutzt

Beispiel eine Zahl auf jedes Element eines Arrays addieren
Aufruf:

```
int[] a = new int[] { 1, 2, 3 };
a = a.Add(2);
for (int i=0; i<a.Length; i++)
  Console.WriteLine(a[i]); // writes: 3 4 5
```

Funktioniert durch Hinzufügen von:

```
public static class Extension {
  public static int[] Add(this int[] a, int b) {
    for (int i = 0; i < a.Length; i++)
    a[i] += b;
    return a;
  }
}
```

Fazit: Cooles Feature, Hauptvorteil: Lesbarkeit, Klarheit

16 LINQ

16.1 NUTZEN VON LINK

- Verschiedene **unterschiedliche Datenquellen** (Datenbanken, XML-Dokumente, …)…
  - …**bedingen Kenntnis über verschiedene (Abfrage-)Sprachen (SQL, XPath, XQuery, …) und APIs** bzw. deren Implementierungen (DOM → System.Xml, Simple API for XML, …)
    - Eine Programmiersprache wie C# alleine reicht nicht aus, um mit diesen Daten umzugehen

      → **Kenntnis der Programmiersprache UND entsprechender Abfragesprache(n)/APIs nötig**
  - …gehen meist mit **Inkompatibilitäten zwischen den verschiedenen Systemen** einher („impedance missmatch“)
    - Grund sind die zugrundeliegenden unterschiedliche Ansätze bzw. (Daten-)Modelle:
      - **OOP vs. relationaler Ansatz**
        - unterschiedliche oder nicht vorhandene Datentypen (String Länge begrenzt in Datenbank vs. „unbegrenzte“ Länge usw.)
        - keine Vererbung (relationaler Ansatz) vs. Vererbung/Klassenhierarchie (OOP)
        - unterschiedliche Programmiermodelle (deklarativ vs. Imperativ)
          - C# Programmierer wird gezwungen mit Relationen (Tabellen) und Ergebnismengen zur Arbeiten
      - **XML**
        - Hierarchisches Datenmodell (Eltern/Kind-Beziehung der Elemente)
        - Daten gespeichert als Text
        - Deklarative Anfragen via XPath und XQuery bzw. DOM/SAX-APIs

- (Datenbank-)Abfragen werden meist nur in Form von Strings ausgedrückt (vgl. auch [Marguerie et al. 2008, S. 30ff], [msdn_1])
  - **Eingebettetes SQL** (Programmcode, d.h. „Nicht SQL“ wird mit SQL vermischt)

```
SqlCommand command = connection.CreateCommand();
command.CommandText = @"SELECT Vorname, Name FROM Kunde WHERE Ort = 'Dresden'";
```

    - *Prüfung von Syntaxfehlern im Abfrage**string** beim kompilieren? → KEINE! (Erst zur Ausführung → Laufzeit)*

```
SELECT Vorname, Name FROM Kunde WHER Ort = 'Dresden'
```

    - *Typprüfung im Abfrage**string** beim kompilieren? → KEINE! (Erst zur Ausführung → Laufzeit)*

```
SELECT Vorname, Name FROM Kunde WHERE Ort = 123
```

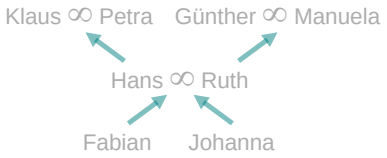
> **Hinweis**
>
> C# stellt für den „klassischen“ Datenbank- und XML-Zugriff mit ADO.NET (in der .NET-Klassenbibliothek enthalten) eine entsprechende Schnittstelle bereit (vgl. [msdn_2], [msdn_3], [Doberenz, Gewinnus 2008, S. 74])

- LINQ…
  - wurde **mit .NET 3.5** (bzw. C# 3.0 und Visual Studio 2008) 2007 **eingeführt** vgl. [Doberenz, Gewinnus 2008, S. 72], [wiki_1]
  - ist eine in (OO-Programmier-)Sprachen integrierte Abfragesprache (**L**anguage **IN**tegrated **Q**uery)
    - **C#**
    - **VB.NET**
    - Delphi Prism (Delphi .NET)
    - …
  - ermöglicht **deklarativ**e Abfragen
    - d.h. Angabe **welches Ergebnis** gewünscht ist (WAS) und nicht WIE ich dieses Ergebnis erreiche (imperativ)
  - bietet **typsichere Abfragen**
    - Es wird mit Objekten gearbeitet und nicht nur mit Strings vgl. auch [Westphal, Weyer 2007, S. 189]
      - Syntax- und Typfehler können beim kompilieren geprüft werden
  - ermöglicht einen **Zugriff auf unterschiedliche Elemente** (Objekte, Datenbankdatensätze, XML-Knoten, ..) **in einer einheitlichen Syntax** unabhängig von der Herkunft der Elemente vgl. [Pialorsi, Russo 2008, S. 24]

16.2 BEISPIELSZENARIO

Eine **Person** hat

- einen **Namen**
- ein **Alter**
- ein **Geschlecht**
- ist (nicht) **verheiratet** mit einer anderen Person
- hat 0…n Kinder

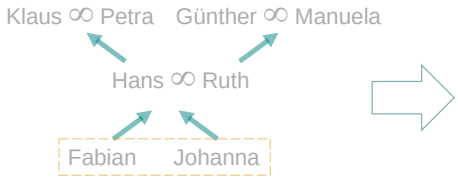


- Klaus ist **verheiratet mit** Petra
- Günther ist **verheiratet mit** Manuela
- Hans ist das **Kind von** Klaus und Petra
- Ruth ist das **Kind von** Günther und Manuela
- ….

Die Beziehung zwischen Elternteil und Kind soll über **isParentOf** dargestellt werden.

Uns interessieren hierbei nur Kinder aller verheirateten Personen unter 30 mit mehr als einem Kind.

→ **Im Beispiel sind das die Personen Hans und Ruth**



- Hans **isParentOf** Fabian
- Hans **isParentOf** Johanna
- Ruth **isParentOf** Fabian
- Ruth **isParentOf** Johanna

16.2.1 OHNE LINQ

```
// ...
List<Person> persons = new List<Person>();
List<IsParentOf> isParent = new List<IsParentOf>();
// ...
foreach (var person in persons)
{
  // Alle VERHEIRATETEN Personen unter 30 mit MEHR ALS EINEM Kind
  if (person.Age < 30 && person.marriedWith != null && person.Children.Count > 1)
  {
     foreach (var child in person.Children)
     {
       isParent.Add(new IsParentOf(person, child));
     }
  }
}
// ...
foreach (var parentRel in isParent)
{
          Console.WriteLine("{0} ist Elternteil von {1}",parentRel.parent.Name,parentRel.child.Name);
}
```

16.2.2 MIT LINQ

```
 // ...
 List<Person> persons = new List<Person>();
// ...

var isParent =  from person in persons
                where person.Age < 30 && person.marriedWith != null && person.Children.Count > 1
                from child in person.Children
                select new IsParentOf(person, child);

foreach (var parentRel in isParent)
{
        Console.WriteLine("{0} ist Elternteil von {1}",parentRel.parent.Name,parentRel.child.Name);
}
```

16.3 LINQ ABFRAGE

Eine **LINQ-Abfrage** besteht aus folgenden drei Schritten:

vgl. auch [Frischalowski 2008, S. 527], [msdn_4]

- Die **Datenquelle** (engl. data source) **auswählen**
  - Datenquelle besteht aus einer Folge (**Sequenz**) von Elementen
  - Über diese Sequenz muss iteriert werden können (z.B. Array oder Liste)
    - **Jede Datenquelle muss die IEnumerable- bzw. IEnumerable<T> Schnittstelle implementieren**, **oder abgeleitete Schnittstellen, wie IQueryable<T>**, um mit LINQ abgefragt werden zu können

      vgl. [Westphal, Weyer 2007, S. 192], [Pialorsi, Russo 2008, S.24, S. 30]

    - Alles was IEnumerable<T> implementiert wird in LINQ auch mit dem Begriff **Sequenz** bezeichnet
    - Nicht generische Collections (z.B. System.Collections.ArrayList) können in generische Collections (die IEnumerable<T> implementieren) mittels des Cast-Operators kann konvertiert werden, damit eine LINQ-Abfrage möglich ist (Cast gehört zu den sogenannten Konvertierungsoperatoren)
- Die eigentliche **LINQ-Abfrage** auf den Datenelementen **erstellen/spezifizieren**
- **Ausführung** der Abfrage

16.3.1 SYNTAX

Eine **LINQ-Abfrage** lässt sich unterschiedlich formulieren:

- **Abfrage-Syntax** (engl. query syntax)
  - **Syntaxerweiterung** der Programmiersprache (C#, VB.NET,…) um SQL ähnliche Schlüsselwörter
  - syntaktisch ähnlich zu SQL
  - deklarativ
  - „syntaktischer Zucker“ (Syntaxerweiterung zur Vereinfachung der Schreibweise)
- **Erweiterungsmethoden-Syntax** (engl. method syntax, method chain syntax)
  - Basiert auf **Methodenaufruf**en
  - **Aufruf von Erweiterungsmethoden** (aus dem Namespace System.Linq), also statischen Methoden, **um eine Abfrage auszudrücken**
  - Diese Erweiterungsmethoden werden in LINQ auch als **Abfrageoperatoren** bezeichnet
  - Vordefinierte Abfrageoperatoren (Select, Where, …) werden als **Standard-Abfrageoperatoren** bezeichnet vgl. [Marguerie et al. 2008, S.114]
  - **Standard-Abfrageoperatoren arbeiten mit Sequenzen**, erwarten also ein Objekt, welches IEnumerable<T> implementiert als Parameter vgl. [Marguerie et al. 2008, S.114]
- **gemischte Schreibweise**
  - Abfrage-Syntax in Verbindung mit Erweiterungsmethoden

16.3.2 GEGENÜBERSTELLUNG

Abfrage-Syntax (C#-Syntax)

```
// Alle Personen unter 30 Jahren ermitteln
// ABFRAGE-SYNTAX

var queryStmt = from person in persons
                where person.Age < 30
                select person;
```

Lesbarer als Erweiterungsmethoden-Syntax

Nicht jede Abfrage lässt sich im Abfrage-Syntax angeben (dazu später mehr)

Erweiterungsmethoden-Syntax

```
// Alle Personen unter 30 Jahren ermitteln
// ERWEITERUNGSMETHODEN-SYNTAX

var exprMethStmt = persons.Where(x => x.Age < 30).Select(x=>x);
// alternativ
var exprMethStmt = persons.Where(x => x.Age < 30);
```

Kombination/Verkettung der Erweiterungsmethoden durch „Dot-Notation" (**Pipeline-Prinzip**) ermöglicht volle Flexibilität bei der Erstellung von Abfragen (Verarbeitungsschritte sind gut nachvollziehbar)

Hinweis

Im Abfrage-Syntax ist die Verwendung von select vorgeschrieben. Im **Erweiterungsmethoden-Syntax ist die Angabe von Select optional**. vgl. [Kühnel 2013]

gemischte Schreibweise

```
var mixedStmt = from person in persons.Where(x => x.Age < 30)
                select person;
```

Abfrage-Syntax / **gemischte Schreibweise** → Werden in Methodenaufrufe kompiliert → **Erweiterungsmethoden-Syntax**

Hinweis

Egal für welche Notation man sich entscheidet, am Ende erzeugt der Compiler stets automatisch Aufrufe der Standard-Abfrageoperatoren (Erweiterungsmethoden-Syntax). vgl. [Marguerie et al. 2008, S.116], [Pialorsi, Russo 2008, S. 41]

16.3.3 SCHLÜSSELWÄRTER UND (STANDARD-)ABFRAGEOPERATOREN

Schlüsselwörter (C#)

- **from-Klausel**
- **where-Klausel**
- **select-Klausel**
- group-Klausel
- into-Klausel
- orderby-Klausel
- join-Klausel
- let-Klausel

Die Schlüsselwörter im VB.NET Syntax unterstützen mehr Abfrageoperatoren

Nicht jede Abfrage lässt sich im Abfrage-Syntax angeben (z.B. Aufruf von Count nicht möglich)

(Standard)-Abfrageoperatoren

Davon gibt es 51 Stück
vgl. auch [Marguerie et al. 2008, S. 154]

- **Where** (Einschränkungs-/Filteroperator)
- Projektionsoperatoren
  - **Select**
  - SelectMany
- Sortieroperatoren
  - OrderBy
  - OrderByDescending
  - …
- Gruppierungsoperatoren
  - GroupBy
- Verknüpfungsoperatoren
  - Join
  - GroupJoin
- Aggregierungsoperatoren
  - Count
  - LongCount
  - …
- … (Vollständige Übersicht + Quelle siehe Anhang A)

16.3.4 STRUKTUR

- **from-Klausel**
- **where-Klausel**
- **select-Klausel**
- **group-Klausel**
- into-Klausel
- orderby-Klausel
- join-Klausel
- let-Klausel

LINQ-Abfrage im Abfrage-Syntax:

- beginnt mit **from**
- endet mit **select** oder **group by**
- **into** (optional) verbindet Abfragen
  - Ergebnisse einer Abfrage dienen als Input für eine andere Abfrage (query continuation)
- **Schlüsselwörter sind „case sensitive“**, d.h. z.B. nur from ist zulässig und nicht FROM oder From

vgl. auch [Pialorsi 2007, S. 51]

16.3.5 FROM UND SELECT – DATENQUELLE UND AUSWAHL

from

- **Spezifiziert die Datenquelle** einer (Unter-)Abfrage, d.h. das Objekt, auf welches weitere LINQ-Operationen angewandt werden können
  - **Datenquelle muss eine Sequenz sein**, d.h. IEnumerable<T> implementieren
- Jede from-Klausel erzeugt eine sogenannte Bereichsvariable (engl. range variable). Wenn die Abfrage ausgeführt wird, entspricht sie den einzelnen Elementen in der Datenquelle
  - Bereichsvariable durchläuft die Elemente der Datenquelle

select

- Hinter select wird angegeben, welche Elemente/Daten als Ergebnis zurückgeliefert („projiziert“) werden
- Elemente werden aus der Menge zurückgeliefert, die durch Auswertung vorhergehender Klauseln entstanden ist
- Abfrageergebnisse werden über ein Objekt verfügbar gemacht, welches IEnumerable<T> implementiert

```
List<Person> persons = new List<Person>();
…
// Einfachste Abfrage im Abfrage-Syntax
var queryStmt = from person in persons
                select person;
```

Die Abfrage wird hierbei noch nicht ausgeführt! Wir haben nur die Datenquelle ausgewählt und die Abfrage spezifiziert

```
List<Person> persons = new List<Person>();
…
// Einfachste Abfrage im Abfrage-Syntax
var queryStmt = from person in persons
                select person;
```

⟺

```
List<Person> persons = new List<Person>();
…
// Einfachste Abfrage im Erweiterungsmethoden-Syntax
var queryStmt = persons.Select(person => person);
```

SELEKTOR

Funktion, die auf jedes Element der Quellsequenz angewendet wird

- **Schlüsselwort var** - implizit typisierte lokale Variable (seit C# 3.0)
  - Deklaration einer Variablen ohne Typangabe
    - **Typinferenz**
      - Typ der Variablen wird (vom Compiler) aus zugewiesenem Ausdruck abgeleitet
  - Wichtig für Rückgabe anonymer Typen durch select
    - **Ohne var könnten keine anonymen Typen verwendet werden** (er hat ja keinen Typ Namen, der angegeben werden könnte, der Compiler erzeugt trotzdem einen echten Typ)

vgl. [Marguerie et al. 2008, S. 93ff], [Pialorsi 2007, S. 32]

- **Erweiterungsmethoden**
  - Erleichtern Verkettung von Methoden, **Pipeline-Prinzip** ist Basis für LINQ-Abfragen und möglich, weil Ergebnistyp der Erweiterungsmethode d. erweiterte Typ selbst ist (z.B. IEnumerable) vgl. [Pialorsi 2007, S. 40ff]

Selektor: `person => person` (Delegate) entspricht hier `SELECT *`

BEISPIEL

- Warum beginnt eine LINQ-Abfrage im Abfrage Syntax mit from (Auswahl der Datenquelle)?
- Welcher Typ könnte wohl von der folgenden LINQ-Abfrage zurückgegeben werden und warum?

```
var queryStmt =  from person in persons
                 select person;
```

Liste vom Typ „Person“ als Datenquelle

- Wenn im Vornherein die Datenquelle bekannt ist, kann IntelliSense entsprechend der Quelle Vorschläge machen.
- Ergebnis ist vom Typ `IEnumerable<Person>`

16.3.6 SELEKTOR

Vorlesung 16.06.2017

```
List<Person> persons = new List<Person>();
…
// Einfachste Abfrage im Erweiterungsmethoden-Syntax
var queryStmt = persons.Select(person => person);
```

Was wird in Ergebnismenge übernommen („projiziert“) → **SELEKTOR** Funktion, die auf jedes Element der Quellsequenz angewendet wird

Allgemeine Methodensignatur:

```
public static IEnumerable<S>  Select<T,S>(this IEnumerable<T> source, Func<T, S> selector);
```

Wiederholung Func-Delegate und Generics:
- Func ist ein vordefiniertes (namespace System) generisches Delegate
- Func hat >= 0 und <=16 Input Parameter und immer 1 Output Parameter (der letzte Parameter wird als Output Parameter betrachtet)
- Func<T, S> hat T als Input und liefert S als Output

Speziell:

```
public static IEnumerable<Person>  Select<Person, Person>(
                                   this IEnumerable<Person> source, Func<Person, Person> selector);
```

16.3.7 UNTERABFRAGEN

- Auf from können weitere froms folgen
- **from spezifiziert Datenquelle einer Abfrage oder Unterabfrage** vgl. auch [msdn_6]
- Zusammengesetzte Abfragen mit from (engl. compound from)
  - Fall 1 – **abhängige Datenquellen**
    - Element in der Quellsequenz ist selbst eine Sequenz oder enthält eine Sequenz
    - Beispiel: Person hat Kinder (Children-Property vom Typ List<Person>) → Enkel aller Personen ermitteln

```
List<Person> persons = new List<Person>();
…
// Enkel
var grandchilds = from child in persons
                  from ch in child.Children // Kinder
                  from c in ch.Children // Kinder d. Kinder (Enkel)
                  select c;
```

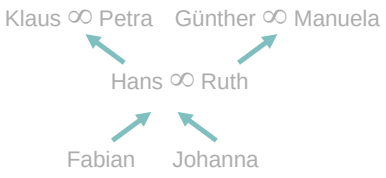
select hat Zugriff auf die Bereichsvariablen „child“, „ch“ und „c“

  - Fall 2 – **unabhängige**/verschiedenartige **Datenquellen**
    - Beispiel: Personen werden Fahrzeuge zugeordnet

BEISPIEL

- Welche Elemente enthalten die Ergebnismengen? (bei späterer Ausführung)

```
// Kinder
var childs =   from person in persons
               from ch in person.Children
               select ch;

// Enkel
var grandchilds = from child in persons
                  from ch in child.Children // Kinder
                  from c in ch.Children // Kinder d. Kinder (Enkel)
                  select c;
```



Antwort Kinder:
8 Elemente:

- Hans hat 2 Kinder,
- Ruth hat 2 Kinder,
- Klaus hat 1 Kind,
- Petra hat 1 Kind,
- Günther hat 1 Kind,
- Manuela hat 1 Kind

(die entstandenen Duplikate könnten mit `DISTINCT` entfernt werden.

Antwort Enkel:
8 Elemente

- Klaus, Petra, Günther und Manuela haben je zwei Enkelkinder

16.3.8 SELECT VS SELECTMANY

- **Select**
  - Das von Select zurückgegebene **Objekt vom Typ IEnumerable** enthält die Elemente. die durch die Anwendung des **SELEKTORS** auf jedes Element der Quellsequenz entstehen.
    - Beispiel: Person hat Kinder (Property **Children** liefert Element **vom Typ IEnumerable**, welches Objekte vom Typ Person enthält). D.h. **Anwendung des Selektors liefert jeweils IEnumerable<Person>**.
    - Selektor (*Func<T, S> selector*) liefert für EINEN Input vom Typ T (hier Person) EINEN Output vom Typ S (hier IEnumerable<Person>) in einer Sequenz vom Typ IEnumerable zurück

```
List<Person> persons = new List<Person>();
…
// Kinder im Abfrage-Syntax
IEnumerable<IEnumerable<Person>> childrenSelectqueryStmt = from person in persons
                                                           select person.Children;
```

```
List<Person> persons = new List<Person>();
…
// Kinder im Erweiterungsmethoden-Syntax
IEnumerable<IEnumerable<Person>> childrenSelectqueryStmt = persons.Select(p => p.Children);
```

- **SelectMany**
  - „Verflacht“ Menge von Sequenzen zu einer Sequenz
  - Wird bei mehreren from-Klauseln im Abfrage-Syntax benutzt vgl. [Griffiths 2012, S. 357ff]
  - **Selektor** liefert eine Sequenz zurück (und nicht nur einen Wert)
  - Selektor (*Func<T, **IEnumerable<S>**> selector*) liefert für EINEN Input vom Typ T (hier Person) EINEN Output vom Typ IEnumerable<S> (hier IEnumerable<Person>). Die einzelnen Elemente dieser „Zwischensequenz“ werden in einer Sequenz vom Typ IEnumerable zurückgeliefert → **Selektiert mehrere Elemente** (deshalb SelectMany) **für einen Input vom Typ T.** Die Zwischensequenzen selbst werden zu einer Ergebnismenge vereinigt vgl. [Griffiths 2012, S. 357ff]

```
List<Person> persons = new List<Person>();
…
// Kinder im Abfrage-Syntax
IEnumerable<Person> childrenSelectqueryStmt = from person in persons
                                              from ch in person.Children
                                              select ch;
```

Statt IEnumerable<IEnumerable<Person>>

```
List<Person> persons = new List<Person>();
…
// Kinder im Erweiterungsmethoden-Syntax
IEnumerable<Person> childrenSelectqueryStmt = persons.SelectMany(p => p.Children).Select(p=>p);
```

Selektor

- **SelectMany**
  - „In query expression syntax, each **from** clause … after the initial one translates to an invocation of SelectMany...“ [msdn_12]

```
// Abfrage-Syntax
from person in persons
from ch in person.Children
select ch;
```

```
// Erweiterungsmethoden-Syntax
persons.SelectMany(p => p.Children).Select(p=>p);
```

SelectMany vereinigt die „Zwischensequenzen“ (p.Children) zu **EINER Ergebnismenge**
vgl. [Griffiths 2012, S. 357ff]

Hat Zugriff auf die Bereichsvariablen „person“ und „ch“
(select gehört zur einleitenden from-Klausel)

```
persons.SelectMany(p => p.Children).Select(p=>p);
```

(Erweiterung) IEnumerable<Person> IEnumerable<Person>.SelectMany<Person,Person>(Func<Person,IEnumerable<Person>> selector) (+ 3 Überladung(en))
Projiziert jedes Element einer Sequenz in ein System.Collections.Generic.IEnumerable<T> und fasst die resultierenden Sequenzen in einer einzigen Sequenz zusammen.

Ausnahmen:
System.ArgumentNullException

16.3.9 WHERE

- Einschränkungs- bzw. **Restriktionsoperator**
- Filtert Werte heraus, die bestimmte Kriterien erfüllen
  - Nutzt dafür ein „predicate-delegate“
    - Methode, die true oder false zurück liefert
  - **Datenquelle muss eine Sequenz sein**, d.h. IEnumerable<T> implementieren
  - Ergebnis der Filterung ist eine Quellsequenz (implementiert IEnumerable<T>), die nur die gefilterten Werte enthält

```
List<Person> persons = new List<Person>();
…
// Alle Personen unter 30 Jahren im
// ABFRAGE-SYNTAX
var queryStmt = from person in persons
                where person.Age < 30
                select person;
```

```
List<Person> persons = new List<Person>();
…
// Alle Personen unter 30 Jahren im
// ERWEITERUNGSMETHODEN-SYNTAX
var exprMethStmt = persons.Where(x => x.Age < 30)
                          .Select(x=>x);
```

- Der (nullbasierte) Index der Elemente der Quellsequenz kann, dank Überladung des Where-Operators. im predicate-delegate ebenfalls genutzt werden vgl. auch [Marguerie et al. 2008, S. 156]
- Überladung nur im Methoden-Syntax möglich

```
List<Person> persons = new List<Person>();
...
 // Personen unter 30 Jahren
//  am Index 0...4 (wenn es welche gibt)
//  ERWEITERUNGSMETHODEN-SYNTAX
   var exprMethStmt2 = persons.Where((x, index) => x.Age < 30 && index < 5).Select(x=>x);
```

```
▲ 2 of 2 ▼ (extension) IEnumerable<Person> IEnumerable<Person>.Where<Person>(Func<Person, int, bool> predicate)
Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.
predicate: A function to test each source element for a condition; the second parameter of the function represents the index of the source element.
```

16.3.10 AUSFÜHREN DER ABFRAGE

1. **Datenquelle** (engl. data source) **auswählen**
2. **LINQ-Abfrage** auf den Datenelementen **erstellen/spezifizieren**

☝ Bisher haben wir nur die Punkte 1 und 2 betrachtet

3. **Ausführung** der Abfrage

- Mit einer Abfragevariable hat man zunächst nur die Möglichkeit der Ausführung einer Abfrage vgl. [Marguerie et al. 2008, S. 108]
- Die eigentliche Ausführung der Abfrage erfolgt erst, wenn die Abfragevariable in einer foreach-Anweisung durchlaufen wird → **Verzögerte Ausführung** vgl. [msdn_7]
- **Sofortige Ausführung der Abfrage kann erzwungen werden**
  - Abfrage wird sofort ausgeführt, wenn **ToArray()** oder **ToList()** aufgerufen werden, die diese Methoden Instanzen zurückgeben, die alle Ergebnisse der Abfrage enthalten
  - Der Aufruf einiger anderer Operatoren (**OrderByDescending**, **Count**, **Min**, …) bewirkt ebenfalls eine sofortige Ausführung, da diese die komplette Datenquelle benötigen vgl. [Marguerie et al. 2008, S. 114, S. 208]
    - Logisch, um alle Elemente zu zählen, zu ordnen oder den Minimalwert zu bestimmen werden alle Elemente benötigt

😀 Vorteile:

- **Trennung von Definition und Ausführung der Abfrage** ermöglichen:
  - beliebig **wiederholbare Ausführung** einer einmal definierten Abfrage
  - Änderungen in der Quellsequenz, auf der die Abfrage operiert, werden bei (erneuter) Ausführung stets mit berücksichtigt
- Der Standard-Abfrageoperator bzw. die Erweiterungsmethode **Select ist ein Iterator**
  - Im Gegensatz zu einem Array oder einer Liste müssen **nicht alle Ergebnisse auf einmal in den Speicher** geladen werden vgl. [Marguerie et al. 2008, S. 108ff]

😒 Nachteile:

- Kein Caching, die Abfrage wird jedes Mal erneut ausgeführt, wenn auf die Ergebnismenge zugegriffen wird

BEISPIEL 1

```
List<Person> persons = new List<Person>();
…
// Alle Personen unter 30 Jahren im
// ABFRAGE-SYNTAX
var queryStmt =  from person in persons
                 where person.Age < 30
                 select person;
…
 Console.WriteLine("UNTER 30 sind: ");
// Ausführung erfolgt erst jetzt
foreach (var item in queryStmt) {
        Console.WriteLine(item.Name + "(" + item.Age + ")");
 }
```

```
// foreach wird zu...
var enumerator = queryStmt.GetEnumerator();

while (enumerator.MoveNext()) {
    Person item = enumerator.Current;
    Console.WriteLine(item.Name + "(" + item.Age + ")");
}
```

BEISPIEL 2

- Können all diese unterschiedlichen Datenquellen mit nur jeweils einer Implementierung der (Standard)-Abfrageoperatoren abgefragt werden?

```
var queryStmt =  from person in persons
                 where person.Age < 30
                 select person;
```

⇔

```
var exprMethStmt = persons.Where(x => x.Age < 30).Select(x=>x);
```

Unterschiedliche Datenquellen

Objekte | XML-Dokumente | RDBMS | Excel-Dokumente | Web-Service

→ Funktioniert SELECT immer oder braucht jede Datenquelle ein eigenes SELECT?
NEIN!
Abfragesyntax ist nicht fest mit den entsprechenden Standardabfrageoperatoren verdrahtet.
Es wird eine Zwischenschicht benötigt, die zwischen LINQ abfrage und Datenquelle liegt (→ LINQ-Provider)

16.3.11 GROUPBY

- Gruppierungsoperator
- Gruppenbildung basiert auf Gruppierungsbedingung bzw. Schlüssel(wert)/key value
- Gruppierungsbedingung legt fest, ob ein Element zur Gruppe gehört oder nicht
- Ergebnis der Gruppierung ist IEnumerable von Gruppen (Gruppe wird repräsentiert durch Interface IGrouping, welches ebenfalls IEnumerable implementiert)

LINQ-Abfrage im Abfrage-Syntax:

- beginnt mit **from**
- endet mit **select** oder **group by**
- **into** (optional) verbindet Abfragen
  - Ergebnisse einer Abfrage dienen als Input für eine andere Abfrage (query continuation)

Allgemeine Methodensignatur (es gibt noch mehr Überladungen von GroupBy):
public static IEnumerable<IGrouping<TKey, TSource>>**GroupBy**<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> selector);

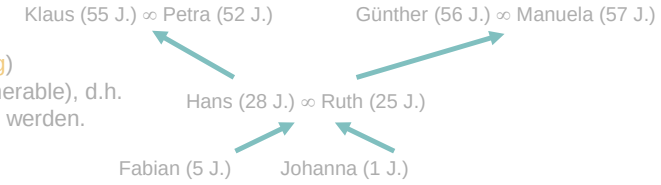
→ **„Groups the elements of a sequence according to a specified key selector function."** [msdn_8]

(KEY)SELEKTOR

```
List<Person> persons = new List<Person>();
…
//  Personen gruppiert nach Geschlecht
// ABFRAGE-SYNTAX
var queryStmt = from person in persons
                group person by person.Gender;
```

```
List<Person> persons = new List<Person>();
…
// Personen gruppiert nach Geschlecht
// ERWEITERUNGSMETHODEN-SYNTAX
// GroupBy SELEKTOR(funktion) wird auf jedes Element der
// Quellsequenz (Person) angewendet
var exprMethStmt = persons.GroupBy(person => person.Gender);
```

- Gruppierungsbedingung
- Gruppe (repräsentiert durch Interface IGrouping) ist aufzählbar (wg. Implementierung von IEnumerable), d.h. es kann über jedes Element der Gruppe iteriert werden.

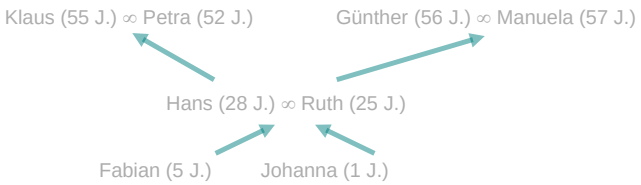



```
List<Person> persons = new List<Person>();
…
//  Personen gruppiert nach Geschlecht
// ABFRAGE-SYNTAX
var queryStmt = from person in persons
                group person by person.Gender;
…

// Ausführung der Abfrage
foreach (var group in queryStmt) {
        // (Gruppen)Schlüssel (engl. Key) ist hier Geschlecht (male oder female)
        Console.WriteLine(group.Key);
        foreach (var person in group)
        {
            Console.WriteLine(person.Name);
        }
}
```

Elemente gehören zur gleichen Gruppe, wenn für die Gruppierungsbedingung der gleiche Wert zurückgeliefert wird vgl. [Doberenz, Gewinnus 2008, S. 60], [mssqltips]

male	female
Klaus	Petra
Günther	Manuela
Hans	Ruth
Fabian	Johanna

BEISPIEL



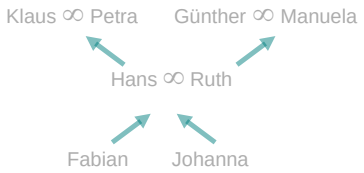

```
List<Person> persons = new List<Person>();
…
// ABFRAGE-SYNTAX
var queryStmt = from person in persons
                // Gruppieren nach Wahrheitswert
                // „Grouping by bool"
                group person by person.Age < 18;
```

Welche Gruppen werden gebildet und welche Elemente enthalten die Gruppen?

16.3.11.1 COMPOSITE-KEYS

- Gruppieren nach mehreren Kriterien ist mittels anonymer Typen möglich

  → „ **Group keys can** be any type… or a user-defined named type or **anonymous type**… Use a composite key when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element." [msdn_9]

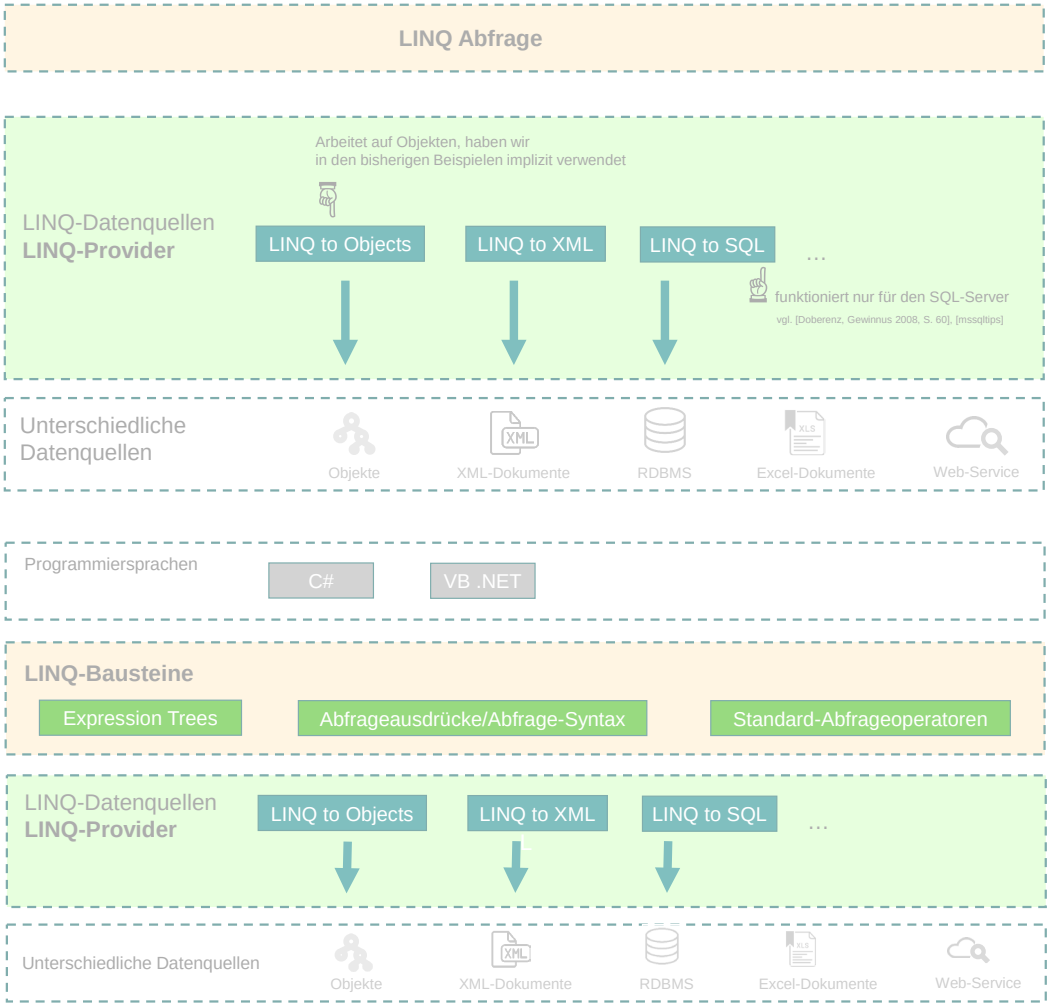



```
List<Person> persons = new List<Person>();
…
//  Personen gruppiert nach
// Geschlecht und Anzahl d. Kinder
// ABFRAGE-SYNTAX
var queryStmt = from person in persons
                group person by new { person.Children.Count, person.Gender };
…
```

new erzeugt anonymen Typ mit Property „Count" und Property „Gender"

{ Count = 1, Gender = male }	{ Count = 2, Gender = male }	{ Count = 0, Gender = male }
Klaus	Hans	Fabian
Günther		
{ Count = 1, Gender = female }	**{ Count = 2, Gender = female }**	**{ Count = 0, Gender = female }**
Petra	Ruth	Johanna
Manuela		

16.4 LINQ-PROVIDER/IMPLEMENTIERUNGEN/FLAVORS



16.5 WEITERES

16.5.1 ABFRAGEOPERATOREN

- Where (Einschränkungs-/Filteroperator)
- Projektionsoperatoren
  - Select
  - SelectMany
- Sortieroperatoren
  - OrderBy
  - OrderByDescending
  - Then By
  - ThenByDescending
  - Reverse
- Gruppierungsoperatoren
  - GroupBy
- Verknüpfungsoperatoren
  - Join
  - GroupJoin
- Mengenoperatoren
  - Distinct
  - Union
  - Intersect
  - Except
- Aggregierungsoperatoren
  - Count
  - LongCount
  - Sum
  - Min
  - Max
  - Average
  - Aggregate
- Generierungsoperatoren
  - Range
  - Repeat
  - Empty
- Quantifizierungsoperatoren
  - Any
  - All
  - Contains
- Partitionierungsoperatoren
  - Take
  - TakeWhile
  - Skip
  - SkipWhile
- Elementoperatoren
  - First
  - FirstOrDefault
  - Last
  - LastOrDefault
  - Single
  - SingleOrDefault
  - ElementAt
  - ElementAtDefault
  - DefaultIfEmpty
- Konvertierungsoperatoren
  - AsEnumerable
  - ToArray
  - ToList
  - ToDictionary
  - ToLookup
  - OfType
  - Cast
- Sonstige
  - Concat
  - SequenceEqual

16.5.2 ANWENDUNGSBEISPIEL

```
    // Angelehnt an die Beispiele in [Marguerie et al. 2008, S. 440ff, S. 443ff]
    public static class HTWLinqProvider
    {

        public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector)
        {
            Console.WriteLine("Aufruf von eigenem Select...");
            Console.WriteLine("...macht nichts weiter...");
            Console.WriteLine("...als den echten Standard-Operator Select aufzurufen...");
            return Enumerable.Select(source, selector);
        }


        // Abfrageoperator über einem bestimmten Typ
        public static IEnumerable<TResult> Select<TResult>(this IEnumerable<int> source, Func<int, TResult> selector)
        {
            Console.WriteLine("Ich mache nichts weiter mit den Zahlen. Ich gebe diese aus...");
            foreach (var item in source)
            {
                Console.WriteLine("Zahl: "+item);
            }
            Console.WriteLine("und rufe Select auf...");
            return Enumerable.Select(source, selector);
        }

    }

// ... in main-Methode

var intTest = from x in new int[] { 5, 6, 7, 8 }
              select x;

var stringTest = from x in new string[] { "Hallo", "du", "schöne", "Welt" }
                 select x;

        foreach (var item in intTest)
        {
            Console.WriteLine("int ist: "+item);
        }

        foreach (var item in stringTest)
        {
            Console.WriteLine("string ist: " + item);
        }
```

```
Ich mache nichts weiter mit den Zahlen. Ich gebe diese aus...
Zahl: 5
Zahl: 6
Zahl: 7
Zahl: 8
und rufe Select auf...
Aufruf von eigenem Select...
...macht nichts weiter...
...als den echten Standard-Operator Select aufzurufen...
int ist: 5
int ist: 6
int ist: 7
int ist: 8
string ist: Hallo
string ist: du
string ist: schöne
string ist: Welt
Drücken Sie eine beliebige Taste . . .
```

17 WIEDERHOLUNG

Vorlesung
07.07.2017

17.1 LAMBDA EXPRESSIONS

Zur Veranschaulichung:

```
delegate double myDel(double x, int i);
//...
myDel myMethodWithLambda = (x,i) => x + i;
double result = myMethodWithLambda(1.2, 2);
```

Oder:

```
double[] x = { 1, 2, 5, 3, 7, 4 };
x = x.OrderBy((xi) => xi).ToArray();
// Orderby sortiert, aber weiß nicht wie. Mit der Lambda-Expr.
   teilen wir es ihr mit, dass nur die Zahl als solche übergibt.
// Alternativ zum Beispiel rückwärts sortiert:
x = x.OrderBy((xi) => -xi).ToArray();
// auch noch komplizierter (erst gerade, dann ungerade Zahlen):
x = x.OrderBy((xi) => {
  if (xi%2 == 0)
    return 0
  else
    return 1
}).ToArray();
// ohne Lambda, mit vorher definierter Methode:
x = x.OrderBy(MySortMethod).ToArray();
```

Zusammenfassung:

```
// Lambda-Expression ohne Übergabewert:
() => 0;
// einfache Lambda-Expression:
(x,y,z) => 0;
// komplexere Lambda-Expression:
(x,y,z) => { /* ... */ }
```

Lambda Expressions sind sinnvoll, wenn Methode nur einmalig / an einer Stelle benötigt wird – braucht/hat keinen Methodennamen.

ZUGRIFF AUF LOKALE VARIABLE

```
static double MySortMethod(double xi){
  return xi;  // xi*i nicht möglich!
}

i = 5;
```

```
x = x.OrderBy((xi) => {
  if (xi%i == 0)
    return 0
  else
    return 1
}).ToArray(); // Zugriff auf lokales i möglich
x = x.OrderBy(MySortMethod).ToArray();  // Kein Zugriff auf lokales
    i möglich

// Der Zustand der Variable bleibt aktuell:
for (int i=0; i<2; i++){
  Task t = new Task(() => { Console.WriteLine(i); }); // diese Line
      wird wahrscheinlich erst gestartet, wenn Schleife schon lä
     ngst beendet ist (da Task ggf. initialisiert werden muss o.ä.)
      Ausgabe ist also nicht, wie erwartet "0 1", sondern "2 2"
  t.Start();
}
// Soll gezählt werden, müsste folgendes gemacht werden:
for (int i=0; i<2; i++){
  k = i;  // i wird explizit gefangen.
  Task t = new Task(() => { Console.WriteLine(k); });
  t.Start();
} // Ausgabe nun "0 1"
```